Université de Paris

École Doctorale de Sciences Mathématiques de Paris Centre - 386

Institut de Recherche en Informatique Fondamentale (IRIF)

Automated Verification of Programs Running on top of Distributed Systems

Par: Sidi Mohamed Beillahi

Thèse de doctorat d'informatique

Dirigée par: Ahmed Bouajjani

Et par: Constantin Enea

Présentée et soutenue publiquement le 15 Mars 2021

Devant un jury composé de :

Mihaela Sighireanu	Professeure	École Normale Supérieure de Paris-Saclay	Présidente
Ahmed Bouajjani	Professeur	Université de Paris	Directeur
Constantin Enea	Maitre de conférences (HDR)	Université de Paris	Co-directeur
Aarti Gupta	Professeure	Princeton University	Rapporteur
Roland Meyer	Professeur	Technical University of Braunschweig	Rapporteur
Parosh Aziz Abdulla	Professeur	Uppsala University	Examinateur
Viktor Vafeiadis	Tenured researcher	Max Planck Institute for Software Systems	Examinateur

Contents

Abstract

Over the past decades, distributed software became an integral part of our society, being used in various domains like online banking or shopping, distance learning, supply chain, and telecommuting. Developing correct and efficient distributed systems is a major and timely challenge. The objective of this dissertation is to propose algorithmic techniques for improving the reliability of such software, focusing on applications ran on top of distributed storage systems like *databases* and *blockchain*. Databases allow applications to access data concurrently from multiple sites in a network. Blockchain is a cryptographically-secure distributed ledger that allows to perform irreversible actions between different parties without a trusted authority.

The effect of a set of database transactions executing in parallel is specified using a formalism called consistency model. For instance, *serializability* states that a set of transactions behave as if they were executed serially one after another even if they actually overlap in time. Although simple to understand, serializability carries a significant penalty on performance and modern databases implement *weaker consistency* models. In general, these weak models are more complex to reason about. In this dissertation, we investigate the problem of checking a property of applications called *robustness*. Given two comparable consistency models, an application is called robust if it has the same behaviors when ran on top of databases implementing these two models. This dissertation investigates the theoretical complexity of checking robustness in the context of several consistency models: causal consistency, prefix consistency, snapshot isolation, and serializability. It provides non-trivial reductions to a well-studied problem in formal verification, assertion checking, that enables the reuse of existing verification technology. Besides theoretical results, it proposes pragmatic approaches based on under/over-approximations that are evaluated on practical applications.

Applications ran on top of blockchain are deployed in the form of *smart contracts* that manipulate the blockchain state. Smart contracts are mainly used to govern trading in cryptoassets that are worth billions of US dollars, and bugs can lead to huge financial losses. Exacerbating the impact of these bugs is the fact that smart contracts cannot be modified once they are deployed on the blockchain. Applying techniques from formal verification to audit smart contracts can help in avoiding expensive bugs. However, since most smart contracts are not annotated with formal specifications, formal verification of functional properties is impeded. To overcome this problem, this dissertation investigates notions of *refinement* between smart contracts, which enable the reuse of verified contracts as specifications for other contracts, thus scaling up the overall verification effort.

Keywords: Distributed systems, Databases, Blockchain, Smart contracts, Concurrency, Verification, Model checking, Program analysis, Program synthesis.

Résumé

Au cours des dernières décennies, les logiciels distribués ont pris une place centrale dans notre société. Ils sont utilisés dans divers domaines tels que la gestion des transactions bancaires et des achats en ligne, le télétravail, et l'enseignement à distance. Développer des logiciels distribués corrects et efficaces est un défi majeur. L'objectif de cette thèse est de proposer des techniques algorithmiques pour améliorer la fiabilité de ces logiciels, en se concentrant sur les applications logiciels qui s'exécutent au-dessus des systèmes de stockage distribués comme les bases de données ou la blockchain. Les bases de données permettent à des applications d'accéder simultanément aux données grâce à plusieurs sites répartis sur un réseau. La blockchain est un registre de stockage distribué et sécurisé par des techniques cryptographiques qui permet d'effectuer des tâches irréversibles entre différentes entités sans autorité de confiance centrale.

L'exécution en parallèle d'un ensemble de transactions sur des bases de données est spécifiée à l'aide d'un formalisme appelé le modèle de cohérence. Par exemple, le modèle de sérialisabilité indique qu'un ensemble de transactions se comporte comme si elles étaient exécutées en série l'un après l'autre, même si elles se chevauchent dans le temps. Bien que simple à comprendre, la sérialisabilité entraîne une pénalité significative en terme de performance. Pour cette raison les bases de données modernes mettent en oeuvre des modèles de cohérence plus faibles. En général, il est plus complexe de mener des raisonnement sur ces modèles faibles. Dans cette thèse, nous étudions le problème de la vérification d'une propriété des applications logiciels qui s'exécutent au-dessus des bases de données appelée la robustesse. Étant donné deux modèles de cohérence comparables, une application est dite robuste si elle a le même comportement lorsqu'elle est exécutée sur deux bases de données mettant en oeuvre les deux modèles de cohérence. Dans cette thèse, nous étudions la complexité théorique de la vérification de la robustesse dans le contexte de plusieurs modèles de cohérence: causal consistency, prefix consistency, snapshot isolation, et la sérialisabilité. Nous donnons des réductions non triviales à un problème bien étudié dans la littérature de la vérification formelle, la vérification des assertions, qui permet la réutilisation des technologies de vérification existantes. Outre des résultats théoriques, nous proposont aussi des approches basées sur des sous/sur-approximations que nous évaluons sur des applications pratiques.

Les applications logiciels exécutées au-dessus de la blockchain sont déployées sous la forme de smart contracts qui manipulent l'état de la blockchain. Les smart contracts sont principalement utilisés pour des operations basées sur des crypto-monnaies valant plusieurs milliards de dollars.

Par conséquent, des erreurs dans les smart contracts peuvent entraîner d'énormes pertes financières. Ces erreurs sont exacerbées par le fait que les smart contracts ne peuvent pas être modifiés une fois qu'ils sont déployés sur la blockchain. L'application des techniques de la vérification formelle pour auditer les smart contracts peut aider à éviter des erreurs coûteuses. Cependant, comme la plupart des smart contracts ne sont pas annotés avec leurs spécifications, la vérification formelle des propriétés fonctionnelles est entravée. Pour surmonter ce problème, nous explorons dans cette thèse les notions de raffinement entre smart contracts, qui permettent la réutilisation des smart contracts vérifiés comme spécifications pour d'autres smart contracts, améliorant ainsi l'effort global de vérification.

Mots clefs: Systèmes distribués, Base de données, Blockchain, Smart contracts, Concurrence, Vérification, Vérification de modèles, Analyse de programmes, Synthèse de programmes.

Acknowledgments

First and foremost, the work in this dissertation is the result of many supportive and great advisors, teachers, mentors, colleagues, family, and friends who accompanied me through my studies and showed me the values to pursue in both research and personal life.

I would like to express my deep gratitude to my advisors Ahmed Bouajjani and Constantin Enea. They taught me countless valuable lessons about research, being a good researcher, and to always challenge myself. Throughout my dissertation, Ahmed and Constantin were always there, even during a pandemic, to support and guide me. Thank you very much.

I would like to thank Michael Emmi for giving me the opportunity to work with him at SRI International. Michael is a great advisor, working with him is like winning the jackpot of valuable lessons for any student.

I would like to express my deep appreciation to Parosh Aziz Abdulla, Aarti Gupta, Roland Meyer, Mihaela Sighireanu, and Viktor Vafeiadis for agreeing to be the jury of my dissertation. Aarti and Roland were very kind to review my dissertation during this challenging time and to accommodate the delay in the completion of the dissertation writing. Thank you all very much.

I would lik to thank my colleagues at IRIF, in particular Giovanni Bernardi, Ranadeep Biswas, Berk Cirisci, German Andres Delbianco, Yassine Hamoudi, Suha Orhun Mutluergil, and Rachid Zennou, and during my internship at SRI International, in particular Manuel Carrasco, Giabriella Ciocarlie, Ákos Hajdu, and Dejan Jovanoviċ, for their conviviality. I would also like to thank the administrative staff at ED 386, IRIF, and UFR Informatique for their help during my dissertation.

Last but not the least, I would like to thank my family and friends for their support and understanding. Doing all my university studies far from home is a quite life journey full of fortunate and unfortunate coincidences, joyful and triumphant moments, and painful sacrifices and losses.

Chapter 1

Introduction

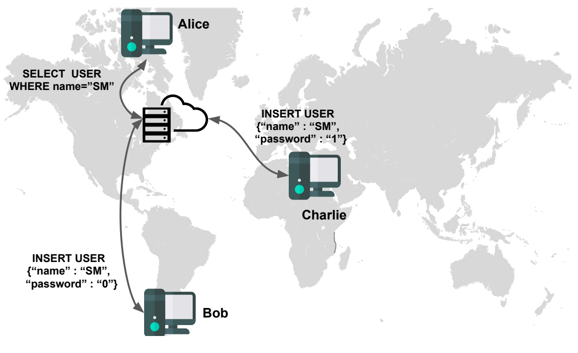
1.1 Distributed Software

Over the past decades, distributed software became an integral part of our society, being used in various domains like online banking or shopping, distance learning, and telecommuting. With this increased reliance, the demand for reliable and secure distributed software is large and it is growing. Hence, developing efficient and correct distributed software is a major and timely challenge. Databases and Blockchain are two commonly used distributed software systems.

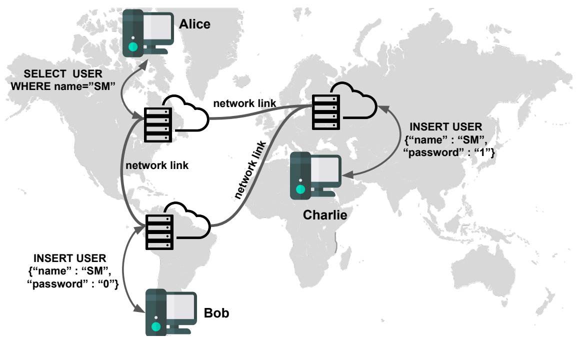
Modern production databases are distributed over multiple sites in a network. This is because many applications that use these databases require fast and reliable access to the data from different regions of the world, which is impossible to satisfy if these databases are centralized. For instance, in Figure 1.1a, we show a subscription system, it is constituted of three clients from three distinct regions of the world connected to a single server. The clients manipulate the data stored on the sever, e.g., adding new users or querying existing users. The subscription suffers from performance and availability issues. In particular, the response time to the clients Bob and Charlie queries will be very slow because of the long distance between the server and these clients. Also, if the server stops responding then the whole system will crash. A distributed subscription is shown in Figure 1.1b, it addresses the above issues of the centralized subscription. It is constituted of three connected severs distributed over three sites and three clients, each client is connected to one server. The data is replicated across the three servers. Thus, by reducing the distance between the severs and clients, the response time is improved. Also, when one server stops responding the two other severs can replace it and the system continues functioning. However, the distributed subscription must address new consistency issues, caused by the fact that it is distributed, such as: (1) how can the servers coordinate to disallow the two clients Bob and Charlie from adding two users with the same name?, and (2) how often should the servers exchange data to ensure that the reply to Alice query is not obsolete? To faithfully address the above questions, distributed databases implement complex network communication and memory access protocols. Because of this complexity, implementing correct and reliable applications that use these databases is difficult.

A blockchain is a distributed storage system. Distribution and replication stimulate trust and resilience necessary for the blockchain to operate correctly. This is because blockchain allows to perform irreversible and verifiable actions between different parties without a trusted central authority. Concretely, a blockchain is constituted of a sequence of blocks, which holds a log of completed action records. It uses cryptographic mechanisms to enforce an append-only pattern, disallowing the deletion or alteration of blocks already added to the chain. Applications ran on top of blockchain are called *smart contracts*, and they allow to manipulate the blockchain state. Smart contracts are commonly used to govern trading in cryptoassets such as *Bitcoin* [1] and *Ether* [2] that are worth billions of US dollars, and bugs can lead to huge financial losses. Exacerbating the impact of these bugs is the fact that smart contracts cannot be modified once they are deployed on the blockchain. Two famous smart contracts exploitations are TheDAO and the Parity wallet bugs that caused a combined loss of $240 million USD.

The objective of this dissertation is to propose algorithmic techniques for improving the reliability of applications ran on top of distributed storage systems like databases and blockchain.


Alice
SELECT USER WHERE name="SM"
INSERT USER {"name" : "SM", "password" : "1"}
Charlie
INSERT USER {"name" : "SM", "password" : "0"}
Bob

(a) A centralized subscription system.

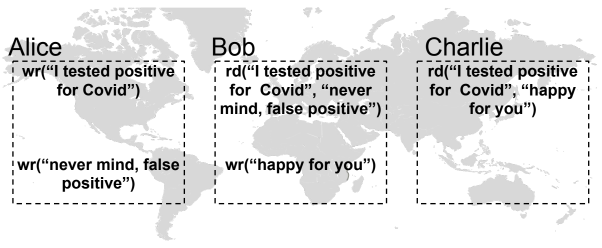

Alice
SELECT USER WHERE name="SM"
network link
network link
network link
INSERT USER {"name" : "SM", "password" : "1"}
Charlie
INSERT USER {"name" : "SM", "password" : "0"}
Bob

(b) A distributed subscription system.

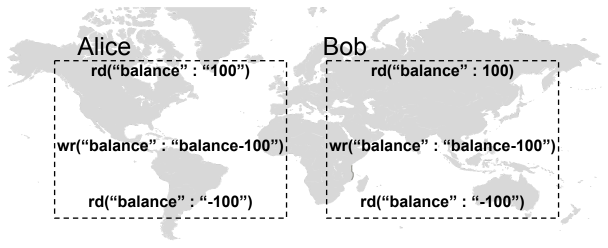
Figure 1.1: A subscription system.

1.2 Databases

Modern databases manage complex workloads for various applications, e.g., cloud storage, cloud computing, e-commerce, finances, and healthcare. Distribution and replication are widely adopted by databases providers to increase performance and tolerate failures of some database sites. Under these settings, modern databases implement different kinds of software optimizations (e.g., changing data structures and communication and consensus protocols) to properly and efficiently execute database operations. For instance, new noSQL (no-relational) databases such as key-value stores have been proposed to optimize data accesses. MongoDB and Cassandra are two popular noSQL databases. Also, distributed databases implement complex mechanisms to resolve conflicting updates stored on different database sites and to update a stale value stored in a database site by the latest value stored in another site. To ease the burden on programmers of applications that use these databases, the implementations of these databases must ensure consistency guarantees allowing to reason about their behaviors in an abstract and simple way.


Alice
wr("I tested positive for Covid")
wr("never mind, false positive")
Bob
rd("I tested positive for Covid", "never mind, false positive")
wr("happy for you")
Charlie
rd("I tested positive for Covid", "happy for you")

(a) Chat room.



(b) Bank.

Figure 1.2: Applications running on top of a causally consistent database.

1.2.1 Consistency

A database consistency model consists of a set of rules specifying the interaction between the database and the applications that use the database. Ideally, programmers of these applications would like to have strong consistency guarantees, i.e., all updates occurring anywhere in the system are seen immediately and executed in the same order by all the database sites. The strongest consistency level is *sequential consistency* [117], i.e., every computation of a program is equivalent to another one where operations (read or write) are executed atomically and sequentially one after another without interference by all sites. However, while sequential consistency is easier to apprehend by application programmers, their enforcement (by databases implementors) requires the use of global synchronization between all sites, which is hard to achieve while ensuring availability and acceptable performances [84, 90]. The fact that distributed database systems cannot ensure both availability and strong consistency is known as the consistency, availability, partition tolerance (CAP) theorem in the literature of distributed systems [90]. Modern databases often provide weaker consistency guarantees choosing availability over strong consistency. *Causal consistency* [116] is a fundamental weak consistency model implemented in several production databases, e.g., AntidoteDB, CockroachDB, and MongoDB, and extensively studied in the literature [31, 77, 123, 124, 144]. Causal consistency guarantees that every two operations where the second operation causally depends on the first operation, say the first operation affects the outcome of the second operation, are executed in the same order by all sites. For instance, causal consistency disallows the kind of execution of a chat room application shown in Figure 1.2a. This is because the write done by Bob causally depend on the last write done by Alice, however, Charlie observes the Bob's write without observing the last write done by Alice. In comparison to sequential consistency, causal consistency allows that conflicting operations, i.e., which read or write to a common location, be executed in different orders by different sites as long as they are not causally related. For instance, causal consistency allows the kind of execution of a bank application shown in Figure 1.2b. The execution shows two people Alice and Bob who share the same bank account withdrawing a total amount that leads to a negative balance. Sequential consistency disallows this execution since one of withdraws must occur after the other one finishes and thus will be rejected because of insufficient balance.

1.2.2 Transactions

Transaction is an abstract mechanism that refers to a block of operations (writes and reads) of a site can be considered as executing atomically without interferences from actions of other sites. A transaction ensures that either the entire block of operations is applied or non of the operations is applied. Modern databases provide transactions with various semantics corresponding to different tradeoffs between consistency and availability. *Serializability* [140] is the sequential consistency equivalent strong consistency model in the transactional setting, i.e., every computation of a program is equivalent to another one where transactions are executed serially one after another without interference. In the transactional setting, causal consistency refers to the same consistency guarantees as causal consistency in the non-transactional setting. Another popular weak consistency model in the transactional setting is *snapshot isolation* [46]. It is implemented in several production databases, e.g., Microsoft SQL Database, Oracle, and PostgreSQL, and it is extensively studied in the literature [83, 47, 129, 145, 50]. Compared to causal consistency, snapshot isolation further requires that transactions follow a total order, called *commit order*, such that each transaction observes all the updates in a prefix of this sequence and two different transactions observe different prefixes if they both write to a common location. Thus, snapshot isolation disallows the class of anomalies caused by two conflicting writes done concurrently. Two transactions may observe the *same* prefix as long as they do not write to a common variable, which again may lead to behaviors which are not admitted by serializability.

1.2.3 Correctness and Robustness

Implementing programs that run on top of databases and are both highly performant and correct with respect to their safety specifications is an extremely hard and error prone task. Checking correctness of programs with regard to preserving their safety specifications requires computing the set of reachable states. However, computing the set of reachable states under the weak semantics models is in general a hard problem (either decidable but highly complex (non-primitive recursive), undecidable, or unknown) [37, 38, 23, 111]. An alternative approach is to check *robustness* of programs against consistency relaxations: Given a program $\mathcal{P}$ and two consistency models S and W such that S is stronger than W, we say that $\mathcal{P}$ is robust against substituting S with W if for every two implementations I_S and I_W of S and W respectively, the set of observable behaviors (e.g., traces of computations[1] and reachable states) of $\mathcal{P}$ when running with I_S is the same as its set of observable behaviors when running with I_W. This means that $\mathcal{P}$ is not sensitive to the consistency relaxation from S to W, and therefore it is possible to reason about the behaviors of $\mathcal{P}$ assuming that it is running over S, and no additional synchronization is required when $\mathcal{P}$ runs over the weak model W such that it maintains all its properties satisfied with S. Robustness implies that any safety specification of $\mathcal{P}$ is preserved when weakening the consistency model (from S to W). Thus, when the stronger model S corresponds to serialisability and $\mathcal{P}$ is robust then we can check whether $\mathcal{P}$ preserves its safety specifications under the weak consistency model W by checking whether it preserves them under serialisability. The latter is well studied and existing verification tools are well equipped to deal with it.

[1]A trace of computation captures the control and data dependencies between operations in the computation.

Robustness also allows: (1) to identify the *weakest* level of consistency needed by a given program (to satisfy its specification) since a weaker consistency model provides better performance, and (2) to ensure that the level of consistency needed by a given program coincides with the one that is guaranteed by its infrastructure, i.e., the database it uses. We assume that this database (including network communication and conflicts resolution protocols) satisfies its consistency guarantees.

While robustness based on reachable states, which requires that a program is robust if the sets of reachable states under the two consistency models coincide, is the necessary and sufficient concept for preserving safety specifications, its verification amounts to computing the set of reachable states under the weak semantics models which leads to the same problem as discussed above. A stronger notion of robustness that allows to overcome this problem is robustness based on the equivalence between the sets of traces of computations. For instance, in the context of shared memory it was shown in [51] that robustness against substituting sequential consistency with total store ordering (TSO) consistency, based on traces of computations, is as hard as reachability under sequential consistency, which is much simpler than reachability under TSO [37].

1.2.4 Problem Statement

In this dissertation, we investigate the problem of checking robustness of programs in the context of four consistency models: *serializability* (`SER`) , snapshot isolation (`SI`) , *prefix consistency* (`PC`) (another weak consistency model that is stronger than causal consistency and weaker than snapshot isolation) [61, 69], and *causal consistency* (`CC`) in a transactional setting. We focus on robustness based on the criterion of the equivalence between the sets of traces of computations. However, we need to identify the appropriate notions of traces of computations so that in almost all practical cases, programs that are robust based on reachable states criterion are also robust based on traces of computations criterion. Furthermore, checking robustness is also difficult since it requires to apprehend the extra behaviors due to the relaxed model w.r.t. the stronger model. This requires a priori reasoning about complex order constraints between operations in arbitrarily long computations, which may need maintaining unbounded ordered structures, and make robustness checking hard or even undecidable.

We consider two important robustness problems: (1) robustness against substituting the strong model (i.e., `SER`) with one of the weak consistency models, and (2) robustness against substituting a weak consistency model, e.g., `SI`, with a weaker one, e.g., `CC`. The result of the first problem allows to run programs over weakly consistent database while reasoning about the correctness of these programs assuming that they run over SER database. The result of the second problem allows to find the weakest consistency model while a program still satisfy its safety specifications. This is because there is a large class of specifications that can be implemented even in the presence of behaviors which are not admitted under SER (see [152] for a discussion).

1.2.5 State of the Art

There are several works that investigated the problem of verifying databases and shared memory correctness, i.e., checking whether a database or a shared memory indeed implements a consistency criteria it claims [103, 35, 53, 98, 52]. In addition, there are several other works that investigated the problem of databases and shared memories testing [140, 89, 137, 138, 50, 147]. In this dissertation, we assume that databases are correct and investigate the correctness of programs that run on top of these databases.

In the non-transactional case (i.e., shared memory and programming language memory models), existing work on the verification of robustness can be classified into two classes: (1) over- or under-approximate analyses [62, 136, 63, 29], and (2) precise (sound and complete) analyses [55, 51, 74, 113]. They consider robustness only when the stronger model (S) corresponds to strong consistency, i.e., sequential consistency. In the transactional case, all existing work on the verification of robustness provide either over- or under-approximate analyses [33, 47, 57, 58, 70, 129], but none of them provides precise algorithmic verification methods for solving robustness, nor addresses its decidability and complexity. Also, they consider robustness only when the stronger model (S) corresponds to strong consistency, i.e., SER. This dissertation is the first work studying the decidability and complexity of verifying robustness in the context of transactional programs. Also, this dissertation is the first work studying the decidability and complexity of verifying robustness where the stronger model is not strong consistency.

Because of the undecidability/high complexity of reachability problems under weakly consistent models, existing works for the analyses of programs correctness under these models without proving their robustness, e.g., [39, 28, 72, 24, 114, 93, 130, 27], do not provide decision procedures.

1.2.6 Contribution

In this dissertation, we show that the problems of checking robustness of application programs against substituting `SER` with `CC` or `SI`, `SI` with `PC`, and `PC` with `CC` can be reduced in polynomial time to the reachability problem in concurrent programs under `SER`. This allows: (1) to avoid explicit reasoning about weak consistency behaviors (since this may imply memorizing unbounded information), and (2) to leverage available tools for verifying invariants/reachability problems on concurrent programs to reason about distributed applications running on weakly consistent databases. This also implies that the robustness problem is decidable for finite-state programs, PSPACE-complete when the number of sites is fixed, and EXPSPACE-complete otherwise. The above reduction requires non-trivial results that characterize the behaviors of each of the considered weak consistency models such as finding the appropriate formal definitions of traces of computations for each individual consistency model so there are almost no programs in practice that distinguish between robustness based on traces of computations and robustness based on reachable states.

The approach we adopt for tackling the robustness problem is based on a precise characterization of the set of robustness violations, i.e., computations that are possible under the weaker model but not the stronger model. For instance, for robustness against substituting `SER` with `CC` or `SI` we show that it is sufficient to search for a special type of *minimal* robustness violations. We reuse the high-level methodology from [51] of characterizing minimal violations according to some measure. However, we use different notions of measure because of the semantics differences between individual weak consistency models. The key property we prove is that all minimal violations obey to a finite number of patterns. Moreover, using this characterization, we show that given a program P, deciding whether P is not robust can be done by exploring only serial computations: We consider a program P' obtained from P by a linear-size instrumentation. The latter maintains along serial computations of P' (where accesses to the main memory are done in a sequentially consistent way) the information needed to recognize the pattern of a minimal violation that would have occurred in P under the weaker semantics, i.e., `CC` or `SI`, (executing the same set of operations).

However, the above methodology based on minimal violations is applicable when the stronger model in robustness is the model desired for the reachability problem (`SER`) we reduce to. Thus, for robustness against substituting `SI` with `PC` and `PC` with `CC` we derive reductions to `SER` reachability using two completely different methodologies: characterizing violations of robustness against substituting `SI` with `PC` in terms of `SER` computations (of another program) and showing that violations of robustness against substituting PC with CC can be rewritten as violations of robustness against substituting SER with CC (of another program).

In the fist case we use a monitor that checks whether a behavior is admitted by a program P under PC, but not under SI, which raises two non-trivial challenges: (1) defining a monitor for detecting violations of robustness against substituting SI with PC that uses a minimal amount of auxiliary memory (to remember past events), and (2) what is the complexity of checking if the composition of P with the monitor reaches a specific control location (we assume that the monitor goes to a specific error location when detecting a violation) under the (weaker) model. Interestingly enough, we address these two challenges by studying the relationship between these two weak consistency models, PC and SI, and SER. The construction of the monitor is based on the fact that violations of robustness against substituting SI with PC can be defined as roughly, the difference between the violations of robustness against substituting SER with PC and SER with SI (investigated in previous work [47]), and we show that the reachability problem under PC can be reduced to a reachability problem under SER.

In the second case we rely on the reduction from PC reachability to SER reachability mentioned above. This reduction shows that a given program P reaches a certain control location under PC iff a transformed program P', where essentially, each transaction is split in two parts, one part containing all the reads, and one part containing all the writes, reaches the same control location under SER. Interestingly, P reaches a certain control location under CC if and only if P' reaches the same control location under CC. The latter may seem counter-intuitive since it is not true for PC. Thus, the fact that this reduction preserves the structure of the program allows to redefine violations of robustness against substituting PC with CC of a program P to violations of robustness against substituting SER with CC of the transformed program P'.

We study the robustness against substituting SER with three distinct semantics models of CC. We show that the three models coincide for programs containing no *write-write data races*, i.e., concurrent transactions writing on a common location. We also show that if a program has a write-write data race under one of these models, then it must have a write-write data race under any of the other two models. This property is rather counter-intuitive since two of these models are incomparable and the third model is strictly weaker than both of them (in terms of admitted behaviors).

We also developed a proof methodology for establishing robustness against substituting SER with

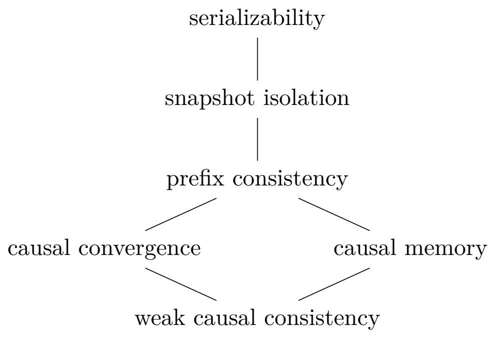


Figure 1.3: Consistency models.

`SI`, `SI` with `PC`, and `PC` with `CC` which builds on Lipton's reduction theory [120] and the characterization of robustness violations. We use the theory of movers to establish whether the relaxations allowed by weaker models are harmless, i.e., they do not introduce new behaviors compared to the stronger model. We applied the proposed techniques for checking robustness against substituting `SER` with `SI`, `SI` with `PC`, and `PC` with `CC` on 17 challenging applications extracted from previous work [76, 58, 161, 32, 105, 47, 93, 129]. Our techniques are precise enough for proving or disproving the robustness of all of these applications, for all combinations of the consistency models discussed above. In Figure 1.3, we list the consistency models we study in this dissertation.

1.3 Blockchain

Blockchain offers an innovative approach that allow to establish trust in an open environment without the need of a centralized authority to do so. Blockchain use cases range from globally deployed cryptoassets like *Bitcoin* [131], to supply chains [9], insurance [4], and banking [5]. Two popular blockchains are *Bitcoin* [1] and *Ethereum* [2] which have a combined market capitalization exceeding $575 billion USD. Blockchain systems rely on a *tamper-proof ledger*, i.e., no entity can delete or modify ledger entries once they have been recorded, that is distributed and replicated across a network of nodes. Ledger entries are organized as a sequence of blocks, each block records a set of completed actions that are called *transactions*. The ledger establishes which transactions happened (e.g., Alice transferred 2 ₿ coins to Bob as shown in Figure 1.4), and the order in which the transactions happened (e.g., Alice transferred 2 ₿ coins to Bob, and then Bob transferred 2 $ coins to Alice as shown in Figure 1.4). To ensure that a distributed ledger remains secure and accessible to all parties, blockchain systems implementors use consensus protocols to determine the state of the ledger (i.e., nodes agree on this unique state), and cryptographic functions to keep a cryptographic audit trail ensuring the ledger integrity.

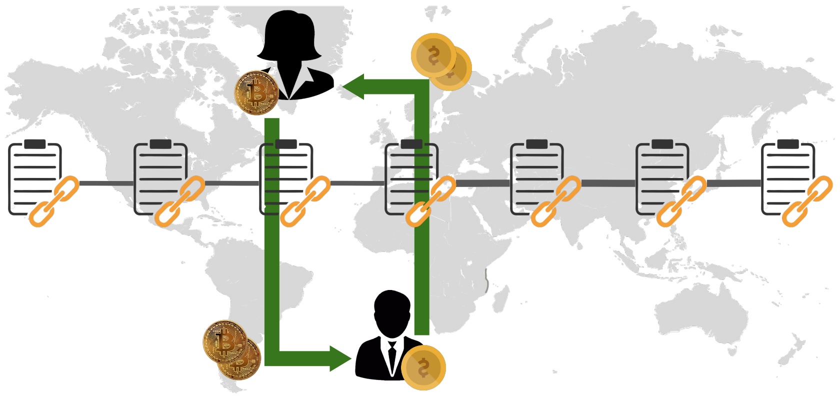


Figure 1.4: Blockchain

1.3.1 Smart Contracts

Several blockchains, most prominently Ethereum, allow the execution of application programs, called *smart contracts*, that are stored on the blockchain. As they offer autonomy for arbitrarily-complex transactions between multiple parties, smart contracts are already powering a sizable economy: applications include decentralized finance [22], supply chains [9], and insurance [4]. A smart contract resembles an object in an object-oriented programming language. It manages a permanent state stored on the blockchain. It is constituted of a set of functions that manipulate the state. Functions can be called either directly by users or indirectly by other smart contracts, through transactions. They allow to perform arbitrarily-complex operations using cryptoassets stored on the blockchain. A concept that distinguishes smart contracts from standard software programs is: a smart contract is immutable once it is deployed on the blockchain, i.e., upgrading a deployed contract is extremely difficult due to the design of blockchain. Solidity [21] is the most popular Turing-complete high-level programming language for smart contracts, which is designed to target the Ethereum Virtual Machine [2].

1.3.2 Correctness

Although blockchain and smart contracts have received growing interests in both academia and industry in the recent years, the security of blockchain and smart contracts continue to be at the center of the discussion when applying them in new applications. This is because of the many exploitations targeting blockchain and smart contracts that caused expensive losses. For instance, [101] shows the possibility of attacking a blockchain consensus protocol. In [40, 150, 107], the authors survey common bugs in smart contracts that caused big financial losses. Thus, it is important to ensure the correctness of both the blockchain infrastructures (e.g., consensus protocols) and the smart contracts that run on them before their deployment. In this dissertation, we focus on the verification of the correctness of smart contracts. We assume that smart contracts run on a blockchain infrastructure (including network communication, and the behavior of individual nodes) that satisfies its guarantees.

1.3.3 Problem Statement

Formal verification has the potential to mitigate against malicious exploitation of smart contracts. However, scaling verification efforts to a large number of smart contracts is an important challenge. In particular, while specifications that are *specialized* to each individual smart contract are useful for proving customized functional properties [141], *generic* specifications that can be applied to large classes of smart contracts would facilitate verifying contracts en masse. Ideally, the specification for a given class of smart contracts could be written once, and reused for the verification of each contract of that class.

Truly generic specifications must be sufficiently weak so that every correct contract in the given class adheres to its functional properties. Moreover, truly generic specifications must be independent from the state variables of any particular contract, since the state variables of other contracts in the same class generally differ in name, number, and type. Such generic specifications are however unsuited for existing verification tools like solc-verify [97], VeriSol [165] and VerX [141], which suppose that input contracts are annotated with expressions that refer to state variables, e.g., pre- and post-conditions. This poses a scalability problem since deriving such annotations for each contract from class-wide generic specifications would be a manual labor-intensive process.

In this dissertation, we address this scalability challenge and introduce an approach for verifying *unannotated* smart contracts via automated semantic comparison against *annotated* smart contracts. Our approach is motivated by the insight that many of the smart contracts instantiated on popular blockchains (e.g., Ethereum [166]) are variations on a relatively small number of canonical contracts and libraries implementing concepts like auction, escrow, tokens, and voting. Intuitively, many of these variations obey the principle of *substitutability* [122], meaning that they adhere to the functional properties captured by the annotations of their canonical counterparts.

1.3.4 State of the Art

Recently, blockchain and smart contracts attracted the interest of the formal verification community. In particular, several recent work, e.g., [143, 34, 56, 48, 119], use theorem provers such as Coq and TLA+ for the verification of the correctness of blockchain consensus protocols. In this dissertation, we assume the correctness of the blockchain infrastructure including the consensus protocols.

Existing work on smart contracts verification can be classified into two classes: (1) full automated techniques that require no manual intervention, and (2) semi-automated techniques. Full automated techniques, e.g., [94, 106, 159, 164, 100, 109, 125, 135, 160, 96], are designed to verify correctness for *bounded* executions or for particular security properties (e.g., integer overflows) and cannot establish full functional correctness. Semi-automated techniques rely on user-provided functional specifications to establish full functional correctness, e.g., [36, 49, 95, 104, 151, 97, 165, 141]. However, in this dissertation, we study the problem of verifying full functional correctness for smart contracts for which functional specifications do not exist.

1.3.5 Contribution

In this dissertation, we propose a technique for verifying *unannotated* smart contracts via automated semantic comparison against *annotated* canonical smart contracts. With a notion of comparison that implies substitutability, we can thus amortize the cost of manually annotating the canonical contracts by verifying a vast number of unannotated contracts. Our notion of *behavioral refinement* relates the input-output behavior of contracts' transactions, i.e., parameters and effects on storage, ignoring internal details like local memory and control flow. By proving that a given contract is a behavioral refinement of another, we guarantee the inheritance of behavioral properties, and in particular that the effects of any sequence of transactions obeys its canonical counterpart's functional properties.

Establishing behavioral refinement for unbounded transaction sequences relies on induction. Akin to inductive invariants for safety properties, proofs of behavioral refinement use induction hypotheses called *simulation relations* [127]. Essentially, a behavioral simulation relation identifies states of two contracts such that initial states are related; the same transaction applied to related states yields related states and identical effects; and related states are *observationally equivalent*, i.e., any function applied to both yields identical values.

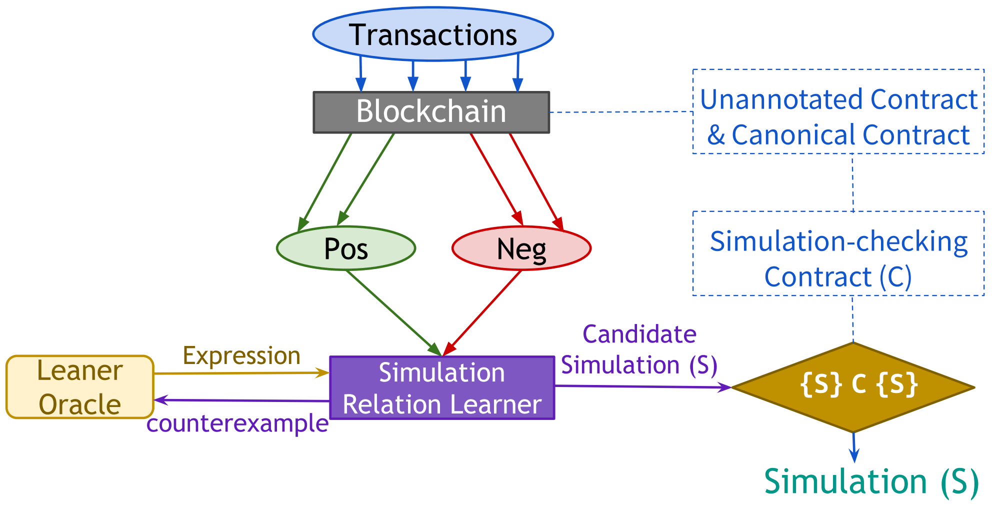


Figure 1.5: Behavioral simulation relations synthesis.

We develop an algorithm for synthesizing behavioral simulation relations. The algorithm is constituted of two steps: a passive learning step generates candidate simulation relations, and a deductive verification step checks the validity of each candidate. Candidate generation can be demand-driven according to *counterexample guided inductive synthesis* [157], e.g., initially proposing the trivial simulation relating each pair of contract states, and incrementally proposing candidates which rule out spurious counterexamples from prior validation steps.

To generate candidate simulation relations, we adopt a paradigm of learning from examples [128]. In this work, we consider only *passive* learning, which assumes that the set of examples is fixed a priori, as opposed to being generated on learner demand. In the context of simulation, an example is a pair of states, i.e., one state of each contract: positive examples are pairs of states which must be similar, and negative examples are pairs which must not be similar. To generate examples, we consider sequences of transactions, executed on a blockchain, starting from the initial states of each contract. Intuitively, negative examples correspond to pairs of states which yield distinct observations, and positive examples correspond to pairs of states reached by identical transaction sequences, unless such a pair yields distinct observations, in which case it is a counterexample to simulation. We then leverage off-the-shelf learning algorithms [139] by providing an oracle to evaluate candidate expressions against pairs of states, i.e., by executing such expressions on the blockchain.

To verify candidate simulation relations, we adopt a notion of product programs inspired by relational verification [43]. In particular, we generate an auxiliary *simulation-checking* contract whose verification implies the validity of a given simulation relation. Intuitively, for each function f of the given unannotated contract, the simulation-checking contract provides a function which executes f in lockstep with its canonical contract's counterpart. Besides asserting the equality of effects and return-values, this function includes the candidate simulation as pre- and post-conditions, ultimately implying inductiveness. We verify the simulation-checking contract using off-the-shelf Solidity smart contracts verifiers [97]. In Figure 1.5 we illustrate the process of generating candidate simulation relations and verifying that they are indeed true simulation relations.

Empirically, we validate our approach by collecting dozens of Solidity-language smart contracts, identifying canonical contracts for several classes, annotating and verifying these canonical contracts with precise formal specifications, and synthesizing simulation relations from multiple variations of each class. Our implementation can correctly synthesize nontrivial simulation relations for many classes, and integrates off-the-shelf tools for example-guided learning and Solidity smart contracts verification.

1.4 Thesis Outline

The rest of this dissertation is organized as follows:

- In Chapter 2, we introduce some concepts necessary for describing the dissertation contributions.
- In Chapter 3, we study the relation between three distinct semantics of `CC`. We then study the robustness against substituting `SER` with `CC` and show its reduction to the reachability problem in concurrent programs under `SER`.
- In Chapter 4, we study the robustness against substituting `SER` with `SI` and show its reduction to the reachability problem in concurrent programs under `SER`. We then present a proof methodology for establishing robustness. Finally, we present an experimental evaluation.
- In Chapter 5, we study the robustness against substituting `PC` with `CC` and show its reduction to the robustness against substituting `SER` with `CC`. We then present a reduction of the robustness

against substituting PC with SI to the reachability problem in concurrent programs under SER. Finally, we present an experimental evaluation.

- In Chapter 6, we demonstrate an application of behavioral simulation to smart contracts. We then develop an algorithm for synthesizing behavioral simulation relations. We develop a smart contract benchmark suite including variations of identified canonical contracts. We develop implementation of our approach. Finally, we evaluate our implementation, verifying functional properties for dozens of unannotated smart contracts.
- In Chapter 7, we summarize the contributions in the dissertation and discuss open problems and future research directions.

Chapter 2

Preliminary

2.1 Introduction

In this chapter, we present a set of formalisms that we use in Chapters 3, 4, and 5. In §2.2, we give the syntax of a simple programming language for application programs that we study in these chapters. Then, in §2.3, we present the program semantics setting. In particular, we formally define the serializability semantics. We also introduce the notions of serializable program executions and traces of serializable executions.

2.2 Program Syntax

We consider a simple programming language grammar which is defined in Figure 2.1. A program is parallel composition of *processes* distinguished using a set of identifiers $\mathbb{P}$. Each process is a sequence of *transactions* and each transaction is a sequence of *labeled instructions*. Each transaction starts with a `begin` instruction and finishes with a `commit` instruction. Each other instruction is either an assignment to a process-local *register* from a set $\mathbb{R}$ or to a *shared variable* from a set $\mathbb{V}$, or an `assume` statement. The read/write assignments use values from a data domain $\mathbb{D}$. An assignment to a register $\langle reg \rangle := \langle var \rangle$ is called a *read* of $\langle var \rangle$ and an assignment to a shared variable $\langle var \rangle := \langle reg\text{-}expr \rangle$ is called a *write* to $\langle var \rangle$ ($\langle reg\text{-}expr \rangle$ is an expression over registers whose syntax we leave unspecified since it is irrelevant for our development). The `assume` $\langle bexpr \rangle$ blocks the process if the Boolean expression $\langle bexpr \rangle$ over registers is false. It can be used to model conditionals. Each instruction is followed by a `goto` statement which defines the evolution of the

$$\begin{array}{lll}
\langle \mathit{prog} \rangle & ::= & \texttt{program}\ \langle \mathit{process} \rangle^* \\
\langle \mathit{process} \rangle & ::= & \texttt{process}\ \langle \mathit{pid} \rangle\ \texttt{regs}\ \langle \mathit{reg} \rangle^* \\
 & & \langle \mathit{ltxn} \rangle^* \\
\langle \mathit{ltxn} \rangle & ::= & \langle \mathit{binst} \rangle\ \langle \mathit{linst} \rangle^*\ \langle \mathit{einst} \rangle \\
\langle \mathit{binst} \rangle & ::= & \langle \mathit{label} \rangle\texttt{: begin; goto}\ \langle \mathit{label} \rangle\texttt{;} \\
\langle \mathit{einst} \rangle & ::= & \langle \mathit{label} \rangle\texttt{: end; goto}\ \langle \mathit{label} \rangle\texttt{;} \\
\langle \mathit{linst} \rangle & ::= & \langle \mathit{label} \rangle\texttt{:}\ \langle \mathit{inst} \rangle\texttt{; goto}\ \langle \mathit{label} \rangle\texttt{;} \\
\langle \mathit{inst} \rangle & ::= & \langle \mathit{reg} \rangle\ \texttt{:=}\ \langle \mathit{var} \rangle \\
 & | & \langle \mathit{var} \rangle\ \texttt{:=}\ \langle \mathit{reg\text{-}expr} \rangle \\
 & | & \texttt{assume}\ \langle \mathit{bexpr} \rangle
\end{array}$$

Figure 2.1: Program syntax. a^* indicates zero or more occurrences of a. $\langle \mathit{pid} \rangle$, $\langle \mathit{reg} \rangle$, $\langle \mathit{label} \rangle$, and $\langle \mathit{var} \rangle$ represent a process identifier, a register, a label, and a shared variable, respectively. $\langle \mathit{reg\text{-}expr} \rangle$ is an expression over registers while $\langle \mathit{bexpr} \rangle$ is a Boolean expression over registers.

program counter. Multiple instructions can be associated with the same label which allows us to write non-deterministic programs and multiple `goto` statements can direct the control to the same label which allows us to mimic imperative constructs like loops and conditionals. We assume that the control cannot pass from one transaction to another without going as expected through `begin` and `end` instructions.

To simplify the technical exposition of the thesis, programs contain a bounded number of processes and each process executes a bounded number of transactions. A transaction may execute an unbounded number of instructions but these instructions concern a bounded number of variables, which makes it impossible to model SQL (select/update) queries that may access tables with a statically unknown number of rows. The thesis contributions can be extended beyond these restrictions.

2.3 Program Semantics

We formally define the semantics of a program $\mathcal{P}$ under a consistency model X using an LTS. The states of the LTS are called configurations, and transitions between configurations are called execution steps. An execution of $\mathcal{P}$ is a run of the LTS.

2.3.1 Labeled Transition Systems

A labeled transition system (LTS) is defined as a tuple $(Q, \Sigma, I, F, \delta)$ where Q is the set of states, Σ is a set of alphabet labels, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of terminal (final) states, and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. A run in the LTS is a sequence of transitions $q_0 \xrightarrow{a_1} q_1 \cdots \xrightarrow{a_n} q_n$, where $q_0 \in I$, $q_n \in F$, and forall $0 \leq i < n$, $(q_i, a_{i+1}, q_{i+1}) \in \delta$. When there exists a run to some state $q \in Q$, we say that q is reachable.

2.3.2 Serializability Semantics

In the semantics of a program under serializability ($\mathtt{SER}$), a configuration keeps a single shared-variable valuation (accessed by all processes) with the standard interpretation of read and write statements. Also, a configuration keeps a global lock to ensure that each transaction executes in isolation. Each process has a local configuration to keep registers valuation and the label of next instruction to execute. In the initial configuration, each shared-variable is set to some initial value $\perp$. Only configurations where the global lock is set to 1 are considered final, meaning that the execution of a $\mathcal{P}$ can end at any time when no transaction is executing.

Formally, the semantics of a program $\mathcal{P}$ under serializability is defined using a LTS $[\mathcal{P}]_{\mathtt{CM}} = (\mathbb{C}, \mathbb{E}\mathsf{v}, \mathsf{gs}_0, \mathbb{C}_{\mathbb{F}}, \rightarrow)$ where $\mathbb{C}$ is the set of program configurations, $\mathbb{E}\mathsf{v}$ is the set of transition labels called *events*, gs_0 is the initial configuration, $\mathbb{C}_{\mathbb{F}}$ is the set of final program configurations, and $\rightarrow \subseteq \mathbb{C} \times \mathbb{E}\mathsf{v} \times \mathbb{C}$ is the transition relation. The set of events is defined by:

$$\mathbb{E}\mathsf{v} = \{\ \mathsf{begin}(p,t), \mathsf{ld}(p,t,x,v), \mathsf{isu}(p,t,x,v), \mathsf{com}(p,t) : p \in \mathbb{P}, t \in \mathbb{T}, x \in \mathbb{V}, v \in \mathbb{D}\}$$

where begin and com label transitions corresponding to the start, resp., the end of a transaction, isu and ld label transitions corresponding to writing, resp., reading, a shared variable during some transaction.

In Figure 2.2, we list the transition relation (execution steps) $\rightarrow$. The events labeling a transition are written on top of $\rightarrow$. A begin transition will just set the global lock to 0 to signal that a transaction is executing while a com transition will set the global lock to 1. The set of serializable executions of a program $\mathcal{P}$ is denoted by $\mathbb{E}\mathsf{x}_{\mathtt{SER}}(\mathcal{P})$.

$$\frac{\texttt{begin} \in \mathsf{inst}(\mathsf{ls}(p).\mathsf{pc}) \quad l = 0 \quad s = \mathsf{ls}(p)[\mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})]}{(\mathsf{ls}, l, \mathsf{store}) \xrightarrow{\mathsf{begin}(p,\, t)} (\mathsf{ls}[p \mapsto s], l \mapsto 1, \mathsf{store})}$$

$$\frac{r := x \in \mathsf{inst}(\mathsf{ls}(p).\mathsf{pc}) \quad rval = \mathsf{ls}(p).\mathsf{rval}[r \mapsto \mathsf{store}[x]] \quad s = \mathsf{ls}(p)[\mathsf{rval} \mapsto rval, \mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})]}{(\mathsf{ls}, l, \mathsf{store}) \xrightarrow{\mathsf{ld}(p,\, t,\, x,\, v)} (\mathsf{ls}[p \mapsto s], l, \mathsf{store})}$$

$$\frac{x := v \in \mathsf{inst}(\mathsf{ls}(p).\mathsf{pc}) \quad s = \mathsf{ls}(p)[\mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})]}{(\mathsf{ls}, l, \mathsf{store}) \xrightarrow{\mathsf{isu}(p,\, t,\, x,\, v)} (\mathsf{ls}[p \mapsto s], l, \mathsf{store}[x \mapsto v])}$$

$$\frac{\texttt{end} \in \mathsf{inst}(\mathsf{ls}(p).\mathsf{pc}) \quad l = 1 \quad s = \mathsf{ls}(p)[\mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})]}{(\mathsf{ls}, l, \mathsf{store}) \xrightarrow{\mathsf{com}(p,\, t)} (\mathsf{ls}[p \mapsto s], l \mapsto 0, \mathsf{store})}$$

Figure 2.2: The set of transition rules defining the serializability semantics. We assume that all the events which come from the same transaction use a unique transaction identifier t. For a function f, we use $f[a \mapsto b]$ to denote a function g such that $g(c) = f(c)$ for all $c \neq a$ and $g(a) = b$. The function inst returns the set of instructions labeled by some given label while next gives the next instruction to execute.

2.3.3 Traces of Serializable Executions

A *trace* abstracts the order in which shared-variables are accessed inside a transaction and the order between transactions accessing different variables. Formally, the trace of a serializable execution ρ is obtained by (1) replacing each sub-sequence of transitions in ρ corresponding to the same transaction with a single "atomic macro-event" (p, t), and (2) adding several standard relations between these atomic macro-events to record the data-flow in ρ, e.g. which transaction wrote the value read by another transaction. The sequence of (p, t) events obtained in the first step is called a *summary of* ρ. We say that a transaction t in ρ performs an *external read* of a variable x if ρ contains an event $\mathsf{ld}(p, t, x, v)$ which is not preceded by a write on x of t, i.e., an event $\mathsf{isu}(p, t, x, v)$. Also, we say that a transaction t *writes* a variable x if ρ contains an event $\mathsf{isu}(p, t, x, v)$, for some v.

The *trace* $\mathsf{tr}(\rho) = (\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$ of a serializable execution ρ consists of the summary τ of ρ along with the *program order* PO, which relates any two events (p, t) and (p, t') that occur in this order in τ, *write-read* relation WR (also called *read-from*), which relates any two events (p, t) and (p', t') that occur in this order in τ such that t' performs an external read of x, and (p, t) is the last event in τ before (p', t') that writes to x (to mark the variable x, we may use $\mathsf{WR}(x)$), the *write-write* order WW (also called store-order), which relates any two events (p, t) and (p', t') that occur in this order in τ and write to the same variable x (to mark the variable x, we may use $\mathsf{WW}(x)$), the *read-write* relation RW (also called *conflict*), which relates any two events (p, t) and (p', t') that occur in this order in τ such that t reads a value that is overwritten by t'. The read-write relation RW is formally defined as $\mathsf{RW}(x) = \mathsf{WR}^{-1}(x); \mathsf{WW}(x)$ (we use ; to denote the standard composition of relations) and $\mathsf{RW} = \bigcup_{x \in \mathbb{V}} \mathsf{RW}(x)$.

We will present additional semantics of other consistency models later in the dissertation. In particular, the notions of trace under these semantics will be very different than the one under serializability. This is because the transition rules, i.e., events, are different from one semantics to another, and therefore, the definition of the dependency relations of a trace will differ from one semantics to another. Moreover, to facilitate the robustness comparison between serializability and other consistency models, the set of serializable traces will be enlarged to include traces that are equivalent, up to reordering of events that are not related by a dependency relation, to traces of serializable executions as defined above.

Chapter 3

Robustness Against Causal Consistency

3.1 Introduction

In this chapter, we investigate three models of causal consistency: causal memory (`CM`), causal convergence (`CCv`), and weak causal consistency (`wCC`). We study the robustness problem against one of these models relative to serializability. In §3.2, we outline the three consistency models and the robustness problem. In §3.3, we formally define the semantics for the three causal consistency models. We also define programs traces and executions under these semantics. In §3.4, we show that programs without write-write data race have the same behaviors under the three causal Consistency models. We also prove that if program admits a write-write data race under a variation of causal consistency, then it must admit a write-write data race under the other two variations. These results will allow us latter to derive the characterization of the robustness against `wCC` using the characterization of the robustness against `CM`. In §3.6, we define a class of robustness violations called *minimal* violations. In §3.7 and §3.8, we present a series of results that characterize the particular shapes of minimal violations for `CCv` and `CM`, respectively. Finally, in §3.10, we show a polynomial-time reduction of robustness checking to a reachability problem in a program running under sequential consistency.

```
                  t3 [x = 2
t1 [z = 1              r1 = z] //0
    x = 1]   ||   t4 [r2 = y  //1
t2 [y = 1]             r3 = x] //2
```

(a) CCv but not CM.

```
t1 [x = 1]              t3 [x = 2]
                   ||
t2 [r1 = x] //2         t4 [r2 = x] //1
```

(b) CM but not CCv.

```
                      t2 [x = 1]
t1 [x = 2]   ||       t3 [r1 = x] //2
                      t4 [r2 = x] //1
```

(c) wCC but not CM nor CCv.

Figure 3.1: Program computations showing the relationship between wCC, CCv and CM. Transactions are delimited using brackets and the transactions issued on the same site are aligned vertically. The values read in a transaction are given in comments.

3.2 Overview

In this section, we overview three variations of causal consistency introduced in the literature, weak causal consistency (wCC) [142, 52], causal memory (CM) [26, 142], and causal convergence (CCv) [59]. We illustrate the robustness problems against these models relative to serializability (SER).

The weakest variation of causal consistency, namely wCC, allows speculative executions and roll-backs of transactions which are not causally related (concurrent). For instance, the computation in Figure 3.1c is only feasible under wCC: the site on the right applies t2 after t1 before executing t3 and roll-backs t2 before executing t4. CCv and CM offer more guarantees. CCv enforces a total *arbitration* order between all transactions which defines the order in which delivered concurrent transactions are executed by *every* site. This guarantees that all sites reach the same state when all transactions are delivered. CM ensures that *all* values read by a site can be explained by an interleaving of transactions consistent with the causal order, enforcing thus PRAM consistency [121] on top of wCC.

Contrary to CCv, CM allows that two sites diverge on the ordering of concurrent transactions, but both models do not allow roll-backs of concurrent transactions. Thus, CCv and CM are incomparable in terms of computations they admit. The computation in Figure 3.1a is not admitted by CM because there is no interleaving of those transactions that explains the values read by the site on the right: reading 0 from z implies that the transactions on the left must be applied after t3 while reading 1 from y implies that both t1 and t2 are applied before t4 which contradicts reading 2 from x. However, this computation is possible under CCv because t1 can be delivered to the right after executing t3 but arbitrated before t3, which implies that the write to x in t1 will be lost. The CM computation in Figure 3.1b is not possible under CCv because there is no arbitration order that

```
    p1                     p2
t1 [r1 = x   //0   || t2 [r2 = x   //0
    x  = r1 + 1]           x = r2 + 1]
```

(a) Lost Update (LU).

```
    p1                 p2
t1 [x = 1          || t2 [y = 1
    r1 = y] //0           r2 = x] //0
```

(b) Store Buffering (SB).

```
             if (a == 1)
[a = 1         [x = 2
 z = 1    ||    r1 = z  //0
 x = 1          r2 = y  //1
 y = 1]         r3 = x] //2
```

(c) Without transactions, non-robust against CCv.

```
[a = 1           if (a == 1)
 x = 1       ||    [x = 2
 r1 = x] //2        r2 = x] //1
```

(d) Without transactions, non-robust only against CM.

```
if ( * )          if ( * )
  [x = 1]     ||    [x = 2]
else              else
  [r1 = x]          [r2 = x]
```

(e) Robust against both CM and CCv.

```
                [r2 = x
[x = 1]     ||   if (r2 == 1)
[r1 = y]           y = 1]
```

(f) Robust against both CM and CCv.

Figure 3.2: (Non-)robust programs. For non-robust programs, the read instructions are commented with the values they return in robustness violations. The condition of `if-else` is checked inside a transaction whose demarcation is omitted for readability ($*$ denotes non-deterministic choice).

could explain both reads from `x`.

Notice that each of the computations in Figures 3.1a, 3.1b, and 3.1c contains a write-write race. We show that the three causal consistency models coincide for programs containing no *write-write races* (i.e., concurrent transactions writing on a common variable), which explains why none of these computations is possible under all three models. We also show that if a program has a write-write race under one of these models, then it must have a write-write race under any of the other two models. This property is rather counter-intuitive since `wCC` is strictly weaker than both `CCv` and `CM`, and `CCv` and `CM` are incomparable (in terms of admitted behaviors).

We now discuss several examples of programs which are (non-) robust against both `CM` and `CCv` or only one of them. Robustness violations are presented in terms of "observable" behaviors, tuples of values that can be read in the different transactions and that are not possible under the serializability semantics (they correspond to traces with acyclic transactional happens-before). Figure 3.2a and Figure 3.2b show examples of programs that are *not* robust against both `CM` and `CCv`, which have also been discussed in the literature on weak memory models, e.g. [30]. The execution of Lost Update under both `CM` and `CCv` allows that the two reads of `x` in transactions $t1$ and $t2$

return 0 although this cannot happen under serializability. Also, executing Store Buffering under both `CM` and `CCv` allows that the reads of `x` and `y` return 0 although this would not be possible under serializability. These values are possible because the transaction in each of the processes may not be delivered to the other process.

Assuming for the moment that each instruction in Figure 3.2c and Figure 3.2d forms a different transaction, the values we give in comments show that the program in Figure 3.2c, resp., Figure 3.2d, is not robust against `CCv`, resp., `CM`. We associate a timestamp with each transaction, which will allow to fix an arbitration order between transactions. The values in Figure 3.2c are possible assuming that the timestamp of the transaction [$\texttt{x} = \texttt{1}$] is smaller than the timestamp of [$\texttt{x} = \texttt{2}$] (which means that if the former is delivered after the second process executes [$\texttt{x} = \texttt{2}$], then it will be discarded). Moreover, enlarging the transactions as shown in Figure 3.2c, the program becomes robust against `CCv`. The values in Figure 3.2d are possible under `CM` because different processes do not need to agree on the order in which to apply transactions, each process applying the transaction received from the other process last. However, under `CCv` this behavior is not possible, the program being actually robust against `CCv`. As in the previous case, enlarging the transactions as shown in the figure leads to a robust program against `CM`.

The approach we use for tackling the robustness verification problem is based on a precise characterization of the set of robustness violations. For both `CCv` and `CM`, we show that it is sufficient to search for a special type of robustness violations, that can be simulated by serial computations of an instrumentation of the original program. These computations maintain the information needed to recognize the pattern of a violation that would have occurred in the original program under a causally consistent semantics (executing the same set of transactions). A surprising consequence of these results is that a program is robust against `CM` iff it is robust against `wCC`, and robustness against `CM` implies robustness against `CCv`. This shows that the causal consistency variations we investigate can be incomparable in terms of the admitted behaviors, but comparable in terms of the robust applications they support.

We end the discussion with several examples of programs that are robust against both `CM` and `CCv`. These are simplified models of real applications reported in [110]. The program in Figure 3.2e can be understood as the parallel execution of two processes that either create a new user of some service, represented abstractly as a write on a variable `x` or check its credentials, represented as a read of `x` (the non-deterministic choice abstracts some code that checks whether the user exists).

Clearly this program is robust against both CM and CCv since each process does a single access to the shared variable. Although we considered simple transactions that access a single shared variable this would hold even for "bigger" transactions that access an arbitrary number of variables. The program in Figure 3.2f can be thought of as a process creating a new user of some service and reading some additional data in parallel to a process that updates that data only if the user exists. It is rather easy to see that it is also robust against both CM and CCv.

3.3 Causal Consistency

3.3.1 Program Semantics Under Causal Memory

Informally, the semantics of a program under causal memory is defined as follows. The shared variables are replicated across each process, each process maintaining its own local valuation of these variables. During the execution of a transaction in a process, the shared-variable writes are stored in a *transaction log* which is visible only to the process executing the transaction and which is broadcasted to all the processes at the end of the transaction[1]. To read a shared variable x, a process p first accesses its transaction log and takes the last written value on x, if any, and then its own valuation of the shared variables, if x was not written during the current transaction. Transaction logs are delivered to every process in an order consistent with the *causal delivery* relation between transactions, i.e., the transitive closure of the union of the *program order* (the order in which transactions are executed by a process), and the *delivered-before* relation (a transaction t_1 is delivered-before a transaction t_2 iff the log of t_1 has been delivered at the process executing t_2 before t_2 starts). By an abuse of terminology, we call this property *causal delivery*. Once a transaction log is delivered, it is immediately applied on the shared-variable valuation of the receiving process. Also, no transaction log can be delivered to a process p while p is executing another transaction, we call this property *transaction isolation*.

Formally, a program configuration is a triple $\mathsf{gs} = (\mathsf{ls}, \mathsf{msgs})$ where $\mathsf{ls} : \mathbb{P} \to \mathbb{S}$ associates a local state in $\mathbb{S}$ to each process in $\mathbb{P}$, and msgs is a set of messages in transit. A local state is a tuple $\langle \mathsf{pc}, \mathsf{store}, \mathsf{rval}, \mathsf{log} \rangle$ where $\mathsf{pc} \in \mathbb{L}\mathsf{ab}$ is the program counter, i.e., the label of the next instruction to be executed, $\mathsf{store} : \mathbb{V} \to \mathbb{D}$ is the local valuation of the shared variables, $\mathsf{rval} : \mathbb{R} \to \mathbb{D}$ is the

[1]For simplicity, we assume that every transaction commits. The effects of aborted transactions shouldn't be visible to any process.

valuation of the local registers, and $\mathsf{log} \in (\mathbb{V} \times \mathbb{D})^*$ is the transaction log, i.e., a list of variable-value pairs. For a local state s, we use $s.\mathsf{pc}$ to denote the program counter component of s, and similarly for all the other components of s. A message $m = \langle t, log\rangle$ is a transaction identifier t from a set $\mathbb{T}$ together with a transaction log $log \in (\mathbb{V} \times \mathbb{D})^*$. We let $\mathbb{M}$ denote the set of messages.

Then, the semantics of a program $\mathcal{P}$ under causal memory is defined using a LTS $[\mathcal{P}]_{\mathtt{CM}} = (\mathbb{C}, \mathbb{E}\mathsf{v}, \mathsf{gs}_0, \mathbb{C}_{\mathbb{F}}, \rightarrow)$ where we assume that any program configuration can be final, i.e., $\mathbb{C}_{\mathbb{F}} = \mathbb{C}$. As it will be explained later in this section, the executions of $\mathcal{P}$ under causal memory are a subset of those generated by $[\mathcal{P}]_{\mathtt{CM}}$. The set of events is defined by:

$$\mathbb{E}\mathsf{v} = \{\ \mathsf{begin}(p,t), \mathsf{ld}(p,t,x,v), \mathsf{isu}(p,t,x,v), \mathsf{del}(p,t),\ \mathsf{com}(p,t) : p \in \mathbb{P}, t \in \mathbb{T}, x \in \mathbb{V}, v \in \mathbb{D}\}$$

where begin and com label transitions corresponding to the start, resp., the end of a transaction, isu and ld label transitions corresponding to writing, resp., reading, a shared variable during some transaction, and del labels transitions corresponding to applying a transition log to the local state of the process issuing the transaction or to the state of another process that received the log. An event isu is called an *issue* while an event del is called a *store*.

The transition relation $\rightarrow$ is partially defined in Figure 3.3 (we will present additional constraints later in this section). The events labeling a transition are written on top of $\rightarrow$. A begin transition will just reset the transaction log while an com transition will add the transaction log together with the transaction identifier to the set msgs of messages in transit. An ld transition will read the value of a shared-variable looking first at the transaction log log and then, at the shared-variable valuation store, while an isu transition will add a new write to the transaction log. Finally, a del transition represents the delivery of a transaction log that was in transit which is applied immediately on the shared-variable valuation store.

We say that an execution ρ satisfies *transaction isolation* if no transaction log is delivered to a process p while p is executing a transaction, i.e., if an event $ev = \mathsf{del}(p,t)$ occurs in ρ before an event $ev' = \mathsf{com}(p,t')$ with $t' \neq t$, then ρ contains an event $ev'' = \mathsf{begin}(p,t')$ between ev and ev'. For an execution ρ satisfying transaction isolation, we assume w.l.o.g. that transactions executed by different processes do not interleave, i.e., if an event ev associated to a transaction t (an event of the process executing t or the delivery of the transaction log of t) occurs in ρ before $ev' = \mathsf{com}(p',t')$, then ρ contains an event $ev'' = \mathsf{begin}(p',t')$ between ev and ev'. Formally, we say that an execution ρ satisfies *causal delivery* if the following hold:

- for any event $\mathsf{begin}(p,t)$, and for any process p', ρ contains at most one event $\mathsf{del}(p',t)$,

$$\frac{\mathsf{begin} \in \mathsf{inst}(\mathsf{ls}(p).\mathsf{pc}) \quad s = \mathsf{ls}(p)[\mathsf{log} \mapsto \epsilon, \mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})]}{(\mathsf{ls}, \mathsf{msgs}) \xrightarrow{\mathsf{begin}(p,\, t)} (\mathsf{ls}[p \mapsto s], \mathsf{msgs})}$$

$$\frac{r := x \in \mathsf{inst}(\mathsf{ls}(p).\mathsf{pc}) \quad \mathit{eval}(\mathsf{ls}(p), x) = v \quad \mathit{rval} = \mathsf{ls}(p).\mathsf{rval}[r \mapsto v] \quad s = \mathsf{ls}(p)[\mathsf{rval} \mapsto \mathit{rval}, \mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})]}{(\mathsf{ls}, \mathsf{msgs}) \xrightarrow{\mathsf{ld}(p,\, t,\, x,\, v)} (\mathsf{ls}[p \mapsto s], \mathsf{msgs})}$$

$$\frac{x := v \in \mathsf{inst}(\mathsf{ls}(p).\mathsf{pc}) \quad \mathit{log} = (\mathsf{ls}(p).\mathsf{log}) \cdot (x, v) \quad s = \mathsf{ls}(p)[\mathsf{log} \mapsto \mathit{log}, \mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})]}{(\mathsf{ls}, \mathsf{msgs}) \xrightarrow{\mathsf{isu}(p,\, t,\, x,\, v)} (\mathsf{ls}[p \mapsto s], \mathsf{msgs})}$$

$$\frac{\mathsf{end} \in \mathsf{inst}(\mathsf{ls}(p).\mathsf{pc}) \quad s = \mathsf{ls}(p)[\mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})]}{(\mathsf{ls}, \mathsf{msgs}) \xrightarrow{\mathsf{com}(p,\, t)} (\mathsf{ls}[p \mapsto s], \mathsf{msgs} \cup \{(t, \mathsf{ls}(p).\mathsf{log})\})}$$

$$\frac{\langle t, \mathit{log}\rangle \in \mathsf{msgs} \quad \mathit{store} = \mathsf{ls}(p).\mathsf{store}[x \mapsto \mathit{last}(\mathit{log}, x) : x \in \mathbb{V}, \mathit{last}(\mathit{log}, x) \neq \bot] \quad s = \mathsf{ls}(p)[\mathsf{store} \mapsto \mathit{store}]}{(\mathsf{ls}, \mathsf{msgs}) \xrightarrow{\mathsf{del}(p,\, t)} (\mathsf{ls}[p \mapsto s], \mathsf{msgs})}$$

Figure 3.3: The set of transition rules defining the causal memory semantics. We use $\cdot$ to denote sequence concatenation. The function $\mathit{eval}(\mathsf{ls}(p), x)$ returns the value of x in the local state $\mathsf{ls}(p)$: (1) if $\mathsf{ls}(p).\mathsf{log}$ contains a pair (x, v), for some v, then $\mathit{eval}(\mathsf{ls}(p), x)$ returns the value of the last such pair in $\mathsf{ls}(p).\mathsf{log}$, and (2) $\mathit{eval}(\mathsf{ls}(p), x)$ returns $\mathsf{ls}(p).\mathsf{store}(x)$, otherwise. Also, $\mathit{last}(\mathit{log}, x)$ returns the value v in the last pair (x, v) in log, and $\bot$, if such a pair does not exist.

- for any two events $\mathsf{begin}(p, t)$ and $\mathsf{begin}(p, t')$, if $\mathsf{begin}(p, t)$ occurs in ρ before $\mathsf{begin}(p, t')$, then the event $\mathsf{del}(p, t)$ occurs before $\mathsf{begin}(p, t')$ in ρ. This ensures that when p issues t it must store the writes of t in its local state before issuing another transaction t';
- for any events $ev_1 \in \{\mathsf{del}(p, t_1), \mathsf{com}(p, t_1)\}$, $ev_2 = \mathsf{begin}(p, t_2)$, and $ev_2' = \mathsf{del}(p', t_2)$ with $p \neq p'$, if ev_1 occurs in ρ before ev_2, then there exists $ev_1' = \mathsf{del}(p', t_1)$ such that ev_1' occurs before ev_2' in ρ.

An execution ρ satisfies *causal memory* if it satisfies transaction isolation and causal delivery. The set of executions of $\mathcal{P}$ under causal memory, denoted by $\mathbb{E}\mathrm{x}_{\mathtt{CM}}(\mathcal{P})$, is the set of executions of $[\mathcal{P}]_{\mathtt{CM}}$ satisfying causal memory.

Figure 3.4a shows an execution under `CM`. This satisfies transaction isolation since no transaction is delivered while another transaction is executing.

3.3.2 Program Semantics Under Causal Convergence

Compared to causal memory, causal convergence ensures eventual consistency of process-local copies of the shared variables. Each transaction log is associated with a timestamp and a process applies a

$$\begin{array}{l}\mathsf{begin}(p1,t1)\\ \mathsf{isu}(p1,t1,x,1)\cdot\mathsf{del}(p1,t1)\\ \mathsf{com}(p1,t1)\end{array}\cdot\begin{array}{l}\mathsf{begin}(p2,t3)\\ \mathsf{isu}(p2,t3,x,2)\cdot\mathsf{del}(p2,t3)\cdot\mathsf{del}(p1,t3)\\ \mathsf{com}(p2,t3)\end{array}\cdot\ \mathsf{del}(p2,t1)\ \cdot\begin{array}{l}\mathsf{begin}(p2,t4)\\ \mathsf{ld}(p2,t4,x,1)\\ \mathsf{com}(p2,t4)\end{array}\cdot\begin{array}{l}\mathsf{begin}(p1,t2)\\ \mathsf{ld}(p1,t2,x,2)\\ \mathsf{com}(p1,t2)\end{array}$$

(a) `CM` execution of the program in Figure 3.1b.

$$\begin{array}{l}\mathsf{begin}(p1,t1)\\ \mathsf{isu}(p1,t1,z,1)\\ \mathsf{isu}(p1,t1,x,1)\\ \mathsf{com}(p1,t1)\end{array}\cdot\mathsf{del}(p1,t1)\cdot\begin{array}{l}\mathsf{begin}(p2,t3)\\ \mathsf{isu}(p2,t3,x,2)\\ \mathsf{ld}(p2,t3,z,0)\\ \mathsf{com}(p2,t3)\end{array}\cdot\mathsf{del}(p2,t3)\cdot\mathsf{del}(p1,t3)\ \cdot\ \mathsf{del}(p2,t1)\ \cdot\begin{array}{l}\mathsf{begin}(p1,t2)\\ \mathsf{isu}(p1,t2,y,1)\\ \mathsf{com}(p1,t2)\end{array}\cdot\mathsf{del}(p1,t2)\cdot\mathsf{del}(p2,t2)\cdot\begin{array}{l}\mathsf{begin}(p2,t4)\\ \mathsf{ld}(p2,t4,y,1)\\ \mathsf{ld}(p2,t4,x,2)\\ \mathsf{com}(p2,t4)\end{array}$$

(b) `CCv` execution of the program in Figure 3.1a.

$$\begin{array}{l}\mathsf{begin}(p1,t1)\\ \mathsf{isu}(p1,t1,x,2)\cdot\mathsf{del}(p1,t1)\\ \mathsf{com}(p1,t1)\end{array}\cdot\begin{array}{l}\mathsf{begin}(p2,t2)\\ \mathsf{isu}(p2,t2,x,1)\cdot\mathsf{del}(p2,t2)\\ \mathsf{com}(p2,t2)\end{array}\cdot\ \mathsf{del}(p2,t1)\ \cdot\begin{array}{l}\mathsf{begin}(p2,t3)\\ \mathsf{ld}(p2,t3,x,2)\\ \mathsf{com}(p2,t3)\end{array}\cdot\begin{array}{l}\mathsf{begin}(p2,t4)\\ \mathsf{ld}(p2,t4,x,1)\\ \mathsf{com}(p2,t4)\end{array}\cdot\ \mathsf{del}(p1,t2)$$

(c) `wCC` execution of the program in Figure 3.1c.

Figure 3.4: For readability, the sub-sequences of events delimited by begin and com are aligned vertically, the execution-flow advancing from left to right and top to bottom.

write on some variable x from a transaction log only if it has a timestamp larger than the timestamps of all the transaction logs it has already applied and that wrote the same variable x. For simplicity, we assume that the transaction identifiers play the role of timestamps, which are totally ordered according to some relation $<$. `CCv` satisfies both *causal delivery* and *transaction isolation* as well. Assuming that transactions are constituted of either a read alone or a write alone, `CCv` is equivalent to Strong Release-Acquire (SRA), a strengthening of the standard Release-Acquire semantics of the C11 memory model [112][2].

Formally, we define a variation of the LTS $[\mathcal{P}]_{\texttt{CM}}$, denoted by $[\mathcal{P}]_{\texttt{CCv}}$, where essentially, the transition identifiers play the role of timestamps and are ordered by a total order $<$, each process-local state contains an additional component tstamp storing the largest timestamp the process has seen for each variable, and a write on a variable x from a transaction log is applied on the local valuation store only if it has a timestamp larger than $\mathsf{tstamp}(x)$. Also, a $\mathsf{begin}(p,t)$ transition will choose a transaction identifier t greater than those in the image of the tstamp component of p's local state. The transition rules of $[\mathcal{P}]_{\texttt{CCv}}$ that change w.r.t. those of $[\mathcal{P}]_{\texttt{CM}}$ are given in Figure 3.5.

The set of executions of $\mathcal{P}$ under causal convergence, denoted by $\mathbb{E}\mathsf{x}_{\texttt{CCv}}(\mathcal{P})$, is the set of executions of $[\mathcal{P}]_{\texttt{CCv}}$ satisfying transaction isolation, causal delivery, and the fact that every process p generates

[2]This equivalence excludes the atomic read-modify-write (also know as compare-and-swap) operation which is not provided by `CCv`.

$$\frac{\Phi_1 \quad \mathsf{img}(\mathsf{ls}(p).\mathsf{tstamp}) < t}{(\mathsf{ls}, \mathsf{lk}, \mathsf{msgs}) \xrightarrow{\mathsf{begin}(p,\, t)} (\mathsf{ls}[p \mapsto s], \mathsf{lk}, \mathsf{msgs})}$$

$$\frac{\begin{array}{c}\langle t, \mathit{log}\rangle \in \mathsf{msgs} \quad \mathit{store} = \mathsf{ls}(p).\mathsf{store}[x \mapsto \mathit{last}(\mathit{log}, x) : x \in \mathbb{V}, \mathit{last}(\mathit{log}, x) \neq \bot, \mathsf{tstamp}(x) < t] \\ \mathit{tstamp} = \mathsf{ls}(p).\mathsf{tstamp}[x \mapsto t : x \in \mathbb{V}, \mathit{last}(\mathit{log}, x) \neq \bot, \mathsf{tstamp}(x) < t] \quad s = \mathsf{ls}(p)[\mathsf{store} \mapsto \mathit{store}, \mathsf{tstamp} \mapsto \mathit{tstamp}]\end{array}}{(\mathsf{ls}, \mathsf{lk}, \mathsf{msgs}) \xrightarrow{\mathsf{del}(p,\, t)} (\mathsf{ls}[p \mapsto s], \mathsf{lk}, \mathsf{msgs})}$$

Figure 3.5: Transition rules for defining causal convergence. Φ_1 is the hypothesis of the $\mathsf{begin}(p, t)$ transition rule in Figure 3.3, and img denotes the image of a function.

monotonically increasing transaction identifiers.

The execution in Figure 3.4a is not possible under causal convergence since $t4$ and $t2$ read 2 and 1 from x, respectively. This is possible only if $t1$ and $t3$ write x at $p2$ and $p1$, respectively, which contradicts the definition of del transition given in Figure 3.5 where we cannot have both $t1 < t3$ and $t3 < t1$ at the same time. Figure 3.4b shows an execution under CCv (we assume $t_1 < t_2 < t_3 < t_4$). Notice that $\mathsf{del}(p2, t1)$ did not result in an update of x because the timestamp t_1 is smaller than the timestamp of the last transaction that wrote x at p_2, namely t_3, a behavior that is not possible under CM. The two processes converge and store the same shared variable copy at the end of the execution.

3.3.3 Program Semantics Under Weak Causal Consistency

Compared to the previous semantics, wCC allows that reads of the same process observe concurrent writes as executing in different orders. Each process maintains a *set* of values for each shared variable, and a read returns any one of these values non-deterministically. Transaction logs are associated with *vector clocks* [116] which represent the causal delivery relation, i.e., a transaction t_1 is before t_2 in causal-delivery iff the vector clock of t_1 is smaller than the vector clock of t_2. We assume that transactions identifiers play the role of vector clocks, which are partially ordered according to some relation $<$. In applying the log of a transaction t on the local state of the receiving process p, the final *set* of values for each shared variable in p will be constituted of the value in the log of t and the values that were written by *concurrent* transactions (not related by causal delivery to t). wCC satisfies both *causal delivery* and *transaction isolation*.

Formally, in wCC semantics, the local valuation of the shared variables $\mathsf{store} : \mathbb{V} \to (\mathbb{D} \times \mathbb{T})^*$ is a map that accepts a shared variable and returns a set of pairs. The pairs are constituted of values

$$\frac{\mathtt{begin} \in \mathsf{inst}(\mathsf{ls}(p).\mathsf{pc}) \quad \mathsf{img}(\mathsf{ls}(p).\mathsf{tstamp}) < t \quad s = \mathsf{ls}(p)[\mathsf{log} \mapsto \epsilon, \mathsf{snapshot} \mapsto \mathsf{buildSnapshot}(\mathsf{store}), \mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})]}{(\mathsf{ls}, \mathsf{msgs}) \xrightarrow{\mathsf{begin}(p,\, t)} (\mathsf{ls}[p \mapsto s], \mathsf{msgs})}$$

$$\frac{r := x \in \mathsf{inst}(\mathsf{ls}(p).\mathsf{pc}) \quad cceval(\mathsf{ls}(p), x) = (v, t') \quad rval = \mathsf{ls}(p).\mathsf{rval}[r \mapsto v] \quad s = \mathsf{ls}(p)[\mathsf{rval} \mapsto rval, \mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})]}{(\mathsf{ls}, \mathsf{msgs}) \xrightarrow{\mathsf{ld}(p,\, t,\, x,\, v)} (\mathsf{ls}[p \mapsto s], \mathsf{msgs})}$$

$$\frac{\mathtt{end} \in \mathsf{inst}(\mathsf{ls}(p).\mathsf{pc}) \quad s = \mathsf{ls}(p)[\mathsf{snapshot} \mapsto \epsilon, \mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})]}{(\mathsf{ls}, \mathsf{msgs}) \xrightarrow{\mathsf{com}(p,\, t)} (\mathsf{ls}[p \mapsto s], \mathsf{msgs} \cup \{(t, \mathsf{ls}(p).\mathsf{log})\})}$$

$$\frac{\langle t, log\rangle \in \mathsf{msgs} \quad store = \mathsf{ls}(p).\mathsf{store}[x \mapsto update(\mathsf{ls}(p), x, t, last(log, x)) : x \in \mathbb{V}] \quad s = \mathsf{ls}(p)[\mathsf{store} \mapsto store, \mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})]}{(\mathsf{ls}, \mathsf{msgs}) \xrightarrow{\mathsf{del}(p,\, t)} (\mathsf{ls}[p \mapsto s], \mathsf{msgs})}$$

Figure 3.6: Transition rules for defining weak causal consistency semantics: buildSnapshot(store) returns a consistent snapshot of store. $cceval(\mathsf{ls}(p), x)$ returns the pair $(last(log, x), t)$ if $last(log, x) \neq \bot$, and returns the pair (v, t') in $\mathsf{ls}(p).\mathsf{snapshot}(x)$, otherwise. $update(\mathsf{ls}(p), x, t, last(log, x))$ returns the result of appending the pair $(last(log, x), t)$ to the set $\mathsf{ls}(p).\mathsf{store}(x)$ after removing all pairs that contain values overwritten by t.

that were written concurrently and identifiers of the transactions that wrote those values. When applying a transaction log on the local valuation store, we keep the values that were written by transactions that are concurrent with the current transaction. Additionally, in the wCC semantics, the local state of a process has an additional component snapshot : $\mathbb{V} \to (\mathbb{D} \times \mathbb{T})$ that maps each shared variable to a single pair. snapshot is obtained by taking a "consistent" snapshot from store when a new transaction starts. Such a snapshot corresponds to a linearization of the transactions that were delivered to the process, which is consistent with the vector clock order. The snapshot associates to each variable the last value written in this linearization. When a process does a read from a shared variable x, it looks first at the transaction log log and then, at the variable valuation snapshot. In Figure 3.6, we provide the transition rules of $[\mathcal{P}]_{\mathtt{wCC}}$ that change w.r.t. those of $[\mathcal{P}]_{\mathtt{CCv}}$ and $[\mathcal{P}]_{\mathtt{CM}}$.

The set of executions of $\mathcal{P}$ under weak causal consistency model, denoted by $\mathbb{E}\mathsf{x}_{\mathtt{wCC}}(\mathcal{P})$, is the set of executions of $[\mathcal{P}]_{\mathtt{wCC}}$ satisfying transaction isolation and causal delivery. We denote by $\mathbb{T}\mathsf{r}(\mathcal{P})_{\mathtt{wCC}}$ the set of traces of executions of a program $\mathcal{P}$ under weak causal consistency.

Figure 3.4c shows an execution under wCC, which is not possible under CCv and CM because $t3$ and $t4$ read 2 and 1, respectively. Since the transactions t_1 and t_2 are concurrent, p_2 stores both values 2 and 1 written by these transactions. A read of x can return any of these two values.

3.3.4 Execution Summary

Let ρ be an execution under $\mathsf{X} \in \{\texttt{CCv}, \texttt{CM}, \texttt{wCC}\}$, a sequence τ of events $\mathsf{isu}(p, t)$ and $\mathsf{del}(p, t)$ with $p \in \mathbb{P}$ and $t \in \mathbb{T}$ is called a *summary of* ρ if it is obtained from ρ by substituting every sub-sequence of transitions in ρ delimited by a begin and an com transition, with a single "macro-event" $\mathsf{isu}(p, t)$. For example, $\mathsf{isu}(p1, t1) \cdot \mathsf{isu}(p2, t3) \cdot \mathsf{del}(p1, t3) \cdot \mathsf{del}(p2, t1) \cdot \mathsf{isu}(p2, t4) \cdot \mathsf{isu}(p1, t2)$ is a summary of the execution in Figure 3.4a.

We say that a transaction t in ρ performs an *external read* of a variable x if ρ contains an event $\mathsf{ld}(p, t, x, v)$ which is not preceded by a write on x of t, i.e., an event $\mathsf{isu}(p, t, x, v)$. Under $\texttt{CM}$ and $\texttt{wCC}$, a transaction t *writes* a variable x if ρ contains an event $\mathsf{isu}(p, t, x, v)$, for some v. In Figure 3.4a, both $t2$ and $t4$ perform external reads and $t2$ writes to y. A transaction t executed by a process p *writes* x *at process* p' if t writes x and ρ contains an event $\mathsf{del}(p', t)$ (e.g., in Figure 3.4a, $t1$ writes x at $p2$). Under $\texttt{CCv}$, we say that a transaction t executed by a process p *writes* x *at process* p' if t writes x and ρ contains an event $\mathsf{del}(p', t)$ which is not preceded by an event $\mathsf{del}(p', t')$ with $t < t'$ and t' writing x (if it would be preceded by such an event then the write to x of t will be discarded). For example, in Figure 3.4b, $t1$ does *not* write x at $p2$.

3.3.5 Trace

We define an abstract representation of executions that satisfy transaction isolation[3], called *trace*. More precisely, the *trace* of an execution ρ is a tuple $\mathsf{tr}(\rho) = (\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{STO})$ where τ is the summary of ρ, PO is the *program order*, which relates any two issue events $\mathsf{isu}(p, t)$ and $\mathsf{isu}(p, t')$ that occur in this order in τ, WR is the *write-read* relation, which relates events of two transactions t and t' such that t writes a value that t' reads, WW is the *write-write* order, which relates events of two transactions that write to the same variable, RW is the *read-write* relation, which relates events of two transactions t and t' such that t reads a value overwritten by t', and STO is the *same-transaction* relation, which relates events of the same transaction.

Definition 3.1 (Traces). *Formally, the* trace *of an execution ρ satisfying transaction isolation is* $\mathsf{tr}(\rho) = (\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{STO})$ *where τ is a summary of ρ, and*

[3]We refer collectively to executions in $[\mathcal{P}]_{\mathsf{X}}$ with $\mathsf{X} \in \{\texttt{CCv}, \texttt{CM}, \texttt{wCC}\}$.

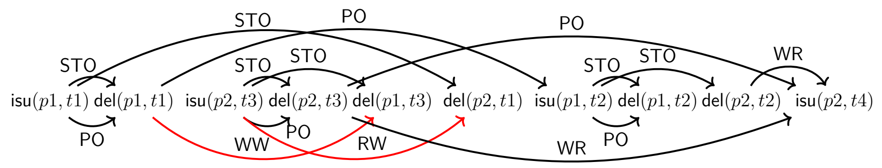
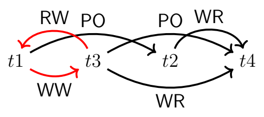

STO
PO
PO
STO
STO
STO
STO
STO
WR
isu(p1, t1) del(p1, t1) isu(p2, t3) del(p2, t3) del(p1, t3) del(p2, t1) isu(p1, t2) del(p1, t2) del(p2, t2) isu(p2, t4)
PO
WW
PO
RW
WR
PO


Figure 3.7: The trace of the execution in Figure 3.4b and its transactional happens-before.

PO*: relates the issue and store events* $\mathsf{isu}(p,t)$ *and* $\mathsf{del}(p,t)$ *of* t *and subsequently, the event* $\mathsf{del}(p,t)$ *with any issue event* $\mathsf{isu}(p,t')$ *that occurs after it in* τ.

WR*: relates any store and issue events* $ev_1 = \mathsf{del}(p,t)$ *and* $ev_2 = \mathsf{isu}(p,t')$ *that occur in this order in* τ *such that* t' *performs an external read of* x*, and* ev_1 *is the last event in* τ *before* ev_2 *such that* t *writes* x *at* p*. To make the shared variable* x *explicit, we may use* $\mathsf{WR}(x)$ *to name the relation between* ev_1 *and* ev_2.

WW*: relates events of two transactions that write to the same variable. More precisely,* WW *relates any two store events* $ev_1 = \mathsf{del}(p,t_1)$ *and* $ev_2 = \mathsf{del}(p,t_2)$ *that occur in this order in* τ *provided that* t_1 *and* t_2 *both write the same variable* x*, and if* ρ *is an execution under causal convergence, then* t_1 *and* t_2 *writes* x *at* p*, and* $t_1 < t_2$*. To make the shared variable* x *explicit, we may use* $\mathsf{WW}(x)$ *to name the relation between* ev_1 *and* ev_2.

RW*: relates events of two distinct transactions* t *and* t' *such that* t *reads a value that is overwritten by* t'*. Formally,* $\mathsf{RW}(x) = \mathsf{WR}^{-1}(x);\mathsf{WW}(x)$ *and* $\mathsf{RW} = \bigcup_{x\in\mathbb{V}} \mathsf{RW}(x)$*. If a transaction* t *reads the initial value of* x *then* $\mathsf{RW}(x)$ *relates* $\mathsf{isu}(p,t)$ *with every event* $\mathsf{del}(p',t')$ *with* $p' \in \mathbb{P}$ *of any other transaction* t' *that writes to* x *at* p'.

STO*: relates issue events with store events of the same transaction. More precisely,* STO *relates every event* $\mathsf{isu}(p,t)$ *with every event* $\mathsf{del}(p',t)$ *with* $p' \in \mathbb{P}$.

The following result states an important property of the store order relation WW that is enforced by the `CCv` semantics. It holds because the writes in different transactions are applied by different processes in the same order given by their timestamps, when visible (delivered) to those processes.

Lemma 3.1. *Let* $\tau \in \mathbb{T}r_{\mathtt{CCv}}(\mathcal{P})$ *be a trace. If* $(\mathsf{del}(p_0,t_0), \mathsf{del}(p_0,t_1)) \in \mathsf{WW}(x)$*, then for every process* p*,* $(\mathsf{del}(p,t_1), \mathsf{del}(p,t_0)) \notin \mathsf{WW}(x)$.

We define the *happens-before* relation HB as the transitive closure of the union of all the relations in the trace, i.e., $\mathsf{HB} = (\mathsf{PO} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW} \cup \mathsf{STO})^+$. Since we reason about only one trace at a time, we may say that a trace is simply a summary τ, keeping the relations implicit. The trace of the `CCv` execution in Figure 3.4b is shown on the left of Figure 3.7. $\mathbb{T}\mathsf{r}(\mathcal{P})_{\mathsf{X}}$ denotes the set of traces of executions of a program $\mathcal{P}$ under $\mathsf{X} \in \{\texttt{CCv}, \texttt{CM}, \texttt{wCC}\}$.

For readability, we write $ev_1 \rightarrow_{\mathsf{HB}} ev_2$ instead of $(ev_1, ev_2) \in \mathsf{HB}$ and ev_1 and ev_2 can be either $\mathsf{isu}(p,t)$ or $\mathsf{del}(p,t)$. We use the notation $ev_1 \rightarrow_{\mathsf{HB}^1} ev_2$ (resp., $(ev_1, ev_2) \in \mathsf{HB}^1$) to denote $(ev_1, ev_2) \in (\mathsf{PO} \cup \mathsf{WW} \cup \mathsf{WR} \cup \mathsf{STO} \cup \mathsf{RW})$.

The *causal order* CO of a trace $tr = (\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{STO})$ is the transitive closure of the union of the program order, write-read relation, and the same-transaction relation, i.e., $\mathsf{CO} = (\mathsf{PO} \cup \mathsf{WR} \cup \mathsf{STO})^+$. For readability, we write $ev_1 \rightarrow_{\mathsf{CO}} ev_2$ instead of $(ev_1, ev_2) \in \mathsf{CO}$.

Let t_1 and t_2 be two transactions issued in a trace tr that originate from two different processes p_1 and p_2, respectively. If $(\mathsf{isu}(p_1,t_1), \mathsf{isu}(p_2,t_2)) \notin \mathsf{CO}$ and $(\mathsf{isu}(p_2,t_2), \mathsf{isu}(p_1,t_1)) \notin \mathsf{CO}$, then t_1 and t_2 are called *concurrent* transactions.

The happens-before relation between events is extended to transactions as follows: a transaction t_1 *happens-before* another transaction $t_2 \neq t_1$ if the trace tr contains an event of transaction t_1 which happens-before an event of t_2. The happens-before relation between transactions is denoted by HB_t and called *ransactional happens-before* (an example is given on the right of Figure 3.7). For a trace of serializable execution, the transactional happens-before and the happens-before relation coincide.

Remark 3.1. *The operational models of causal consistency we described are equivalent to the axiomatic models defined in [52]. These axiomatic models are defined as a set of constraints on abstractions of executions, called* histories*, that consist of a set of read and write operations along with a program order, denoted by* PO'*, and a read-from relation, denoted by* WR'*:* PO' *relates operations in the same process and* WR' *associates every read operation to the write operation which wrote the read value. For instance, the axiomatic model of* `wCC` *requires that the union of* PO' *and* WR' *(denoted* CO'*) is acyclic*[4]*, and its composition with a variation of the conflict relation, denoted by* RW'*, (*$(a,b) \in \mathsf{RW}'$[5] *iff* $\exists\ c.\ (c,b) \in \mathsf{CO}' \wedge (c,a) \in \mathsf{WR}'$*) is irreflexive*[6]*. These models can be extended easily to histories that contain transactions instead of operations by adapting the above relations. Note that every program trace (cf. Definition 3.1) can be "projected" to a history where issue and store events from the same transaction in the trace are mapped to a single transaction in the history. Also, the read-from and the program order between trace events are mapped to the* WR'

[4]This constraint corresponds to the absence of the CyclicCO bad pattern in [52].

[5]b is overwriting the value a is reading.

[6]This constraint corresponds to the absence of the WriteCORead bad pattern in [52].

and PO' *of the history.*

To show equivalence between these models, it is sufficient to show that (1) every history corresponding to a trace in the operational model satisfies the constraints of the axiomatic model, and (2) every history that is valid under the axiomatic model is the "projection" of a trace of the operational model. For instance, for wCC*, it is easy to see that the relation* $\mathsf{CO}' = \mathsf{PO}' \cup \mathsf{WR}'$ *in a history that is the projection of a trace* $\tau \in \mathbb{T}\mathsf{r}_{\mathtt{wCC}}(\mathcal{P})$ *is acyclic because the causal order* CO *in* τ *is. Also, the proof that* $\mathsf{CO}'; \mathsf{RW}'$ *is irreflexive can be derived easily by contradiction (for instance, if* $(a, b) \in \mathsf{RW}'$ *and* $(b, a) \in \mathsf{CO}'$*, then there exists* c *such that* $(c, b) \in \mathsf{CO}'$ *which means that by causal delivery, a can never read the value written by c).*

3.4 Write-Write Race Freedom

We say that an execution ρ has a *write-write race* on a shared variable x if there exist two concurrent transactions t_1 and t_2 that were issued in ρ and each transaction contains a write to the variable x. We call ρ write-write race free if there is no variable x such that ρ has a write-write race on x. Also, we say a program $\mathcal{P}$ is *write-write race free* under a consistency model $\mathsf{X} \in \{\mathtt{CCv}, \mathtt{CM}, \mathtt{wCC}\}$ iff for every $\rho \in \mathbb{E}\mathsf{x}_{\mathsf{X}}(\mathcal{P})$, ρ is write-write race free.

We show that if a given program has a write-write race under one of the three causal consistency models then it must have a write-write race under the remaining two. The intuition behind this is that the three models coincide for programs without write-write races. Indeed, without concurrent transactions that write to the same variable, every process local valuation of a shared variable will be a singleton set under wCC and no process will ever discard a write when applying an incoming transaction log under CCv.

Theorem 3.1. *Given a program* $\mathcal{P}$ *and two consistency models* $\mathsf{X}, \mathsf{Y} \in \{\mathtt{CCv}, \mathtt{CM}, \mathtt{wCC}\}$*,* $\mathcal{P}$ *has a write-write race under* X *iff* $\mathcal{P}$ *has a write-write race under* Y*.*

Proof. Since wCC is weaker than both CCv and CM, it is sufficient to prove the following two cases: (1) if $\mathcal{P}$ has a write-write race under wCC, then $\mathcal{P}$ has a write-write race under CCv and (2) if $\mathcal{P}$ has a write-write race under wCC, then $\mathcal{P}$ has a write-write race under CM.

We prove the first case by induction on the number of transactions in $\mathcal{P}$. The second case can be proved in a similar way.

Base case: $\mathcal{P}$ is constituted of two transactions t_1 and t_2. Assume that $\mathcal{P}$ has a write-write race under wCC then the transactions t_1 and t_2 must originate from different processes. Thus, in any trace τ of $\mathcal{P}$ under CCv where the transactions t_1 and t_2 are executed concurrently we will have a write-write race between these two transactions. Thus, $\mathcal{P}$ has a write-write race under CCv.

Induction step: If $n > 2$ is the number of transactions in $\mathcal{P}$, we assume that for any program $\mathcal{P}'$ with $n' < n$ transactions, if $\mathcal{P}'$ has a write-write race under wCC, then $\mathcal{P}'$ has a write-write race under CCv. Assume that $\mathcal{P}$ has a write-write race under wCC. Let τ be a trace of $\mathcal{P}$ under wCC where we have a write-write race between two transactions t_1 and t_2 that were issued by processes p_1 and p_2, respectively. Executing t_1 and t_2 concurrently while writing to a common variable is not possible under CCv only if the writes were enabled by some events that occurred before t_1 and t_2 under wCC and are not possible under CCv. However, based on the semantic models of both wCC and CCv, if all the transactions that write to common variables are causally related then such events cannot occur under wCC but not CCv. Thus, we must have two other transactions t'_1 and t'_2 of $\mathcal{P}$ that were executed concurrently in τ under wCC and occurred before t_1 (or t_2 or both) which write to a common variable. Without loss of generality, let $\mathcal{P}_1$ be the program resulting from removing the transaction t_1 from $\mathcal{P}$. We know that $\mathcal{P}_1$ admits a trace τ_1 under wCC where the transactions t'_1 and t'_2 are involved in a data race. Also, the size of $\mathcal{P}_1$ is $n - 1 < n$. Thus, from the induction hypothesis we get that $\mathcal{P}_1$ has a write-write race under CCv. Because adding a new transaction to $\mathcal{P}_1$ will not eliminate existing data races, $\mathcal{P}$ has a write-write race under CCv as well.

□

The following result shows that indeed, the three causal consistency models coincide for programs which are write-write race free under any one of these three models.

Theorem 3.2. *Let $\mathcal{P}$ be a program. Then, $\mathbb{E}\mathsf{x}_{\mathtt{wCC}}(\mathcal{P}) = \mathbb{E}\mathsf{x}_{\mathtt{CCv}}(\mathcal{P}) = \mathbb{E}\mathsf{x}_{\mathtt{CM}}(\mathcal{P})$ iff $\mathcal{P}$ has no write-write race under neither* wCC, CM, *and* CCv.

Proof. Left-to-right direction: By Theorem 3.1, it is sufficient to prove that $\mathcal{P}$ has no write-write race under CM. Suppose by contradiction that $\mathcal{P}$ has a write-write race under CM. Then, there must exist a trace $\tau \in \mathbb{T}\mathsf{r}_{\mathtt{wCC}}(\mathcal{P})$ such that we have two concurrent transactions t_1 and t_2 that are issued in τ and write to a variable x. Assume w.l.o.g that the issue event of t_1 occurs before the issue event of t_2 in τ. Since t_1 and t_2 are concurrent in τ, the issue event of t_1 and the store events of t_2 are commutative, and the issue event of t_2 and the store events of t_1 are commutative. Then, $\tau' = \alpha \cdot \mathsf{isu}(p_1, t_1) \cdot \mathsf{del}(p_1, t_1) \cdot \beta \cdot \mathsf{isu}(p_2, t_2) \cdot \mathsf{del}(p_2, t_2) \cdot \mathsf{del}(p_1, t_2) \cdot \mathsf{del}(p_2, t_1)$ where α and

β are sequences of events in τ that t_1 and t_2 causally depend on (since we are not interested in other events)[7], is a trace of $\mathcal{P}$ under $\texttt{CM}$. In τ', both store events $\mathsf{del}(p_2, t_1)$ and $\mathsf{del}(p_1, t_2)$ do not discard any writes (guaranteed under $\texttt{CM}$). Therefore, $(\mathsf{del}(p_1, t_1), \mathsf{del}(p_1, t_2)) \in \mathsf{WW}(x)$ and $(\mathsf{del}(p_2, t_2), \mathsf{del}(p_2, t_1)) \in \mathsf{WW}(x)$ since both t_1 and t_2 write to x. However, it is impossible to obtain τ' under $\texttt{CCv}$ as we cannot have $(\mathsf{del}(p_2, t_2), \mathsf{del}(p_2, t_1)) \in \mathsf{WW}(x)$ if $(\mathsf{del}(p_1, t_1), \mathsf{del}(p_1, t_2)) \in \mathsf{WW}(x)$ which leads to a contradiction ($\mathcal{P}$ has different sets of traces under $\texttt{CM}$ and $\texttt{CCv}$).

Right-to-left direction: It is sufficient to prove the following two cases: if τ has no write-write race under $\texttt{wCC}$ then $\tau \in \mathbb{T}\mathsf{r}_{\texttt{wCC}}$ implies $\tau \in \mathbb{T}\mathsf{r}_{\texttt{CM}}$ and $\tau \in \mathbb{T}\mathsf{r}_{\texttt{CCv}}$ ($\mathbb{T}\mathsf{r}_{\texttt{CCv}}(\mathcal{P}) \subseteq \mathbb{T}\mathsf{r}_{\texttt{wCC}}(\mathcal{P})$ and $\mathbb{T}\mathsf{r}_{\texttt{CM}}(\mathcal{P}) \subseteq \mathbb{T}\mathsf{r}_{\texttt{wCC}}(\mathcal{P})$ hold by definition).

Let $\tau \in \mathbb{T}\mathsf{r}_{\texttt{wCC}}$ be a trace under $\texttt{wCC}$. Then, τ satisfies transactions isolation and causal delivery. It is important to notice that if τ has no write-write race then the contents of *store* at a given variable will contain a single value at any time during τ. This implies that *store* can be simulated by a single value memory which does not discard writes. Thus, we obtain a program semantics that is the same as the one for $\texttt{CM}$. Thus, τ is also a trace of $\mathcal{P}$ under $\texttt{CM}$. To prove that $\tau \in \mathbb{T}\mathsf{r}_{\texttt{CCv}}$, we also need to ensure that the transitive closure of store order in τ is acyclic which is enough to guarantee the existence of a total arbitration between transactions which is ensured by $\texttt{CCv}$ semantics. Suppose by contradiction that the transitive closure of store order is cyclic then there must exist a sequence of events $ev_1 \cdot ev_2 \cdot \ldots ev_n$ in τ such that $(ev_i, ev_{i+1}) \in \mathsf{WW}$, for all $1 \leq i \leq n-1$ and $(ev_n, ev_1) \in \mathsf{WW}$. Since τ has no write-write races then $(ev_i, ev_{i+1}) \in \mathsf{WW}$ implies that the issue events corresponding to ev_i and ev_{i+1} must be related by causal ordered (since the corresponding transactions must be causally related to prevent concurrency which will lead to write-write races for transactions that write to a common variable). For all i s.t. $1 \leq i \leq n-1$, let ev'_i and ev'_{i+1} denote these issue events then $(ev'_i, ev'_{i+1}) \in \mathsf{CO}$ which implies that the causal order CO is cyclic. This is a contradiction since it is not possible under $\texttt{wCC}$. Thus, there exists a total order between transactions in τ that includes both the causal order and the transitive closure of store order. Thus, τ is also a trace of $\mathcal{P}$ under $\texttt{CCv}$. $\square$

[7]Note that other cases such as $\tau' = \alpha \cdot \mathsf{isu}(p_1, t_1) \cdot \beta \cdot \mathsf{isu}(p_2, t_2) \cdot \mathsf{del}(p_2, t_2) \cdot \mathsf{del}(p_1, t_2) \cdot \mathsf{del}(p_1, t_1)$ implies that $\tau'' = \alpha \cdot \mathsf{isu}(p_1, t_1) \cdot \mathsf{del}(p_1, t_1) \cdot \beta \cdot \mathsf{isu}(p_2, t_2) \cdot \mathsf{del}(p_2, t_2) \cdot \mathsf{del}(p_1, t_2) \cdot \mathsf{del}(p_2, t_1)$ is a trace of $\mathcal{P}$ as well since all events in β are not causally dependent on t_1.

```
     p1:
                        p2:
t1: [ r = y   ||
                    t2: [y = 2]
      x = 1]
```

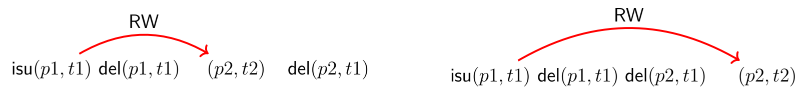

RW
isu(p1, t1) del(p1, t1) (p2, t2) del(p2, t1)
RW
isu(p1, t1) del(p1, t1) del(p2, t1) (p2, t2)

Figure 3.8: Two executions of the same serializable trace.

3.5 Program Robustness

Let $tr = (\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{STO})$ be a trace such that every event $\mathsf{isu}(p,t)$ in τ is immediately followed by all $\mathsf{del}(p',t)$ with $p' \in \mathbb{P}$. For simplicity, we write τ as a sequence of “atomic macro-events” (p,t) where (p,t) denotes a sequence $\mathsf{isu}(p,t) \cdot \mathsf{del}(p,t) \cdot \mathsf{del}(p_1,t) \cdot \ldots \cdot \mathsf{del}(p_n,t)$ with $\mathbb{P} = \{p, p_1, \ldots, p_n\}$. We say that t is *atomic*. Then, $(\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$ is a trace of a serializable execution as defined in Section 2.3. In Figure 3.7, $t3$ is atomic and we can use $(p2, t3)$ instead of $\mathsf{isu}(p2,t3) \cdot \mathsf{del}(p2,t3) \cdot \mathsf{del}(p1,t3)$.

The following result characterizes traces of serializable executions, and follows from previous works [25, 156] that considered a notion of history/trace that corresponds to our notion of transactional happens-before. The transactional happens-before of any trace under SER is acyclic, and conversely, any trace obtained under a weaker semantics $\mathsf{X} \in \{$CCv, CM, wCC$\}$ with an acyclic transactional happens-before can be transformed into a trace under SER by successive swaps of consecutive events in its summary, which are not related by happens-before (the happens-before relations remain the same). Indeed, note that multiple executions/traces can have the same (transactional) happens-before (an example for traces is given in Figure 3.8). In particular, it is possible that a trace tr produced by a variation of causal consistency has an acyclic transactional happens-before even though $\mathsf{isu}(p,t)$ events are not immediately followed by the corresponding $\mathsf{del}(p',t)$ events. However, tr would be equivalent, up to reordering of consecutive summary events that are not related by happens-before, to a trace serializable execution.

Theorem 3.3 ([25, 156]). *For any trace $tr \in \mathbb{T}r_{\mathtt{SER}}(\mathcal{P})$, the transactional happens-before of tr is acyclic. Moreover, for any trace $tr = (\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{STO}) \in \mathbb{T}r_{\mathsf{X}}(\mathcal{P})$ with $\mathsf{X} \in \{$CCv, CM, wCC$\}$, if the transactional happens-before of tr is acyclic, then there exists a permutation τ' of τ such that $(\tau', \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{STO}) \in \mathbb{T}r_{\mathtt{SER}}(\mathcal{P})$.*

As a consequence of Theorem 3.3, we define a trace tr to be *serializable* if it *has the same happens-before relations* as a trace of a serializable execution. Let $\mathbb{T}r_{\mathtt{SER}}(\mathcal{P})$ denote the set of serializable traces of a program $\mathcal{P}$.

We now consider the problem of checking whether the causally-consistent semantics of a program produces only serializable traces (it produces all serializable traces because every issue event can be immediately followed by all the corresponding store events).

Definition 3.2. *A program $\mathcal{P}$ is called* robust *against a semantics $\mathsf{X} \in \{\texttt{CCv}, \texttt{CM}, \texttt{wCC}\}$ relative to serializability iff $\mathbb{T}r_{\mathsf{X}}(\mathcal{P}) = \mathbb{T}r_{\texttt{SER}}(\mathcal{P})$.*

A trace $tr \in \mathbb{T}r_{\mathsf{X}}(\mathcal{P}) \setminus \mathbb{T}r_{\texttt{SER}}(\mathcal{P})$ is called a *robustness violation* (or *violation*, for short). By Theorem 3.3, the transactional happens-before HB_t of tr is cyclic.

3.6 Minimal Violations

We define a class of robustness violations called *minimal* violations. The particular shapes of these violations, that we determine through a series of results in this section, §3.7, and §3.8, enables a polynomial-time reduction of robustness checking to a reachability problem in a program running under serializability.

For simplicity, we use "atomic macro-events" (p, t) even in traces obtained under causal consistency (recall that this notation was introduced to simplify serializable traces), i.e., we assume that any sequence of events formed of an issue $\mathsf{isu}(p, t)$ followed immediately by all the store events $\mathsf{del}(p', t)$ is replaced by (p, t). Then, all the relations that held between an event ev of such a sequence and another event ev', e.g., $(ev, ev') \in \mathsf{PO}$, are defined to hold as well between the corresponding macro-event (p, t) and ev', e.g, $((p, t), ev') \in \mathsf{PO}$.

3.6.1 Happens-Before Through Relation

To decide if two events in a trace are "independent" (or commutative) we use the information about the existence of a happens-before relation between the events. If two events are not related by happens-before then they can be swapped while preserving the same happens-before. Thus, we extend the happens-before relation to obtain the *happens-before through* relation as follows:

Definition 3.3. *Let $\tau = \alpha \cdot a \cdot \beta \cdot b \cdot \gamma$ be a trace where a and b are events (or atomic macro events), and α, β, and γ are sequences of events (or atomic macro events) under a semantics $\mathsf{X} \in \{\texttt{CCv}, \texttt{CM}\}$. We say that a* happens-before *b* through *β if there is a non empty sub-sequence $c_1 \cdots c_n$ of β that* *satisfies:*

$$c_i \rightarrow_{\mathsf{HB}^1} c_{i+1} \quad \textit{for all } i \in [0, n]$$

where $c_0 = a$, $c_{n+1} = b$.

The following result shows that any two events in a trace which are not related via the happens-before through relation can be reordered without affecting the happens-before or they can be placed one immediately after the other.

Lemma 3.2. *Let* τ *be a trace of a program* $\mathcal{P}$ *under a semantics* $X \in \{\mathtt{CCv},\ \mathtt{CM}\}$*, and* a *and* b *be two events such that* $\tau = \alpha \cdot a \cdot \beta \cdot b \cdot \gamma$*. Then, one of the following holds:*

1. a *happens-before* b *through* β*;*
2. $\tau' = \alpha \cdot \beta_1 \cdot a \cdot b \cdot \beta_2 \cdot \gamma \in \mathbb{T}r_X(\mathcal{P})$ *where* $(a, b) \in \mathsf{HB}^1$ *has the same happens-before as* τ*;*
3. $\tau' = \alpha \cdot \beta_1 \cdot b \cdot a \cdot \beta_2 \cdot \gamma \in \mathbb{T}r_X(\mathcal{P})$ *has the same happens-before as* τ*.*

Proof. We prove that $\neg(1) \Rightarrow ((2)\ \mathsf{or}\ (3))$ using induction on the size of β.
Base case: If $|\beta| = 0$, then $\tau = \alpha \cdot a \cdot b \cdot \gamma$, which implies that a does not happen-before b through β (by definition, β cannot be empty). Thus, either a and b are HB^1-related, which corresponds to (2), or a and b are not HB^1-related, which implies that b can move to the left of a producing the trace $\tau' = \alpha \cdot b \cdot a \cdot \gamma$ that has the same happens-before as τ and that corresponds to (3).
Induction step: We assume that the lemma holds for $|\beta| \leq n$. Consider $\tau_{n+1} = \alpha \cdot a \cdot \beta \cdot b \cdot \gamma$ with $|\beta| = n + 1$. Consider c the last event in the sequence $\beta = \beta_1 \cdot c$. If a does not happen before b through β, then either a does not happen before c through β_1 and a and c are not HB^1-related, or c and b are not HB^1-related.

First case: suppose that a does not happen before c through β_1 and a and c are not HB^1-related. Using the induction hypothesis over τ_{n+1} with respect to a and c (since $|\beta_1| \leq n$) results in $\tau'_{n+1} = \alpha \cdot \beta_{11} \cdot c \cdot a \cdot \beta_{12} \cdot b \cdot \gamma$ that has the same happens-before as τ_{n+1}. We know that if a happens-before b through β_{12} then a happens-before b through β because β_{12} is a subset of β. Therefore, a does not happen-before b through β_{12}. Since $|\beta_{12}| \leq |\beta_1| \leq n$, then we can apply the induction hypothesis to τ'_{n+1} with respect to a and b which yields either $\tau''_{n+1} = \alpha \cdot \beta_{11} \cdot c \cdot \beta_{121} \cdot b \cdot a \cdot \beta_{122} \cdot \gamma$ which has the same happens-before as τ'_{n+1}, if a and b are not HB^1-related, or $\tau''_{n+1} = \alpha \cdot \beta_{11} \cdot c \cdot \beta_{121} \cdot a \cdot b \cdot \beta_{122} \cdot \gamma$ which has the same happens-before as τ'_{n+1}, otherwise.

Second case: suppose c and b are not HB^1-related. We apply the induction hypothesis to τ_{n+1} with respect to c and b, and we get $\tau'_{n+1} = \alpha \cdot a \cdot \beta_1 \cdot b \cdot c \cdot \gamma$ with the same happens-before as τ_{n+1}. As we already know that a does not happen before b through β then a does not happen before b through β_1. Subsequently by using the induction hypothesis over τ'_{n+1} with respect to a and b, we obtain $\tau''_{n+1} = \alpha \cdot \beta_{11} \cdot b \cdot a \cdot \beta_{12} \cdot c \cdot \gamma$ where τ''_{n+1} has the same happens-before as τ'_{n+1}, if a and b are not HB^1-related, or $\tau''_{n+1} = \alpha \cdot \beta_{11} \cdot a \cdot b \cdot \beta_{12} \cdot c \cdot \gamma$ where τ''_{n+1} has the same happens-before as τ'_{n+1}, otherwise. □

We show next that a robustness violation should contain at least an issue and a store event of the same transaction that are separated by another event that occurs after the issue and before the store and which is related to both via the happens-before relation. Otherwise, since any two events which are not related by happens-before could be swapped in order to derive a trace with the same happens-before, every store event could be swapped until it immediately follows the corresponding issue and the trace would be serializable.

Lemma 3.3. *Given a violation τ, there must exist a transaction t such that $\tau = \alpha \cdot \mathit{isu}(p,t) \cdot \beta \cdot \mathit{del}(p_0,t) \cdot \gamma$ and $\mathit{isu}(p,t)$ happens-before $\mathit{del}(p_0,t)$ through β.*

Proof. Assume by contradiction that the lemma does not hold. For every transaction t of τ suppose there exist $p' \in \mathbb{P}$ such that $\mathsf{del}(p',t)$ does not occur immediately after $\mathsf{isu}(p,t)$. Thus, $\tau = \alpha \cdot \mathsf{isu}(p,t) \cdot \beta \cdot \mathsf{del}(p',t) \cdot \gamma$, and $(\mathsf{isu}(p,t), \mathsf{del}(p',t)) \in \mathsf{STO} \subset \mathsf{HB}^1$. From Lemma 3.2, $\tau' = \alpha \cdot \beta_1 \cdot \mathsf{isu}(p,t) \cdot \mathsf{del}(p',t) \cdot \beta_2 \cdot \gamma$ has the same happens-before as τ (since $\mathsf{isu}(p,t)$ does not happens-before $\mathsf{del}(p',t)$ through β). Then, the trace τ^* where for every transaction t of τ the store events occur immediately after the issue event has the same happens-before as τ. Thus, τ^* is serializable which means that its HB_t is acyclic which contradicts the fact that τ is a violation. □

The transaction t in the trace τ above is called a *delayed* transaction. The happens-before constraints imply that t belongs to a transactional happens-before cycle in the trace. In the remainder of the chapter, when given a violation $\tau = \alpha \cdot \mathsf{isu}(p,t) \cdot \beta \cdot \mathsf{del}(p_0,t) \cdot \gamma$, we assume that t is the *first* delayed transaction in τ.

3.6.2 Minimal Violations

Given a trace $\tau = \alpha \cdot b \cdot \beta \cdot c \cdot \omega$ containing two events $b = \mathsf{isu}(p,t)$ and c, the *distance* between b and c, denoted by $d_\tau(b,c)$, is the number of events in β that are causally related to b, excluding events that correspond to the delivery of t, i.e., $d_\tau(b,c) = |\{d \in \beta \mid (b,d) \in \mathsf{CO} \ \wedge d \neq \mathsf{del}(p',t) \text{ for every } p' \in \mathbb{P}\}|$

The *number of delays* $\#(\tau)$ in a trace τ is the sum of all distances between issue and store events that originate from the same transaction:

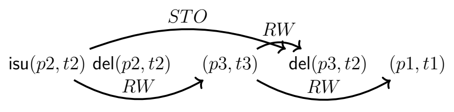
$$\#(\tau) = \sum_{\mathsf{isu}(p,t),\ \mathsf{del}(p',t)\ \in\ \tau} d_\tau(\mathsf{isu}(p,t), \mathsf{del}(p',t))$$

Definition 3.4 (Minimal violation)**.** *A robustness violation τ is called* minimal *if it has the least number of delays among all robustness violations (for a given program $\mathcal{P}$ and semantics $X \in \{\mathtt{wCC},\ \mathtt{CCv},\ \mathtt{CM}\}$).*

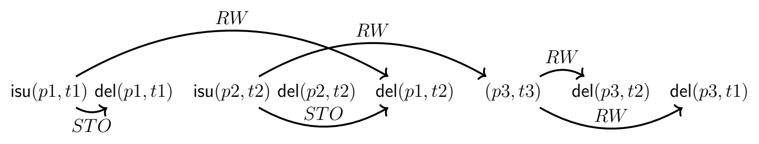
Remark 3.2. *It is important to note that a non-robust program can admit multiple minimal violations with different happens-before relations. For instance, Figure 3.9 pictures two minimal violations that do not have the same happens-before and both traces have 0 delays. In the trace in Figure 3.9b a single transaction is delayed while in the trace in Figure 3.9c two transactions are delayed and are not causally related. For the trace τ_1 in Figure 3.9b, we have that $\#(\tau_1) = d_{\tau_1}(\mathsf{isu}(p2,t2), \mathsf{del}(p2,t2)) + d_{\tau_1}(\mathsf{isu}(p2,t2), \mathsf{del}(p3,t2)) = 0$. For the trace τ_2 in Figure 3.9c, we have that $\#(\tau_2) = d_{\tau_2}(\mathsf{isu}(p1,t1), \mathsf{del}(p1,t1)) + d_{\tau_2}(\mathsf{isu}(p1,t1), \mathsf{del}(p3,t1)) + d_{\tau_2}(\mathsf{isu}(p2,t2), \mathsf{del}(p3,t2)) = 0$. Hence, the number of delays for both cases is 0.*

```
                                          p3:
    p1:              p2:
                                     t3: [z = 3
t1: [x = 1    || t2: [y = 2     ||
                                          r3 = x
     r1 = y]          r2 = z]
                                          r4 = y]
```

(a) A program.



(b) A minimal violation of (a).



(c) Another minimal violation of (a).

Figure 3.9: Example of two minimal violation traces that do not have the same happens-before relation (possible under both $\mathtt{CCv}$ and $\mathtt{CM}$). Both traces have the same number of delays which is equal to 0. The minimal violation in (b) contains a single delayed transaction ($t2$), and the minimal violation in (c) contains two delayed transactions ($t1$ and $t2$). For readability, we do not show all PO and STO transitions.

Given a minimal violation $\tau = \alpha \cdot \mathsf{isu}(p,t) \cdot \beta \cdot \mathsf{del}(p_0,t) \cdot \gamma$, the following lemma shows that we can assume w.l.o.g. that γ contains only store events from transactions that were issued before $\mathsf{del}(p_0,t)$ in τ.

Lemma 3.4. *Let $\tau = \alpha \cdot \mathsf{isu}(p,t) \cdot \beta \cdot \mathsf{del}(p_0,t) \cdot \gamma$ be a minimal violation such that $\mathsf{isu}(p,t)$ happens-before $\mathsf{del}(p_0,t)$ through β. Then, $\tau' = \alpha \cdot \mathsf{isu}(p,t) \cdot \beta \cdot \mathsf{del}(p_0,t) \cdot \gamma'$, such that γ' contains only store events from transactions that were issued before $\mathsf{del}(p_0,t)$ in τ, is also a minimal violation.*

Proof. The prefix $\alpha \cdot \mathsf{isu}(p,t) \cdot \beta \cdot \mathsf{del}(p_0,t)$ has a cyclic transactional happens-before and it is already a minimal violation independently of whether γ contains additional transactions. □

The following result shows that for every minimal violation, we can extract another minimal violation of the shape $\tau = \alpha \cdot \mathsf{isu}(p,t) \cdot \beta \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma$ such that $(\mathsf{isu}(p,t),(p',t')) \in \mathsf{HB}$, and $((p',t'),\mathsf{del}(p',t)) \in \mathsf{HB}^1$.

Lemma 3.5. *If $\mathcal{P}$ is a program that is not robust against some $\mathsf{X} \in \{\texttt{CCv}, \texttt{CM}, \texttt{wCC}\}$, then its set of traces under the semantics X must admit a minimal violation of the shape $\tau = \alpha \cdot \mathsf{isu}(p,t) \cdot \beta \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma$ such that $(\mathsf{isu}(p,t),(p',t')) \in \mathsf{HB}$ and $((p',t'),\mathsf{del}(p',t)) \in \mathsf{HB}^1$.*

Proof. Let $\tau = \alpha \cdot \mathsf{isu}(p,t) \cdot \beta \cdot \mathsf{del}(p_0,t) \cdot \gamma$ be a minimal violation of $\mathcal{P}$, such that $\mathsf{isu}(p,t)$ happens-before $\mathsf{del}(p_0,t)$ through β. By Lemma 3.4, we assume that γ contains only store events. We prove by induction on the size of β that $\mathcal{P}$ admits another minimal violation against X of the form $\tau' = \alpha' \cdot \mathsf{isu}(p_1,t_1) \cdot \beta' \cdot (p_2,t_2) \cdot \mathsf{del}(p_2,t_1) \cdot \gamma'$ such that $(\mathsf{isu}(p_1,t_1),(p_2,t_2)) \in \mathsf{HB}$, $((p_2,t_2),\mathsf{del}(p_2,t_1)) \in \mathsf{HB}^1$, and τ' is a permutation of a subsequence of τ.

Note that $\mathsf{isu}(p,t)$ happens-before $\mathsf{del}(p_0,t)$ through β implies that there exists a sub-sequence $c_1 \cdots c_n$ of β that satisfies: $c_i \rightarrow_{\mathsf{HB}^1} c_{i+1} \quad$ for all $i \in [0,n]$ where $c_0 = \mathsf{isu}(p,t)$, $c_{n+1} = \mathsf{del}(p_0,t)$. Then, we have three possibilities for c_n: (p',t'), $\mathsf{isu}(p',t')$, or $\mathsf{del}(p_0,t')$.

Base case: $|\beta| = 1$ implies that $\beta = c_n$. If $c_n = (p',t')$ then τ is a minimal violation s.t. $\mathsf{isu}(p,t) \rightarrow_{\mathsf{HB}} (p',t')$ and $(p',t') \rightarrow_{\mathsf{HB}^1} \mathsf{del}(p_0,t)$. If $c_n = \mathsf{isu}(p',t')$ then we regroup together the issue event $\mathsf{isu}(p',t')$ with its store events obtaining $\tau' = \alpha \cdot \mathsf{isu}(p,t) \cdot (p',t') \cdot \mathsf{del}(p_0,t) \cdot \gamma'$ to be a minimal violation as well (since the transactional happens-before of the trace resulting from reordering store events in $\mathsf{del}(p_0,t) \cdot \gamma'$ will always be cyclic). Since $(p',t') \rightarrow_{\mathsf{HB}^1} \mathsf{del}(p_0,t)$ implies that $(p',t') \rightarrow_{\mathsf{HB}^1} \mathsf{del}(p',t) \in \gamma$, then $\tau'' = \alpha \cdot \mathsf{isu}(p,t) \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma''$, where the two store events $\mathsf{del}(p',t)$ and $\mathsf{del}(p',t)$ are reordered, is a minimal violation. $c_n = \mathsf{del}(p_0,t')$ is not possible since t is the first delayed transaction in τ.

Induction step: We assume that the induction hypothesis holds for $|\beta| \leq m$. The case $c_n = (p', t')$ is trivial. If $c_n = \mathsf{isu}(p', t')$ then removing the issue events that occur after c_n will not impact the happens-before. Thus, we remove every issue and atomic marco event that occurs after $\mathsf{isu}(p', t')$ with all their store events and regroup together the event $\mathsf{isu}(p', t')$ with its store events obtaining $\tau' = \alpha \cdot \mathsf{isu}(p, t) \cdot \beta' \cdot (p', t') \cdot \mathsf{del}(p_0, t) \cdot \gamma'$ to be a minimal violation. Similar to before, $\tau'' = \alpha \cdot \mathsf{isu}(p, t) \cdot \beta' \cdot (p', t') \cdot \mathsf{del}(p', t) \cdot \gamma''$ is a minimal violation.

If $c_n = \mathsf{del}(p_0, t')$, then the corresponding issue event $\mathsf{isu}(p', t')$ must occur in β (α contains only atomic macro events because t is the first delayed transaction). If $\mathsf{isu}(p', t')$ does not happen before $\mathsf{del}(p_0, t')$ (or any store event of t' in $\beta \cdot \mathsf{del}(p_0, t) \cdot \gamma$) through a subsequence of β (resp., $\beta \cdot \mathsf{del}(p_0, t) \cdot \gamma$) then we can regroup together the issue and store events of t' and get that $\tau' = \alpha \cdot \mathsf{isu}(p, t) \cdot \beta' \cdot (p', t') \cdot \beta'' \cdot \mathsf{del}(p_0, t) \cdot \gamma'$ is a minimal violation. Otherwise, if $\mathsf{isu}(p', t')$ happens-before $\mathsf{del}(p_0, t')$ through a subsequence of β, then τ can be written as $\tau = \alpha \cdot \mathsf{isu}(p, t) \cdot \beta_1 \cdot \mathsf{isu}(p', t') \cdot \beta_2 \cdot \mathsf{del}(p_0, t') \cdot \beta_3 \cdot \mathsf{del}(p_0, t) \cdot \gamma$. Note that if there exists an issue event $\mathsf{isu}(p_1, t_1)$ in $\beta_1 \cdot \mathsf{isu}(p', t') \cdot \beta_2$ s.t. $(\mathsf{isu}(p_1, t_1), \mathsf{del}(p_0, t)) \in \mathsf{RW}$ (or $(\mathsf{isu}(p1, t1), \mathsf{del}(p_1, t)) \in \mathsf{RW}$) then similar to before the following trace $\tau' = \alpha \cdot \mathsf{isu}(p, t) \cdot \beta' \cdot (p_1, t_1) \cdot \mathsf{del}(p_0, t) \cdot \gamma'$ (resp., $\tau' = \alpha \cdot \mathsf{isu}(p, t) \cdot \beta' \cdot (p_1, t_1) \cdot \mathsf{del}(p_1, t) \cdot \gamma'$) is a minimal violation. Assume now that there does not exist an issue event $\mathsf{isu}(p_1, t_1)$. Then, let $\mathsf{isu}(p_2, t_2)$ be the first issue event in $\mathsf{isu}(p, t) \cdot \beta_1 \cdot \mathsf{isu}(p', t')$ s.t. $\tau = \alpha \cdot \mathsf{isu}(p, t) \cdot \beta_1' \cdot \mathsf{isu}(p_2, t_2) \cdot \beta_2' \cdot \mathsf{del}(p_3, t_2) \cdot \beta_3' \cdot \gamma$ and $\mathsf{isu}(p_2, t_2)$ happens-before $\mathsf{del}(p_3, t_2)$ through β_2' and s.t. for every issue event in $\mathsf{isu}(p, t) \cdot \beta_1'$ of a transaction t_4 there does not exist an event in $\beta_1' \cdot \mathsf{isu}(p_2, t_2) \cdot \beta_2'$ that reads from a variable that t_4 overwrites. We can remove every issue event and atomic marco event which occur after $\mathsf{del}(p_3, t_2)$ with all related stores: $\tau' = \alpha \cdot \mathsf{isu}(p, t) \cdot \beta_1' \cdot \mathsf{isu}(p_2, t_2) \cdot \beta_2' \cdot \mathsf{del}(p_3, t_2) \cdot \gamma'$ where γ' contains only store events is a minimal violation. Then, not delaying the transactions in $\mathsf{isu}(p, t) \cdot \beta_1'$ does not affect the reads in $\beta_1' \cdot \mathsf{isu}(p_2, t_2) \cdot \beta_2'$, and thus, we get that $\tau'' = \alpha \cdot (p, t) \cdot \beta_1'' \cdot \mathsf{isu}(p_2, t_2) \cdot \beta_2'' \cdot \mathsf{del}(p_3, t_2) \cdot \gamma''$, where t_2 is the first delayed transaction in τ'' and $\mathsf{isu}(p_2, t_2)$ happens-before $\mathsf{del}(p_3, t_2)$ through β_2'', is a minimal violation. Note that $|\beta_2''| < |\beta| = m + 1$, and we can apply the induction hypothesis to τ'' and conclude the proof. □

Next, we show that a program which is not robust against `CCv` or `CM` admits violations of particular shapes. For the remainder of the chapter, we write a minimal violation in the shape $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p, t) \cdot \beta \cdot (p', t') \cdot \mathsf{del}(p', t) \cdot \gamma_{\mathsf{S}}$ to say that all the events in the sequence α_{A} are atomic macro events and all the events in the sequence γ_{S} are store events. As before, we assume that t is the first delayed transaction in τ, and by Lemma 3.5, we assume that $(\mathsf{isu}(p, t), (p', t')) \in \mathsf{HB}$ and

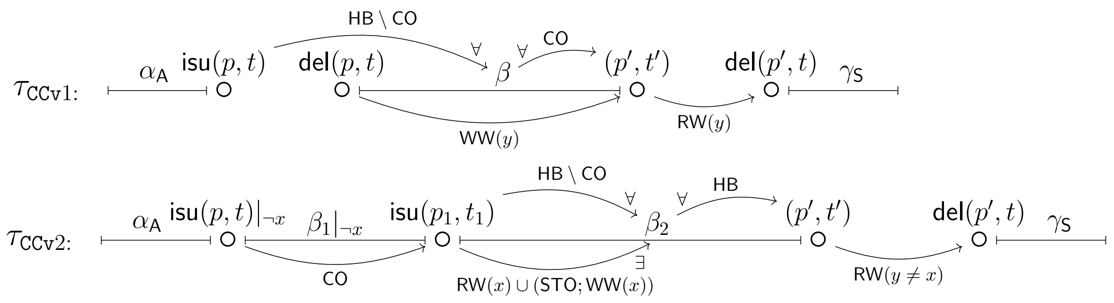
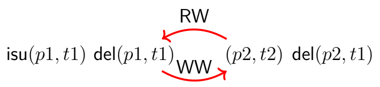
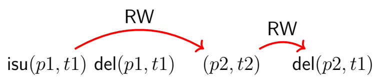


Figure 3.10: Robustness violation patterns under $\texttt{CCv}$. We use $a \xrightarrow{R}{}^{\forall} \beta$ to denote $\forall\, b \in \beta.\ (a,b) \in R$. We use $\beta_1|_{\neg x}$ to say that all delayed transactions in β_1 do not access x. For violation $\tau_{\texttt{CCv1}}$, t is the only delayed transaction. For $\tau_{\texttt{CCv2}}$, all delayed transactions are in $\mathsf{isu}(p,t) \cdot \beta_1 \cdot \mathsf{isu}(p_1,t_1)$ and they form a causality chain that starts at $\mathsf{isu}(p,t)$ and ends at $\mathsf{isu}(p_1,t_1)$.


RW
$\mathsf{isu}(p1,t1)$ $\mathsf{del}(p1,t1)$
WW
$(p2,t2)$ $\mathsf{del}(p2,t1)$

(a) Violation of LU program in Figure 3.2a.



(b) Violation of SB program in Figure 3.2b.

Figure 3.11: (a) A $\tau_{\texttt{CCv1}}$ violation where $\beta_2 = \epsilon$, $\gamma_{\mathsf{S}} = \epsilon$, and t and t' correspond to $t1$ and $t2$. (b) A $\tau_{\texttt{CCv2}}$ (resp., $\tau_{\texttt{CM2}}$) violation where t and t_1 coincide and correspond to $t1$. Also, $\beta_1 = \epsilon$, $\beta_2 = (p2,t2)$, $\gamma_{\mathsf{S}} = \epsilon$, such that $(\mathsf{isu}(p1,t1),(p2,t2)) \in \mathsf{RW}(y)$ and $((p2,t2),\mathsf{del}(p2,t1)) \in \mathsf{RW}(x)$. In all traces, we show only the relations that are part of the happens-before cycle.

$((p',t'),\mathsf{del}(p',t)) \in \mathsf{HB}^1$.

3.7 Robustness Violations Under Causal Convergence

In this section, we present a precise characterization of minimal violations under $\texttt{CCv}$. In particular, we show that in these violations, the first delayed transaction (which must exist by Lemma 3.3) is followed by a possibly-empty sequence of delayed transactions that form a "causality chain", i.e., the issue of every new delayed transaction is causally ordered after the issue of the first delayed transaction. Also, we show that the issue event of the last delayed transaction happens-before an event of another transaction that reads a variable updated by the first delayed transaction (which implies a cycle in the transactional happens-before). This characterization will allow us to build a monitor for detecting the existence of robustness violations that is linear in the size of the input program.

Next, we give a precise definition of the "causality chain". It consists of a sequence of issue events such that the first issue is causally ordered before every other issue event and every issued transaction is delivered to the process executing the next issue event in the chain, before this issue event executes.

Definition 3.5. *We say that a sequence of issue events $ev_1 \cdot ev_2 \cdot \ldots ev_n$ forms a* causality chain *that starts with ev_1 and ends at ev_n in a trace τ if the followings hold:*

1. $(ev_1, ev_i) \in \mathsf{CO}$*, for all* $2 \leq i \leq n$
2. *for all $1 \leq i \leq n-1$ such that $ev_i = \textsf{isu}(p_i, t_i)$, $ev_{i+1} = \textsf{isu}(p_{i+1}, t_{i+1})$, the store event $\textsf{del}(p_{i+1}, t_i)$ occurs before the issue event ev_{i+1} in τ.*

The characterization of robustness violations under `CCv` is stated in the following theorem and pictured in Figure 3.10.

Theorem 3.4. *A program $\mathcal{P}$ is not robust under* `CCv` *iff there exists a minimal violation in $\mathbb{T}r(\mathcal{P})_{\mathtt{CCv}}$ of one of the following forms:*

1. $\tau_{\mathtt{CCv1}} = \alpha_{\mathsf{A}} \cdot \textsf{isu}(p,t) \cdot \textsf{del}(p,t) \cdot \beta_2 \cdot (p',t') \cdot \textsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$ *where:*
   - *(a)* $\textsf{isu}(p,t)$ *is the issue of the first and only delayed transaction (Lemma 3.6);*
   - *(b)* $\exists\ y.$ *s.t.* $(\textsf{del}(p,t), (p',t')) \in \mathsf{WW}(y)$ *and* $((p',t'), \textsf{del}(p',t)) \in \mathsf{RW}(y)$ *(Lemma 3.6);*
   - *(c)* $\forall\ a \in \beta_2.\ (\textsf{isu}(p,t), a) \in \mathsf{HB} \setminus \mathsf{CO}$ *and* $(a, (p',t')) \in \mathsf{CO}$ *(Lemma 3.6).*
2. $\tau_{\mathtt{CCv2}} = \alpha_{\mathsf{A}} \cdot \textsf{isu}(p,t) \cdot \beta_1 \cdot \textsf{isu}(p_1,t_1) \cdot \beta_2 \cdot (p',t') \cdot \textsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$ *where:*
   - *(a)* $\textsf{isu}(p,t)$ *and* $\textsf{isu}(p_1,t_1)$ *are the issues of the first and last delayed transactions (Lemmas 3.6 and 3.7);*
   - *(b) the issues of all delayed transactions are in β_1 and are included in a causality chain that starts with $\textsf{isu}(p,t)$ and ends at $\textsf{isu}(p_1,t_1)$ (Lemma 3.7);*
   - *(c) for every $a \in \beta_2$, we have that $(\textsf{isu}(p_1,t_1), a) \in \mathsf{HB} \setminus \mathsf{CO}$ and $(a, (p',t')) \in \mathsf{HB}$ (Lemma 3.6);*
   - *(d) there exist $a \in \beta_2 \cdot (p',t')$, x, and y s.t. $x \neq y$, $(\textsf{isu}(p_1,t_1), a) \in \mathsf{RW}(x) \cup (\mathsf{STO}; \mathsf{WW}(x))$, $(a, (p',t')) \in \mathsf{HB}?$[8], and $((p',t'), \textsf{del}(p',t)) \in \mathsf{RW}(y)$ (Lemma 3.6);*

[8]HB? is the reflexive closure of HB.

(e) all delayed transactions in $\mathsf{isu}(p,t)\cdot\beta_1$ do not access the variable x (Lemma 3.9).

Above, τ_{CCv1} contains a single delayed transaction while τ_{CCv2} may contain arbitrarily many delayed transactions. In τ_{CCv1} the store event $\mathsf{del}(p,t)$ of the only delayed transaction happens before (p',t') which is conflicting with t, thus resulting in a cycle in the transactional happens-before. In τ_{CCv2} the issue event of the last delayed transaction t_1, which is causally ordered after the issue of the first delayed transaction t, happens before (p',t') which is conflicting with t, thus resulting in a cycle in the transactional happens-before as well. The theorem above allows $\alpha_{\mathsf{A}}=\epsilon$, $\beta_1=\epsilon$, $\beta_2=\epsilon$, $\beta=\epsilon$, $\gamma_{\mathsf{S}}=\epsilon$, $p=p_1$, $t=t_1$, and t_1 to be a read-only transaction. Figure 3.11a and Figure 3.11b show two violations under CCv where such equalities hold. If t_1 is a read-only transaction then $\mathsf{isu}(p_1,t')$ has the same effect as (p_1,t_1) since t_1 does not contain writes.

The minimality of the violation enforces the constraints stated above. For example, in the context of τ_{CCv2}, the delayed transactions in β_1 cannot create a cycle in the transactional happens-before (otherwise, there exists a sequence of store events γ'_{S} such that $\alpha_{\mathsf{A}}\cdot\mathsf{isu}(p,t)\cdot\mathsf{del}(p,t)\cdot\beta_1\cdot\mathsf{del}(p_0,t)\cdot\gamma'_{\mathsf{S}}$ is a violation with a smaller measure, which contradicts minimality). Moreover, (c) implies that β_2 contains no stores of delayed transactions from β_1. If this were the case, then these stores can either be reordered after $\mathsf{del}(p',t)$ or if this is not possible due to happens-before constraints, then there would exist an issue event which is after such a store in the happens-before order and thus causally after $\mathsf{isu}(p,t)$, which would contradict the fact that $\mathsf{isu}(p_1,t_1)$ is the last issue event in τ that is causally ordered after $\mathsf{isu}(p,t)$. Also, if it were to have a delayed transaction t_2 in β_2 (resp., β for τ_{CCv1}), then it is possible to remove some transaction (the issue and all its store events) from the original trace and obtain a new violation trace with a smaller number of delays. For instance, in the case of β_2, if $t_1\neq t$, then we can remove the events of the last delayed transaction (i.e., t_1), that is causally related to $\mathsf{isu}(p,t)$, since all events in $\beta_2\cdot\mathsf{del}(p_0,t)\cdot\gamma_{\mathsf{S}}$ neither read from the writes of t_1 nor are issued by the same process as t_1 (because of the $\mathsf{HB}\setminus\mathsf{CO}$ relation between events β_2 and $\mathsf{isu}(p_1,t_1)$). The resulting trace is still a robustness violation (because of the transactional happens-before cycle involving t_2 since it is delayed in β_2) but with a smaller measure. Note that all processes that delayed transactions, stop executing new transactions in β_2 (resp., β) because of the relation $\mathsf{HB}\setminus\mathsf{CO}$, shown in Figure 3.10, between the delayed transaction t_1 (resp., t) and events in β_2 (resp., β).

In the following we give a series of lemmas that collectively imply Theorem 3.4. Next lemma gives the decomposition of minimal violations under CCv into two possible patterns. It also characterizes the nature of the happens-before dependencies in these traces. For instance, we show that the last dependency in the happens-before cycle is always a conflict dependency. The lemma proof starts with a minimal violation as characterized in Lemma 3.5 and uses induction to show that we can always obtain a minimal violation which follows one of the two patterns. The induction is based on the size of the sequence of events between the issue and delayed store events of the first delayed transaction (the sequence β in Lemma 3.5).

Lemma 3.6. *If $\mathcal{P}$ is a program that is not robust under* CCv*, then it must admit a minimal violation τ that satisfies one of the following:*

1. $\tau = \alpha_{\mathsf{A}} \cdot \mathit{isu}(p,t) \cdot \mathit{del}(p,t) \cdot \beta \cdot (p',t') \cdot \mathit{del}(p',t) \cdot \gamma_{\mathsf{S}}$ *where:*
   - *(a)* $\exists\ y.\ s.t.\ (\mathit{del}(p,t),(p',t')) \in \mathsf{WW}(y)$ *and* $((p',t'),\mathit{del}(p',t)) \in \mathsf{RW}(y)$*;*
   - *(b)* $\forall\ a \in \beta.\ (\mathit{isu}(p,t),a) \in \mathsf{HB} \setminus \mathsf{CO}$ *and* $(a,(p',t')) \in \mathsf{CO}$.
2. $\tau = \alpha_{\mathsf{A}} \cdot \mathit{isu}(p,t) \cdot \beta_1 \cdot \mathit{isu}(p_1,t_1) \cdot \beta_2 \cdot (p',t') \cdot \mathit{del}(p',t) \cdot \gamma_{\mathsf{S}}$ *where:*
   - *(a)* $\mathit{isu}(p_1,t_1)$ *is the last issue event in* $\{c \in \beta \mid (\mathit{isu}(p,t),c) \in \mathsf{CO}\}$*;*
   - *(b)* $\exists\ x,\ y,$ *and* $a \in \beta_2\cdot(p',t')$ *s.t.* $(\mathit{isu}(p_1,t_1),a) \in \mathsf{RW}(x) \cup (\mathsf{STO};\mathsf{WW}(x))$, $(a,(p',t')) \in \mathsf{HB}?$, *and* $((p',t'),\mathit{del}(p',t)) \in \mathsf{RW}(y)$*;*
   - *(c)* $\forall\ a \in \beta_2.\ (\mathit{isu}(p_1,t_1),a) \in \mathsf{HB} \setminus \mathsf{CO}$ *and* $(a,(p',t')) \in \mathsf{HB}$.

Proof. Let $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$ be a minimal violation under CCv (cf. Lemma 3.5). We prove by induction on the size of β that there exists a minimal violation trace τ' that satisfies (1) or (2) and τ' is obtained from τ. By the definition of the happens-before $((p',t'),\mathsf{del}(p',t)) \in \mathsf{HB}^1$ implies that $((p',t'),\mathsf{del}(p',t)) \in \mathsf{RW} \cup \mathsf{WW}$. Since t' was issued after t in τ, then based on the total order of timestamps under CCv, we cannot have $((p',t'),\mathsf{del}(p',t)) \in \mathsf{WW}$. Then, there must exist y s.t. $((p',t'),\mathsf{del}(p',t)) \in \mathsf{RW}(y)$.

Base case: $|\beta| = 0$. Since $(\mathsf{isu}(p,t),(p',t')) \in \mathsf{HB}$ then from the definition of the happens-before the only possible relation is $(\mathsf{isu}(p,t),(p',t')) \in \mathsf{RW}$. Thus, there must exist x s.t. $(\mathsf{isu}(p,t),(p',t')) \in \mathsf{RW}(x)$. If $x = y$ then both t and t' write to x. Thus, by reordering the store event $\mathsf{del}(p,t) \in \gamma_{\mathsf{S}}$ to occur just after the corresponding issue event we get $\tau' = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \mathsf{del}(p,t) \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma'_{\mathsf{S}}$ is also a minimal violation where $(\mathsf{del}(p,t),(p',t')) \in \mathsf{WW}(x)$ (since t was issued before t' and both write to x) and $((p',t'),\mathsf{del}(p',t)) \in \mathsf{RW}(x)$. τ' satisfies the first case of the lemma. If $x \neq y$ then $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$ where there exist x and y s.t. $x \neq y$, $(\mathsf{isu}(p,t),(p',t')) \in \mathsf{RW}(x)$, and $((p', t'), \mathsf{del}(p', t)) \in \mathsf{RW}(y)$ satisfies the second case of the lemma where t and t_1 coincide and a corresponds to (p', t').

Induction step: We assume the induction hypothesis holds for $|\beta| \leq m$. Let $\sigma = \{c \in \beta \mid (\mathsf{isu}(p, t), c) \in \mathsf{CO}\}$, we will consider the following three possible cases:

First, assume that σ is empty. Since $(\mathsf{isu}(p, t), (p', t')) \in \mathsf{HB}$ then there must exist $a \in \beta \cdot (p', t')$ s.t. $(\mathsf{isu}(p, t), a) \in \mathsf{HB}^1$ and $(a, (p', t')) \in \mathsf{HB}?$. σ is empty implies that β does not contain events that are related to $\mathsf{isu}(p, t)$ through CO (which includes $\mathsf{PO} \cup \mathsf{WR} \cup \mathsf{STO}$), therefore, $(\mathsf{isu}(p, t), a) \in \mathsf{WW} \cup \mathsf{RW}$. It is impossible to have $(\mathsf{isu}(p, t), a) \in \mathsf{WW}$ since $\mathsf{isu}(p, t)$ does not contain writes. Thus, there must exist x s.t. $(\mathsf{isu}(p, t), a) \in \mathsf{RW}(x)$. If $x = y$ then both the transaction of the event a, denoted t_2, and t write to x. We consider the two cases of $(a, (p', t')) \in \mathsf{HB}?$: i) $a = (p', t')$ (i.e., $t_2 = t'$), and ii) $(a, (p', t')) \in \mathsf{HB}$. Assume $a = (p', t')$ then by reordering the store event $\mathsf{del}(p, t) \in \tau$ to occur just after the corresponding issue event (since the events in β are not causally related to $\mathsf{isu}(p, t)$) we get $\tau' = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p, t) \cdot \mathsf{del}(p, t) \cdot \beta \cdot (p', t') \cdot \mathsf{del}(p', t) \cdot \gamma'_{\mathsf{S}}$ is also a minimal violation where $(\mathsf{del}(p, t), (p', t')) \in \mathsf{WW}(x)$ (since t was issued before t' and both write to x) and $((p', t'), \mathsf{del}(p', t)) \in \mathsf{RW}(x)$. In τ' we remove all events in β that are not causally ordered before (p', t') since they do not contribute to the happens-before cycle. We obtain a new violation trace that satisfies the first case of the lemma. Assume now that $(a, (p', t')) \in \mathsf{HB}$. This implies that $\mathsf{isu}(p_2, t_2) \in \beta$ happens-before (p', t') (since a is an event t_2). Since both t_2 and t write to x and t occurs before t_2 in τ then from the definition of store and conflict relations $((p', t'), \mathsf{del}(p', t)) \in \mathsf{RW}(x)$ implies that $((p', t'), \mathsf{del}(p', t_2)) \in \mathsf{RW}(x)$. Also, since in β we do not have events that are causally related to $\mathsf{isu}(p, t)$ then let τ' be the trace resulting from removing all events of t in τ: $\tau' = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p_2, t_2) \cdot \beta' \cdot (p', t') \cdot \mathsf{del}(p', t_2) \cdot \gamma'_{\mathsf{S}}$ where τ' is a subsequence of τ and β' is a subsequence of β. τ' is a minimal violation as well since it was obtained from τ by just removing events and $(\mathsf{isu}(p_2, t_2), (p', t')) \in \mathsf{HB}$ and $((p', t'), \mathsf{del}(p', t_2)) \in \mathsf{RW}(x)$. Since $|\beta'| \leq m$ then we can apply the induction hypothesis on τ'. If $x \neq y$ we get that in τ, $(\mathsf{isu}(p, t), a) \in \mathsf{RW}(x)$ and $((p', t'), \mathsf{del}(p', t)) \in \mathsf{RW}(y)$ which satisfies the second case of the lemma.

Second, assume that σ is not empty and all the elements of σ are store events. Since t is the first delayed transaction in τ then all stores in σ are stores of t. Then, following the same analogy as before there must exist x and an event $a \in \beta \cdot (p', t')$ that is not a store event of t s.t. $(\mathsf{isu}(p, t), a) \in (\mathsf{STO}; \mathsf{WW}(x)) \cup \mathsf{RW}(x)$ and $(a, (p', t')) \in \mathsf{HB}?$. Similar to before we consider the two cases $x = y$ and $x \neq y$ and apply the induction hypothesis in the first case.

Third, assume that σ is not empty and $\mathsf{isu}(p_1, t_1)$ is the last issue event in σ, i.e., $\beta = \beta_1 \cdot$

$\mathsf{isu}(p_1, t_1) \cdot \beta_2$ where all the events in β_2 are either stores of transactions that are causally related to $\mathsf{isu}(p, t)$ (we can reorder these stores to be part of γ_{S} except the store $\mathsf{del}(p_1, t_1)$) or other events that are not causally related to $\mathsf{isu}(p, t)$. We also have that $\mathsf{isu}(p, t)$ is causally ordered before $\mathsf{isu}(p_1, t_1)$. Since $(\mathsf{isu}(p, t), (p', t')) \in \mathsf{HB}$ then $(\mathsf{isu}(p_1, t_1), (p', t')) \in \mathsf{HB}$, otherwise, we remove $\mathsf{isu}(p_1, t_1)$ and all related store events from τ and the resulting trace is still a violation and it has less delays since $\mathsf{isu}(p, t)$ was not delayed after $\mathsf{isu}(p_1, t_1)$ in the trace. Thus, $(\mathsf{isu}(p_1, t_1), (p', t')) \in \mathsf{HB}$. Similar to before we obtain that there exist x and an event $a \in \beta_2 \cdot (p', t')$ that is not a store event of t_1 s.t. $(\mathsf{isu}(p_1, t_1), a) \in (\mathsf{STO}; \mathsf{WW}(x)) \cup \mathsf{RW}(x)$ and $(a, (p', t')) \in \mathsf{HB}?$. If $x = y$ then both the transaction of the event a, denoted t_2, and t write to x. Thus, $(\mathsf{isu}(p, t), a) \in \mathsf{STO}; \mathsf{WW}(x)$. Then, since the events in $\beta_2 \cdot (p', t')$ do not causally depend on $\mathsf{isu}(p_1, t_1)$ then we can remove the events of t_1 and obtain τ' where $(\mathsf{isu}(p, t), a) \in \mathsf{STO}; \mathsf{WW}(x)$, $(a, (p', t')) \in \mathsf{HB}?$, and $((p', t'), \mathsf{del}(p', t)) \in \mathsf{RW}(y)$ where t was not delayed after $\mathsf{isu}(p_1, t_1)$ in the trace, which means that τ' has less delays than τ (a contradiction to τ being a minimal violation). Therefore, we must have $x \neq y$ s.t. $(\mathsf{isu}(p_1, t_1), a) \in (\mathsf{STO}; \mathsf{WW}(x)) \cup \mathsf{RW}(x)$ and $(a, (p', t')) \in \mathsf{HB}?$ and $((p', t'), \mathsf{del}(p', t)) \in \mathsf{RW}(y)$ which satisfies the second case of the lemma.

□

We use $\mathbb{T}\mathsf{ccv1}$ and $\mathbb{T}\mathsf{ccv2}$ to denote the class of minimal violations that satisfy the first and second case in Lemma 3.6, respectively. The following lemma shows that we can always obtain a minimal violation trace in either $\mathbb{T}\mathsf{ccv1}$ or $\mathbb{T}\mathsf{ccv2}$ where β and β_2 contain no delayed transactions, respectively. We distinguish two cases in the proof: i) a minimal violation in $\mathbb{T}\mathsf{ccv2}$ where t and t_1 are distinct transactions, and ii) a minimal violation in $\mathbb{T}\mathsf{ccv1}$ or in $\mathbb{T}\mathsf{ccv2}$ where t and t_1 coincide. In the first case, we show that if it were to have a delayed transaction in β_2, then it is possible to remove some transaction from τ that is causally dependent on the first delayed transaction in τ, and obtain a new violation with a smaller number of delays (which contradicts the minimality assumption). The second case is proved by induction on the size of β (note that if t and t_1 coincide, then $\beta = \beta_2$) where the base case is trivial (i.e., $\beta = \epsilon$), and in the induction step, we show that if it were to have a delayed transaction in β then we can remove one of the delayed transactions in the trace and obtain another violation with the same number of delays as the original violation and for which we can apply the induction hypothesis.

Lemma 3.7. *Let τ be a minimal violation in $\mathbb{T}\mathsf{ccv1}$ or $\mathbb{T}\mathsf{ccv2}$. Then, there exist a violation τ_1 in $\mathbb{T}\mathsf{ccv1}$ where β contains no delayed transactions or a violation τ_2 in $\mathbb{T}\mathsf{ccv2}$ where β_2 contains no* *delayed transactions.*

Proof. We consider two cases: i) τ in $\mathbb{T}\mathsf{ccv2}$ where t_1 and t are two distinct transactions, ii) τ in $\mathbb{T}\mathsf{ccv1}$ or τ in $\mathbb{T}\mathsf{ccv2}$ where t_1 and t coincide. We prove the first case by contradiction and the second case by induction on the size β (we abused terminology here and considered $\beta_2 = \beta$ since $\beta_1 = \epsilon$ in the second case).

First case: let $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta_1 \cdot \mathsf{isu}(p_1,t_1) \cdot \beta_2 \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$ and suppose by contradiction that β_2 contains a delayed transaction t_0 issued by a process $q \neq p$. W.l.o.g., we assume that the delayed store events of t_0 occur in β_2. Thus, $\beta_2 = \beta_{21} \cdot \mathsf{isu}(q,t_0) \cdot \beta_{22} \cdot \mathsf{del}(q',t_0) \cdot \beta_{23}$ and $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta_1 \cdot \mathsf{isu}(p_1,t_1) \cdot \beta_{21} \cdot \mathsf{isu}(q,t_0) \cdot \beta_{22} \cdot \mathsf{del}(q',t_0) \cdot \beta_{23} \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$. In τ, $\mathsf{isu}(q,t_0)$ happens-before $\mathsf{del}(q',t_0)$ through β_{22}. Hence, we deduce that we can get a robustness violation when the event $\mathsf{del}(q',t_0)$ is executed, thus we can remove all issued transactions from $\beta_{23} \cdot (p',t')$ except stores of already issued transactions and we obtain: $\tau' = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta_1 \cdot \mathsf{isu}(p_1,t_1) \cdot \beta_{21} \cdot \mathsf{isu}(q,t_0) \cdot \beta_{22} \cdot \mathsf{del}(q',t_0) \cdot \beta'_{23} \cdot \mathsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$ which is a minimal violation because $\mathsf{isu}(q,t_0)$ happens-before $\mathsf{del}(q',t_0)$ through β_{22} and its number of delays is less or equal to the one of τ. We know that in $\beta_{21} \cdot \mathsf{isu}(q,t_0) \cdot \beta_{22} \cdot \mathsf{del}(q',t_0) \cdot \beta'_{23} \cdot \mathsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$ there are no transactions from the process p_1 or that see the effect of transactions from p_1 (because of the $\mathsf{HB} \setminus \mathsf{CO}$ relation between events β_2 and $\mathsf{isu}(p_1,t_1)$). Therefore, $\mathsf{isu}(p_1,t_1)$ is the last issued transaction from p_1 and we do not have any transaction in τ' that depends on it. Thus, we can remove $\mathsf{isu}(p_1,t_1)$ and we obtain the following trace: $\tau'' = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta_1 \cdot \beta_{21} \cdot \mathsf{isu}(q,t_0) \cdot \beta_{22} \cdot \mathsf{del}(q',t_0) \cdot \beta'_{23} \cdot \mathsf{del}(p',t) \cdot \gamma'_{\mathsf{S}}$, which is a robustness violation because $\mathsf{isu}(q,t_0)$ happens-before $\mathsf{del}(q',t_0)$ through β_{22}. τ'' has less delays than τ' ($\mathsf{del}(p',t)$ was not delayed after $\mathsf{isu}(p_1,t_1)$ which was removed), which is a contradiction to the fact that τ is a minimal violation.

Second case: let $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$. We show by induction that we can construct either τ_1 in $\mathbb{T}\mathsf{ccv1}$ where β of τ_1 contains no delayed transactions or τ_2 in $\mathbb{T}\mathsf{ccv2}$ where β_2 of τ_2 contains no delayed transactions.

Base case: $|\beta| = 0$ is trivial.

Induction step: We assume that the induction hypothesis holds for $|\beta| \leq m$. Let t_0 be the first delayed transaction in β. Similar to before, we assume w.l.o.g. that the delayed store events of t_0 occurs in β. Thus, $\beta = \beta_{01} \cdot \mathsf{isu}(q,t_0) \cdot \beta_{02} \cdot \mathsf{del}(q',t_0) \cdot \beta_{03}$ and $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta_{01} \cdot \mathsf{isu}(q,t_0) \cdot \beta_{02} \cdot \mathsf{del}(q',t_0) \cdot \beta_{03} \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$ where $\mathsf{isu}(q,t_0)$ happens-before $\mathsf{del}(q',t_0)$ through β_{02}. Using the same arguments as before, we can remove the event $\mathsf{isu}(p,t)$, its related stores in τ, and all issued transactions in $\beta_{03} \cdot (p', t')$. We obtain: $\tau' = \alpha'_{\mathsf{A}} \cdot \mathsf{isu}(q, t_0) \cdot \beta_{02} \cdot \mathsf{del}(q', t_0) \cdot \gamma'_{\mathsf{S}}$ where $\alpha'_{\mathsf{A}} = \alpha_{\mathsf{A}} \cdot \beta_{01}$ and $\mathsf{isu}(q, t_0)$ happens-before $\mathsf{del}(q', t_0)$ through β_{02}. τ' is a robustness violation, and it has the same number of delays as τ. We now consider two possible case of τ': i) τ' is in $\mathbb{T}\mathsf{ccv2}$ where t_0 and t_{01}, the last delayed transaction causally dependent on $\mathsf{isu}(q, t_0)$ in τ', are two distinct transactions, or ii) τ in $\mathbb{T}\mathsf{ccv1}$ or τ in $\mathbb{T}\mathsf{ccv2}$ where t_0 and t_{01} coincide. From the first part of the proof, it is guaranteed that in the first case there are no delayed transactions after t_{01}. For the second case, we use the induction hypothesis since $|\beta_{02}| \leq m$ (β_{02} is a strict subsequence of β). □

We have now showed all the necessary characterizations for minimal violations that fall under the first pattern (i.e., $\mathbb{T}\mathsf{ccv1}$). In the rest of this section, we focus on minimal violations that fall under the second pattern (i.e., $\mathbb{T}\mathsf{ccv2}$). In particular, we look at minimal violations in $\mathbb{T}\mathsf{ccv2}$ where t and t_1 are distinct transactions. In the following lemma, we show that for these minimal violations the issue events of delayed transactions in $\mathsf{isu}(p, t) \cdot \beta_1 \cdot \mathsf{isu}(p_1, t_1)$ constitute a causality chain. Our proof can be decomposed to two parts. In the first part, we show that we cannot have an issue event of a delayed transaction in $\beta_1 \cdot \mathsf{isu}(p_1, t_1)$ that is not causally dependent on $\mathsf{isu}(p, t)$. We prove this by showing that if this were possible then we can remove a transaction that is causally dependent on one of the two delayed transactions and obtain a new violation trace with less delays than the original violation (which contradicts the minimality assumption). For the second part, we show that for a given minimal violation, we can construct a happens-before equivalent trace where for every two successive issue events of delayed transactions in $\mathsf{isu}(p, t) \cdot \beta_1 \cdot \mathsf{isu}(p_1, t_1)$, the transaction in the first issue is delivered to the process executing the second issue before this event happens.

Lemma 3.8. *Let $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p, t) \cdot \beta_1 \cdot \mathsf{isu}(p_1, t_1) \cdot \beta_2 \cdot (p', t') \cdot \mathsf{del}(p', t) \cdot \gamma_{\mathsf{S}}$ be a minimal violation in $\mathbb{T}\mathsf{ccv2}$ s.t $t \neq t_1$ and β_2 contains no delayed transactions (cf. Lemma 3.7). Then, there exists a violation $\tau' = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p, t) \cdot \beta'_1 \cdot \mathsf{isu}(p_1, t_1) \cdot \beta_2 \cdot (p', t') \cdot \mathsf{del}(p', t) \cdot \gamma'_{\mathsf{S}}$ obtained from τ where $\beta'_1 \cdot \gamma'_{\mathsf{S}}$ is a subsequence of $\beta_1 \cdot \gamma_{\mathsf{S}}$ and the sequence of issue events of delayed transactions forms a causality chain that starts with $\mathsf{isu}(p, t)$ and ends at $\mathsf{isu}(p_1, t_1)$.*

Proof. First, we show that we can obtain a violation τ' from τ where all delayed transactions in $\beta'_1 \cdot \mathsf{isu}(p_1, t_1)$ are causally dependent on $\mathsf{isu}(p, t)$. From the definition of t_1 in Lemma 3.6, we already have that $(\mathsf{isu}(p, t), \mathsf{isu}(p_1, t_1)) \in \mathsf{CO}$. In the proof, we assume w.l.o.g that in $\beta_1 \cdot \mathsf{isu}(p_1, t_1) \cdot \beta_2$ there is no event a that reads a value that t overwrites, otherwise, we can shortcut the trace by removing (p', t') and instead using the conflict relation between a and a store event of t to build the transactional happens-before cycle. Now, assume that β_1 contains a delayed transaction t_0 from another process $q \neq p$ that is not causally dependent on $\mathsf{isu}(p,t)$. We show that we either can obtain a contradiction or we can remove all events of t_0 and obtain a new violation trace τ' that has the same number of delays as τ. We have three possible cases based on whether the delayed store event $\mathsf{del}(q',t_0)$ of t_0 occurs in β_1, β_2 or γ_{S}. Hence, we get that τ can be one of the following:

(a) $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta_{11} \cdot \mathsf{isu}(q,t_0) \cdot \beta_{12} \cdot \mathsf{del}(q',t_0) \cdot \beta_{13} \cdot \mathsf{isu}(p_1,t_1) \cdot \beta_2 \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$

(b) $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta_{11} \cdot \mathsf{isu}(q,t_0) \cdot \beta_{12} \cdot \mathsf{isu}(p_1,t_1) \cdot \beta_{21} \cdot \mathsf{del}(q',t_0) \cdot \beta_{22} \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$

(c) $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta_{11} \cdot \mathsf{isu}(q,t_0) \cdot \beta_{12} \cdot \mathsf{isu}(p_1,t_1) \cdot \beta_2 \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma_{\mathsf{S}}^1 \cdot \mathsf{del}(q',t_0) \cdot \gamma_{\mathsf{S}}^2$

In case (a) (resp., (b)) we can notice that since $\mathsf{isu}(q,t_0)$ happens-before $\mathsf{del}(q',t_0)$ through β_{12} (resp., $\beta_{12} \cdot \mathsf{isu}(p_1,t_1) \cdot \beta_{21}$) then after executing $\mathsf{del}(q',t_0)$ we obtain a cycle in the transactional happens-before. Thus, we can remove (p',t') from both traces and still obtain a robustness violation. Let τ' be the resulting trace. τ' has the same number of delays as τ. In τ', we do not have events that read values that t overwrites. Therefore, we do not need to delay the transaction t to ensure that that the trace is a violation. Let τ'' be the resulting trace where the transaction t executes atomically. In τ'', the transaction t was not delayed after the issue event of t_1 which means that τ'' has less delays than τ. This contradicts the fact that τ is a minimal violation.

Case (c): we assume that $\mathsf{del}(q',t_0)$ happens-before after (p',t'), otherwise, we can reorder it before (p',t') and get case (b). Since γ_{S}^1 contains only store events, then by the happens-before definition, $\mathsf{del}(q',t_0)$ must be a store event executed by p' which means that $q' = p'$. Let e_1 and e_2 be the read/write actions t' that are the source of the conflict between (p',t') and $\mathsf{del}(p',t)$ and the happens-before between (p',t') and $\mathsf{del}(p',t_0)$, respectively. Similar to before we assume w.l.o.g that there is no event in $\beta_{12} \cdot \mathsf{isu}(p_1,t_1) \cdot \beta_2$ that reads a value that t_0 overwrites. We consider the two cases: i) e_2 occurs before e_1 in t' or the two coincide, and ii) e_1 occurs before e_2 in t'. In the first case we can obtain a new violation where we do not delay the transaction t which will not affect the action e_2 that is the source of the happens-before between (p',t') and $\mathsf{del}(p',t_0)$ (since e_1 occurs after e_2 then it cannot disable it). The new trace τ' is a violation since the store event $\mathsf{del}(p',t_0)$ is delayed. Also, since the store event $\mathsf{del}(p',t)$ of t was not delayed after $\mathsf{isu}(p_1,t_1)$ then τ' has less delays than τ, which contradicts the fact that τ is a minimal violation. In the second case, if in $\beta_{12} \cdot \mathsf{isu}(p_1,t_1) \cdot \beta_2$ we do not have any event that is causally dependent on $\mathsf{isu}(q,t_0)$ other than the store events of t_0, then we can remove all events of t_0 from τ without affecting the happens before between $\mathsf{isu}(p,t)$ and

$\mathsf{del}(p',t)$ through $\beta_{12}\cdot\mathsf{isu}(p_1,t_1)\cdot\beta_2\cdot(p',t')$. Let $\tau' = \alpha_{\mathsf{A}}\cdot\mathsf{isu}(p,t)\cdot\beta_1'\cdot\mathsf{isu}(p_1,t_1)\cdot\beta_2\cdot(p',t')\cdot\mathsf{del}(p',t)\cdot\gamma_{\mathsf{S}}'$ be the resulting trace which has the same number of delays as τ. Otherwise, if in $\beta_{12}\cdot\mathsf{isu}(p_1,t_1)\cdot\beta_2$ we have an event a that is causally dependent on $\mathsf{isu}(q,t_0)$ that is not a store event of t_0, then the new trace t' resulting from not delaying t_0 is a violation. This is because the store event $\mathsf{del}(p',t)$ is delayed. τ' has less delays than τ since the store event $\mathsf{del}(p',t_0)$ of t_0 was not delayed after a. This contradicts the fact that τ is a minimal violation.

Now, we show that for every two successive issue events of delayed transactions in τ', we can deliver the first to the process of the second before the second is issued. Let $ev_i = \mathsf{isu}(p_i,t_i)$ and $ev_j = \mathsf{isu}(p_j,t_j)$ be two successive issue events of delayed transactions in $\beta_1\cdot\mathsf{isu}(p_1,t_1)$ s.t. either $(ev_i,ev_j)\in\mathsf{HB}$ or $(ev_i,\mathsf{del}(p_i,t_j))\in\mathsf{HB}$. Note that the only case where the store event $\mathsf{del}(p_j,t_i)$ cannot be moved to occur before ev_j in β_1 is when the two events are related by a happens-before relation, i.e., $(ev_j,\mathsf{del}(p_j,t_i))\in\mathsf{HB}$. In this case, we get that the transactions t_i and t_j are involved in a cycle in the transactional happens-before in τ' which means that $\tau'' = \alpha_{\mathsf{A}}\cdot(p,t)\cdot\beta_1\cdot\mathsf{isu}(p_1,t_1)\cdot\gamma_{\mathsf{S}}$ is a violation which has less delays than τ (since t was not delayed after $\mathsf{isu}(p_1,t_1)$). Therefore, the trace τ'' where the store event $\mathsf{del}(p_j,t_i)$ occurs before ev_2 is happens-before equivalent to τ'. Similarly, when the two events are concurrent, the trace τ'' where the store event $\mathsf{del}(p_j,t_i)$ occurs before ev_j is happens-before equivalent to τ'. Thus, given the sequence of issue events $ev_1\cdot ev_2\cdot\ldots ev_n$ of delayed transactions in τ' s.t. $ev_1 = \mathsf{isu}(p,t)$ and $ev_n = \mathsf{isu}(p_1,t_1)$, the trace τ'' where for every $1\le k\le n-1$ s.t. $ev_k = \mathsf{isu}(p_k,t_k)$ and $ev_{k+1} = \mathsf{isu}(p_{k+1},t_{k+1})$, we have the store event $\mathsf{del}(p_{k+1},t_k)$ occurs before the issue event ev_{k+1} is happens-before equivalent to τ'. Also, in τ'' for every $2\le k\le n$, we have that ev_k is causally dependent on $ev_1 = \mathsf{isu}(p,t)$. Thus, in τ'' the sequence of issue events $ev_1\cdot ev_2\cdot\ldots ev_n$ of delayed transactions forms a causality chain. $\square$

Next, we show that for minimal violations in $\mathbb{T}\mathsf{ccv2}$ where t and t_1 are distinct transactions, all delayed transactions in $\mathsf{isu}(p,t)\cdot\beta_1$ do not access the shared variable x that starts the happens-before path in β_2 (Lemma 3.6) between $\mathsf{isu}(p_1,t_1)$ and (p',t'). If this were not the case, then the events of t_1 can be removed and we still guarantee a happens-before path to $\mathsf{del}(p',t)$ (starting in the delayed transaction accessing the variable x), thus obtaining a new robustness violation trace with less delays (since $\mathsf{del}(p',t)$ was not delayed after $\mathsf{isu}(p_1,t_1)$), which contradicts the minimality assumption.

Lemma 3.9. *Let τ be a minimal violation in $\mathbb{T}\mathsf{ccv2}$ where t_1 and t are two distinct transactions. Then, all the delayed transactions in $\mathsf{isu}(p,t)\cdot\beta_1$ do not access the variable x from Lemma 3.6.*

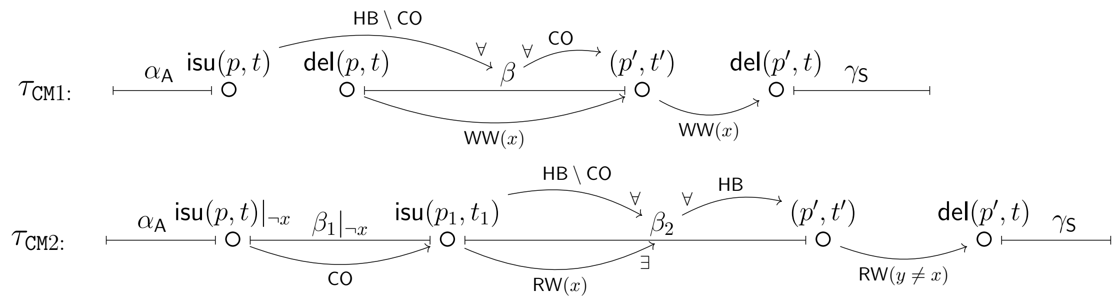


Figure 3.12: Robustness violation patterns under CM. For violation $\tau_{\mathtt{CM1}}$, t is the only delayed transaction. For $\tau_{\mathtt{CM2}}$, all delayed transactions are in $\mathsf{isu}(p,t) \cdot \beta_1 \cdot \mathsf{isu}(p_1,t_1)$ and they form a causality chain that starts at $\mathsf{isu}(p,t)$ and ends at $\mathsf{isu}(p_1,t_1)$.

Proof. Suppose by contradiction that we have an issue event $\mathsf{isu}(p_2,t_2)$ in $\mathsf{isu}(p,t) \cdot \beta_1$ (i.e., $\mathsf{isu}(p,t) \cdot \beta_1 = \mathsf{isu}(p,t) \cdot \beta_{11} \cdot \mathsf{isu}(p_2,t_2) \cdot \beta_{12}$) which accesses the shared variable x with either a read or a write instruction. Then, since there exists an event $a \in \beta_2$ s.t. $(\mathsf{isu}(p_1,t_1), a) \in \mathsf{WW}(x) \cup (\mathsf{STO}; \mathsf{RW}(x))$, we have that $(\mathsf{isu}(p_2,t_2), a) \in \mathsf{WW}(x) \cup (\mathsf{STO}; \mathsf{RW}(x))$. Moreover, because $\beta_2 \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$ does not contain any transaction that causally depends on $\mathsf{isu}(p_1,t_1)$, we get that $\mathsf{isu}(p_1,t_1)$ is the issue event by the process p_1 and we can remove it together with all the related stores in γ_{S} to obtain: $\tau' = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta_{11} \cdot \mathsf{isu}(p_2,t_2) \cdot \beta_{12} \cdot \beta_2 \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma'_{\mathsf{S}}$ which is a violation because $\mathsf{isu}(p,t)$ happens-before $\mathsf{del}(p',t)$ through $\beta_{11} \cdot \mathsf{isu}(p_2,t_2) \cdot \beta_{12} \cdot \beta_2 \cdot (p',t')$. Furthermore, τ' has less delays than τ since $\mathsf{del}(p',t)$ was not delayed after $\mathsf{isu}(p_1,t_1)$. This contradicts the fact that τ is a minimal violation. □

3.8 Robustness Violations Under Causal Memory

The characterization of robustness violations under CM is at some level similar to that of robustness violations under CCv. However, some instance of the violation pattern under CCv is not possible under CM and CM admits some class of violations that is not possible under CCv. This reflects the fact that these consistency models are incomparable in general.

The following theorem gives the characterization of minimal violations under CM which is pictured in Figure 3.12. Roughly, a program is not robust iff it admits a violation that either contains two concurrent transactions that write to the same variable, or it is a restriction of the pattern $\tau_{\mathtt{CCv2}}$ admitted by CCv where the last delayed transaction is related only by RW to future transactions.

The first pattern is not admitted by CCv because the writes to each variable are executed according to the timestamp order (CM does not satisfy the CCv property stated in Lemma 3.1).

Theorem 3.5. *A program $\mathcal{P}$ is not robust under* CM *iff there exists a minimal violation in $\mathbb{T}r(\mathcal{P})_{\mathtt{CM}}$ of one of the following forms:*

1. $\tau_{\mathtt{CM1}} = \alpha_{\mathsf{A}} \cdot \mathit{isu}(p,t) \cdot \mathit{del}(p,t) \cdot \beta \cdot (p',t') \cdot \mathit{del}(p',t) \cdot \gamma_{\mathsf{S}}$, *where:*

   (a) $\mathit{isu}(p,t)$ *is the issue of the first and only delayed transaction;*

   (b) $\exists\ y.\ s.t.\ (\mathit{del}(p,t),(p',t')) \in \mathsf{WW}(y)$ *and* $((p',t'),\mathit{del}(p',t)) \in \mathsf{WW}(y)$ *(Lemma 3.11);*

   (c) $\forall\ a \in \beta.\ (\mathit{isu}(p,t),a) \in \mathsf{HB} \setminus \mathsf{CO}$ *and* $(a,(p',t')) \in \mathsf{CO}$ *(Lemma 3.11).*

2. $\tau_{\mathtt{CM2}} = \alpha_{\mathsf{A}} \cdot \mathit{isu}(p,t) \cdot \beta_1 \cdot \mathit{isu}(p_1,t_1) \cdot \beta_2 \cdot (p',t') \cdot \mathit{del}(p',t) \cdot \gamma_{\mathsf{S}}$, *where*

   (a) $\mathit{isu}(p,t)$ *and* $\mathit{isu}(p_1,t_1)$ *are the issues of the first and last delayed transactions (Lemma 3.11);*

   (b) *the issues of all delayed transactions are in β_1 are included in a causality chain that starts with $\mathit{isu}(p,t)$ and ends with $\mathit{isu}(p_1,t_1)$;*

   (c) *for every $a \in \beta_2$, we have that* $(\mathit{isu}(p_1,t_1),a) \in \mathsf{HB} \setminus \mathsf{CO}$ *and* $(a,(p',t')) \in \mathsf{HB}$ *(Lemma 3.11);*

   (d) *there exist $a \in \beta_2 \cdot (p',t')$, x, and y s.t. $x \neq y$,* $(\mathit{isu}(p_1,t_1),a) \in \mathsf{RW}(x)$, $(a,(p',t')) \in \mathsf{HB}?$, *and* $((p',t'),\mathit{del}(p',t)) \in \mathsf{RW}(y)$ *(Lemma 3.11);*

   (e) *all delayed transactions in $\mathit{isu}(p,t) \cdot \beta_1$ do not access the variable x.*

The violation pattern $\tau_{\mathtt{CM2}}$ is a restriction of the pattern $\tau_{\mathtt{CCv2}}$ under CCv. For instance, the trace in Figure 3.11b is a valid minimal violation of the SB program under CM. The violation pattern $\tau_{\mathtt{CM1}}$ implies the existence of a write-write data race under CM. Figure 3.13 shows a minimal violation under CM that corresponds to a write-write data race in the LU program. Conversely, if a program $\mathcal{P}$ admits a trace τ which contains a write-write data race under CM, then $\mathcal{P}$ also admits a trace τ' where the two transactions t_1 and t_2 that caused the write-write data race form a cycle in the store order (the store events of t_1 and t_2 on the two processes p_1 and p_2 that issued them can be reordered to occur in opposite orders, i.e., $\mathsf{del}(p_1,t_1)$ before $\mathsf{del}(p_1,t_2)$ and $\mathsf{del}(p_2,t_2)$ before $\mathsf{del}(p_2,t_1)$, which

$$\begin{tikzcd}
\mathsf{isu}(p1,t1) & \mathsf{del}(p1,t1) \arrow[r, bend left, "\mathsf{WW}"] & (p2,t2) \arrow[r, bend left, "\mathsf{WW}"] & \mathsf{del}(p2,t1)
\end{tikzcd}$$

Figure 3.13: Violation of LU program in Figure 3.2a. A $\tau_{\mathtt{CM1}}$ violation where $\beta_2 = \gamma_{\mathsf{S}} = \epsilon$, and t and t' correspond to $t1$ and $t2$.

implies that are also in opposite orders w.r.t. the store order). Thus, $\mathcal{P}$ has a trace τ' with a cycle in the transactional happens-before which means that $\mathcal{P}$ is not robust against `CM`. Therefore, a program which is robust against `CM` is also write-write data race free under `CM`. Since without write-write data races, the `CM` and the `CCv` semantics coincide, we get the following the result.

Lemma 3.10. *If a program $\mathcal{P}$ is robust against* `CM`*, then $\mathcal{P}$ is robust against* `CCv`*.*

Next, we discuss the proof of Theorem 3.5. The following lemma reveals the two possible minimal violation patterns under causal memory. The characterization of the patterns in this lemma can be refined further using arguments similar to the case of `CCv` (see the discussion at the end of this section).

Lemma 3.11. *If $\mathcal{P}$ is a program that is not robust under* `CM`*, then it must admit a minimal violation τ that satisfies one of the following:*

1. $\tau = \alpha_{\mathsf{A}} \cdot \mathit{isu}(p,t) \cdot \mathit{del}(p,t) \cdot \beta \cdot (p',t') \cdot \mathit{del}(p',t) \cdot \gamma_{\mathsf{S}}$ *where:*
   (a) $\exists\ y.\ s.t.\ (\mathit{del}(p,t),(p',t')) \in \mathsf{WW}(y)$ *and* $((p',t'),\mathit{del}(p',t)) \in \mathsf{WW}(y)$*;*
   (b) $\forall\ a \in \beta.\ (\mathit{isu}(p,t),a) \in \mathsf{HB} \setminus \mathsf{CO}$ *and* $(a,(p',t')) \in \mathsf{CO}$*.*
2. $\tau = \alpha_{\mathsf{A}} \cdot \mathit{isu}(p,t) \cdot \beta_1 \cdot \mathit{isu}(p_1,t_1) \cdot \beta_2 \cdot (p',t') \cdot \mathit{del}(p',t) \cdot \gamma_{\mathsf{S}}$ *where:*
   (a) $\mathit{isu}(p_1,t_1)$ *is the last issue event from* $\{c \in \beta \mid (\mathit{isu}(p,t),c) \in \mathsf{CO}\}$ *in* τ*;*
   (b) *there exist two variables* $x \neq y$*,* a *in* $\beta_2 \cdot (p',t')$*, and* $b = (p',t')$ *such that* $(\mathit{isu}(p_1,t_1),a) \in \mathsf{RW}(x)$*,* $(b,\mathit{del}(p',t)) \in \mathsf{RW}(y)$*, and* $(a,b) \in \mathsf{HB}?$*;*
   (c) $\forall\ a \in \beta_2.\ (\mathit{isu}(p_1,t_1),a) \in \mathsf{HB} \setminus \mathsf{CO}$ *and* $(a,(p',t')) \in \mathsf{HB}$*.*

Proof. The proof will contain many arguments which are similar to those used in the proof of Lemma 3.6. Let $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$ be a minimal violation under `CM` (cf. Lemma 3.5). We prove that there exists a minimal violation trace τ' obtained from τ that satisfies (1) or (2). Similar to Lemma 3.6, we get that there must exist y s.t. $((p',t'),\mathsf{del}(p',t)) \in \mathsf{RW}(y) \cup \mathsf{WW}(y)$.

We consider two cases: i) $((p',t'),\mathsf{del}(p',t)) \in \mathsf{WW}(y)$, and ii) $((p',t'),\mathsf{del}(p',t)) \in \mathsf{RW}(y)$. If $((p',t'),\mathsf{del}(p',t)) \in \mathsf{WW}(y)$, then by reordering the store event $\mathsf{del}(p,t) \in \gamma_{\mathsf{S}}$ to occur just after the corresponding issue and removing all events in β (and all related stores in γ_{S}) that are not causally ordered before (p',t') (since they do not contribute to the transactional happens-before cycle) we obtain a trace $\tau' = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \mathsf{del}(p,t) \cdot \beta' \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma'_{\mathsf{S}}$ that is also a minimal violation and where $(\mathsf{del}(p,t),(p',t')) \in \mathsf{WW}(y)$ and $((p',t'),\mathsf{del}(p',t)) \in \mathsf{WW}(y)$. The trace τ' satisfies the first case of the lemma.

Now assume that $((p',t'),\mathsf{del}(p',t)) \in \mathsf{RW}(y)$, and let $\sigma = \{c \in \beta \mid (\mathsf{isu}(p,t),c) \in \mathsf{CO}\}$. We consider the following three cases.

First, assume that σ is empty. As in the proof of Lemma 3.6, we obtain that there exist $a \in \beta \cdot (p',t')$ and x s.t. $(\mathsf{isu}(p,t),a) \in \mathsf{RW}(x)$ and $(a,(p',t')) \in \mathsf{HB}?$. If $x = y$ then both t and the transaction t_2 by a process p_2 of the event a write to x. Similar to before we can reorder the store event $\mathsf{del}(p,t) \in \gamma_{\mathsf{S}}$ to occur just after the corresponding issue and remove all issue events in $\beta \cdot (p',t')$ that occur after the issue event of t_2 and all their related stores. Also, we remove all events in β that are not causally ordered before the issue event of t_2. We obtain $\tau' = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \mathsf{del}(p,t) \cdot \beta'(p_2,t_2) \cdot \mathsf{del}(p_2,t) \cdot \gamma'_{\mathsf{S}}$. In τ' the events of t_2 are assembled together, $\mathsf{del}(p_2,t) \in \gamma_{\mathsf{S}}$ is reordered to occur just after (p_2,t_2), and $(\mathsf{del}(p,t),(p_2,t_2)) \in \mathsf{WW}(y)$ and $((p_2,t_2),\mathsf{del}(p_2,t)) \in \mathsf{WW}(y)$. Thus, τ' is a minimal violation and it satisfies the first case of the lemma. If $x \neq y$ then we get the second case of the lemma.

Second, assume that σ is not empty and all the elements of σ are store events. As in the proof of Lemma 3.6, we obtain that there exist x and an event $a \in \beta \cdot (p',t')$ that is not a store event of t s.t. $(\mathsf{isu}(p,t),a) \in (\mathsf{STO};\mathsf{WW}(x)) \cup \mathsf{RW}(x)$ and $(a,(p',t')) \in \mathsf{HB}?$. If $(\mathsf{isu}(p,t),a) \in (\mathsf{STO};\mathsf{WW}(x))$ or $x = y$ then both t and the transaction t_2 by a process p_2 of the event a write to x. Using the same procedure as in the previous paragraph we can obtain $\tau' = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \mathsf{del}(p,t) \cdot \beta' \cdot (p_2,t_2) \cdot \mathsf{del}(p_2,t) \cdot \gamma'_{\mathsf{S}}$ that satisfies the first case of the lemma. Similarly, if $(\mathsf{isu}(p,t),a) \in \mathsf{RW}(x)$ and $x \neq y$ then we get the second case of the lemma.

Third, assume that σ is not empty and $\mathsf{isu}(p_1,t_1)$ is the last issue event in σ, i.e., $\beta = \beta_1 \cdot \mathsf{isu}(p_1,t_1) \cdot \beta_2 \cdot (p',t')$. As in the proof of Lemma 3.6, we obtain that there exist x and an event $a \in \beta_2 \cdot (p',t')$ that is not a store event of t_1 s.t. $(\mathsf{isu}(p_1,t_1),a) \in (\mathsf{STO};\mathsf{WW}(x)) \cup \mathsf{RW}(x)$ and $(a,(p',t')) \in \mathsf{HB}?$. If $x = y$ then both t and the transaction t_2 by a process p_2 of the event a write to x. Using the same procedure as before we can obtain a trace $\tau' = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \mathsf{del}(p,t) \cdot \beta'_1 \cdot \beta'_2 \cdot (p_2,t_2) \cdot \mathsf{del}(p_2,t) \cdot \gamma'_{\mathsf{S}}$ that is a minimal violation. τ' has less delays than τ since the store of t was not delayed after $\mathsf{isu}(p_1,t_1)$. This contradicts the fact that τ is a minimal violation. Assume now that $x \neq y$. We assume w.l.o.g. that all events in β_2 do not read values that any transaction with an issue event in $\mathsf{isu}(p,t) \cdot beta_1 \cdot \mathsf{isu}(p_1,t_1)$ overwrites. If $(\mathsf{isu}(p_1,t_1),a) \in (\mathsf{STO};\mathsf{WW}(x))$ and $a \neq (p',t')$ then we can remove all issue events in $\beta_2 \cdot (p',t')$ that occur after the issue event of t_2 including

(p', t') and assemble together the events of t_2. We obtain that $(\mathsf{del}(p_1, t_1), (p_2, t_2)) \in \mathsf{WW}(x)$ and $((p_2, t_2), \mathsf{del}(p_2, t_1)) \in \mathsf{WW}(x)$ where we do not need to delay the transaction t and obtain $\tau' = \alpha_{\mathsf{A}} \cdot (p, t) \cdot \beta'_1 \cdot \mathsf{isu}(p_1, t_1) \cdot \mathsf{del}(p_1, t_1) \cdot \beta'_2 \cdot (p_2, t_2) \cdot \mathsf{del}(p_2, t_1) \cdot \gamma'_{\mathsf{S}}$ that is a violation and has less delays than τ. This contradicts the fact that τ is a minimal violation. If $(\mathsf{isu}(p_1, t_1), a) \in (\mathsf{STO}; \mathsf{WW}(x))$ and $a = (p', t')$ (i.e., $t' = t_2$) then we construct τ' such that all transactions that have issue events in σ and t are executed atomically after all the events in $(\beta_1 \setminus \sigma) \cdot \beta_2 \cdot \mathsf{isu}(p', t') \cdot \mathsf{del}(p', t')$ are executed first, i.e., $\tau' = \alpha_{\mathsf{A}} \cdot \beta_{11} \cdot \beta_2 \cdot \mathsf{isu}(p', t') \cdot \mathsf{del}(p', t') \cdot (p, t) \cdot \beta_{12} \cdot \beta' \cdot (p_1, t_1) \cdot \mathsf{del}(p_1, t') \cdot \gamma'_{\mathsf{S}}$. τ' is a robustness violation since $(\mathsf{del}(p', t'), (p_1, t_1)) \in \mathsf{WW}(x)$ and $((p_1, t_1), \mathsf{del}(p_1, t')) \in \mathsf{WW}(x)$. Also, τ' has less delays than τ since t' was not delayed after a causally dependent event other than its store events and t is no longer delayed after the issue event of t_1. This contradicts the fact that τ is a minimal violation. Finally, the only remaining possibility is $(\mathsf{isu}(p_1, t_1), a) \in \mathsf{RW}(x)$ where $x \neq y$ which corresponds to the second case of the lemma.

□

We use $\mathbb{T}\mathsf{cm1}$ and $\mathbb{T}\mathsf{cm2}$ to denote the class of minimal violations that satisfy the first and second case in Lemma 3.11, respectively. To show that for a non robust program, we can always find a minimal violation in either $\mathbb{T}\mathsf{cm1}$ or $\mathbb{T}\mathsf{cm2}$ where β and β_2 do not contain delayed transactions we can use the same proof arguments as in Lemma 3.7. For minimal violations in $\mathbb{T}\mathsf{cm2}$ where t and t_1 are distinct transactions, the two properties that issue events of all delayed transactions form a causality chain and that delayed transactions in $\mathsf{isu}(p, t) \cdot \beta_1$ do not access the shared variable x can also be proved in the same manner as in Lemmas 3.8 and 3.9, respectively.

3.9 Robustness Violations Under Weak Causal Consistency

If a program is robust against `CM`, then it must not contain a write-write race under `CM` (note that this is not true for `CCv`). Therefore, by Theorem 3.2, a program which is robust against `CM` has the same set of traces under both `CM` and `wCC`, which implies that it is also robust against `wCC`. Conversely, since `wCC` is weaker than `CM` (i.e., $\mathbb{T}\mathsf{r}_{\mathtt{CM}}(\mathcal{P}) \subseteq \mathbb{T}\mathsf{r}_{\mathtt{wCC}}(\mathcal{P})$ for any $\mathcal{P}$), if a program is robust against `wCC` then it is robust against `CM`. Thus, we obtain the following result.

Theorem 3.6. *A program $\mathcal{P}$ is robust against* `wCC` *iff it is robust against* `CM`*.*

3.10 Reduction to SC Reachability

We describe a reduction of robustness checking to a reachability problem in a program executing under the serializability semantics, which can be simulated on top of standard sequential consistency (SC) by considering that each transaction is an atomic section (guarded by a fixed global lock). Essentially, given a program $\mathcal{P}$ and a semantics $\mathsf{X} \in \{\texttt{CCv}, \texttt{CM}, \texttt{wCC}\}$, we define an instrumentation of $\mathcal{P}$ such that $\mathcal{P}$ is not robust against X iff the instrumentation reaches an error state under the serializability semantics. The instrumentation uses auxiliary variables in order to simulate the robustness violations (in particular, the delayed transactions) satisfying the patterns given in Figure 3.10 and Figure 3.12. We will focus our presentation on the second violation pattern of CCv (which is similar to the second violation pattern of CM): $\tau_{\texttt{CCv2}} = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta_1 \cdot \mathsf{isu}(p_1,t_1) \cdot \beta_2 \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$.

The process p that delayed the first transaction t is called the *Attacker*. The other processes delaying transactions in $\beta_1 \cdot \mathsf{isu}(p_1,t_1)$ are called *Visibility Helpers*. Recall that all the delayed transactions must be causally ordered after $\mathsf{isu}(p,t)$. The processes that execute transactions in $\beta_2 \cdot (p',t')$ and contribute to the happens-before path between $\mathsf{isu}(p_1,t_1)$ and $\mathsf{del}(p',t)$ are called *Happens-Before Helpers*. A happens-before helper cannot be the attacker or a visibility helper since this would contradict the causal delivery guarantee provided by causal consistency (a transaction of a happens-before helper is not delayed, so visible immediately to all processes, and it cannot follow a delayed transaction). γ_{S} contains the stores of the delayed transactions in $\mathsf{isu}(p,t) \cdot \beta_1 \cdot \mathsf{isu}(p_1,t_1)$. It is important to notice that we may have $t = t_1$. In this case, $\beta_1 = \epsilon$ and the only delayed transaction is t. Also, all delayed transactions in β_1 including t_1 may be issued by the same process as t. In all of these cases, the set of Visibility Helpers is empty.

The instrumentation uses two copies of the set of shared variables in the original program. We use primed variables x' to denote the second copy. When a process becomes the attacker or a visibility helper, it will write only to the second copy that is visible only to these processes (and remains invisible to the other processes including the happens-before helpers). The writes made by other processes including the happens-before helpers are made visible to all processes, i.e., they are applied on both copies of every shared variable.

To establish the causality chains of the delayed transactions issued by the attacker and the visibility helpers, we look whether a transaction can extend the causality chain started by the first delayed transaction issued by the attacker. This is to ensure that all such transactions are causally related to the first delayed transaction (of the attacker). In order for a transaction to "join" the causality chain, it has to satisfy one of the following conditions:

- the transaction is issued by a process that has already another transaction in the causality chain. Thus, we ensure the continuity of the causality chain through program order;
- the transaction is reading from a variable that was updated by a previous transaction in the causality chain. Hence, we ensure the continuity of the causality chain through the write-read relation.

We introduce a flag for each shared variable to mark the fact that it was updated by a previous transaction in the causality chain. These flags are used by the instrumentation to establish whether a transaction "joins" a causality chain. Enforcing a happens-before path starting in the last delayed transaction, using transactions of the happens-before helpers, can be done in the same way. Compared to causality chains, there are two more cases in which a transaction can extend a happens-before path:

- the transaction writes to a shared variable that was read by a previous transaction in the happens-before path. Hence, we ensure the continuity of the happens-before path through the read-write relation;
- the transaction writes to a shared variable that was updated by a previous transaction in the happens-before path. Hence, we ensure the continuity of the happens-before path through write-write order.

Thus, we extend the shared variables flags used for causality chains in order to record if a variable was read or written by a previous transaction (in this case, a previous transaction in the happens-before path). Overall, the instrumentation uses a flag $x.event$ or $x'.event$ for each (copy of a) shared variable, that stores the type of the last access (read or write) to the variable. Initially, these flags and other flags used by the instrumentation as explained below are initialized to null ($\perp$).

In general, whether a process is an attacker, visibility helper, or happens-before helper is not enforced syntactically by the instrumentation, and can vary from execution to execution. The role of a process in an execution is set *non-deterministically* during the execution using some additional process-local flags. Thus, during an execution, each process chooses to set to `true` at most one of the flags $p.a$, $p.vh$, and $p.hbh$, implying that the process becomes an attacker, visibility helper, or happens-before helper, respectively. At most one process can be an attacker, i.e., set $p.a$ to `true`.

```
$[\![ \mathsf{l}_1 : \mathsf{begin};\ \mathsf{goto}\ \mathsf{l}_2; ]\!]_{\mathsf{A}} =$
// Typical execution of begin
  $\mathsf{l}_1$: assume $\mathsf{HB} = \perp \wedge (p.a \neq \perp \vee a_{\mathsf{tr_A}} = \perp)$; goto $\mathsf{l}_{x1}$;   (3.1)
  $\mathsf{l}_{x1}$: begin; goto $\mathsf{l}_2$;
// Begin of first delayed transaction
  $\mathsf{l}_1$: assume $\mathsf{HB} = \perp \wedge a_{\mathsf{tr_A}} = \perp$ ; goto $\mathsf{l}_{x2}$;   (3.2)
  $\mathsf{l}_{x2}$: begin; goto $\mathsf{l}_{x3}$;
  $\mathsf{l}_{x3}$: $p.a := 1$; goto $\mathsf{l}_{x4}$;
  $\mathsf{l}_{x4}$: Foreach $x \in \mathbb{V}.\ x' := x$; goto $\mathsf{l}_{x5}$;
  $\mathsf{l}_{x5}$: $a_{\mathsf{tr_A}} := \mathsf{true}$; goto $\mathsf{l}_2$;   (3.3)

$[\![ \mathsf{l}_1 : r := x;\ \mathsf{goto}\ \mathsf{l}_2; ]\!]_{\mathsf{A}} =$
// Read before delaying transactions
  $\mathsf{l}_1$: assume $a_{\mathsf{tr_A}} = \perp$ ; goto $\mathsf{l}_{x1}$;
  $\mathsf{l}_{x1}$: $r := x$; goto $\mathsf{l}_2$;
// Read in delayed transactions
  $\mathsf{l}_1$: assume $a_{\mathsf{tr_A}} \neq \perp \wedge p.a \neq \perp$ ; goto $\mathsf{l}_{x2}$;
  $\mathsf{l}_{x2}$: $r := x'$; goto $\mathsf{l}_{x3}$;
  $\mathsf{l}_{x3}$: $x'.event := \mathsf{ld}$; goto $\mathsf{l}_2$;
// Special read in last delayed transaction
  $\mathsf{l}_1$: assume $x'.event = \perp \wedge p.a \neq \perp$ ; goto $\mathsf{l}_{x4}$;
  $\mathsf{l}_{x4}$: $r := x'$; goto $\mathsf{l}_{x5}$;
  $\mathsf{l}_{x5}$: $\mathsf{HB} := \mathsf{true}$; goto $\mathsf{l}_{x6}$;   (3.4)
  $\mathsf{l}_{x6}$: $x.event := \mathsf{ld}$; goto $\mathsf{l}_2$;   (3.5)
```

```
$[\![ \mathsf{l}_1 : x := e;\ \mathsf{goto}\ \mathsf{l}_2; ]\!]_{\mathsf{A}} =$
// Write before delaying transactions
  $\mathsf{l}_1$: assume $a_{\mathsf{tr_A}} = \perp$ ; goto $\mathsf{l}_{x1}$;
  $\mathsf{l}_{x1}$: $x := e$; goto $\mathsf{l}_2$;
// Write in delayed transactions
  $\mathsf{l}_1$: assume $a_{\mathsf{tr_A}} \neq \perp \wedge p.a \neq \perp$ ; goto $\mathsf{l}_{x2}$;
  $\mathsf{l}_{x2}$: $x' := e$; goto $\mathsf{l}_{x3}$;   (3.6)
  $\mathsf{l}_{x3}$: $x'.event := \mathsf{st}$; goto $\mathsf{l}_2$;   (3.7)
// Special write in first delayed transaction
  $\mathsf{l}_1$: assume $a_{\mathsf{st_A}} = x.event = \perp \wedge p.a \neq \perp$ ; goto $\mathsf{l}_{x4}$;
  $\mathsf{l}_{x4}$: $x' := e$; goto $\mathsf{l}_{x5}$;
  $\mathsf{l}_{x5}$: $a_{\mathsf{st_A}} := `x`$; goto $\mathsf{l}_{x6}$;   (3.8)
  $\mathsf{l}_{x6}$: $x'.event := \mathsf{st}$; goto $\mathsf{l}_2$;
// Special write in last delayed transaction
  $\mathsf{l}_1$: assume $x'.event = \perp \wedge p.a \neq \perp$ ; goto $\mathsf{l}_{x7}$;
  $\mathsf{l}_{x7}$: $x' := e$; goto $\mathsf{l}_{x8}$;
  $\mathsf{l}_{x8}$: $\mathsf{HB} := \mathsf{true}$; goto $\mathsf{l}_{x9}$;   (3.9)
  $\mathsf{l}_{x9}$: $x.event := \mathsf{ld}$; goto $\mathsf{l}_2$;   (3.10)

$[\![ \mathsf{l}_1 : \mathsf{com};\ \mathsf{goto}\ \mathsf{l}_2; ]\!]_{\mathsf{A}} =$
  $\mathsf{l}_1$: assume $p.a \neq \perp \wedge a_{\mathsf{st_A}} = \perp$ ; assume false;
  $\mathsf{l}_1$: com; goto $\mathsf{l}_2$;
```

Figure 3.14: Instrumentation of the Attacker. We use 'x' to denote the name of the variable x.

3.10.1 Instrumentation of the Attacker

We provide in Figure 3.14, the instrumentation of the instructions for the attacker process. Such a process passes through an initial phase where it executes transactions that are visible immediately to all the other processes (i.e., they are not delayed), and then non-deterministically it can choose to delay a transaction. When the attacker randomly chooses the first transaction to start delaying of transactions, it sets a *global* flag $a_{\mathsf{tr_A}}$ to true in the instruction begin (line (3.3)). Then, it sets the flag $p.a$ to 1 to indicate that the current process is the attacker. During the first delayed transaction, the attacker non-deterministically chooses a write instruction to a shared variable y and stores the name of this variable in the flag $a_{\mathsf{st}_\mathsf{A}}$ (line (4.5)). The values written during delayed transactions are stored in the primed variables and are visible only to the attacker and the visibility helpers. For example, given a variable z, all the writes to z from the original program are transformed into writes to the primed version z' (line (4.3)). Each time the attacker writes to a variable z', it sets the flag $z'.event$ to st (line (4.4)) which will allow other processes that read the same variable to join the set of visibility helpers and start delaying their transactions. Once the attacker delays a transaction, it will read only from the primed variables (i.e., z').

To start the happens-before path, the attacker has to execute a transaction that either reads or writes to a shared variable x that was not accessed by a delayed transaction (i.e., $x'.event = \perp$). In this case, it sets the variable HB to true (lines (4.2) and (3.9)) to mark the start of the happens before path and the end of the visibility chains, and it sets the flag $x.event$ to ld (lines (4.1) and (3.10)). We set $x.event$ to ld even in the case of a write to x in order to simplify the instrumentation of the happens-before helpers (to check that this transaction is related to a transaction of a happens-before helper p through $\mathsf{WW}(x)$ or $\mathsf{RW}(x)$ it is enough that p writes to x and it "observers" the same value ld in $x.event$). When the flag HB is set to true the attacker stops executing new transactions. We can notice that when the HB is set to true, we can no longer execute new transactions from the attacker (all conditions in lines (3.1) and (3.2) become false).

3.10.2 Instrumentation of the Visibility Helpers

Figure 3.15 lists the instrumentation of the instructions of a process that belongs to the set of visibility helpers. Such a process passes through an initial phase where it executes the original code instructions (lines (3.18) and (3.13)) until the flag $a_{\mathsf{tr}_\mathsf{A}}$ is set to true by the attacker. Then, it continues the execution of its original instructions but, whenever it stores a value it writes it to both the shared variable z and the primed variable z' so it is visible to all processes. Non deterministically it chooses a first transaction to delay, at which point it joins the set of visibility helpers. It sets the flag $p.vh$ to false signaling its desire to join the visibility helpers, and it chooses a transaction (the begin of this transaction is shown in line (3.12)) through which the process will join the set of visibility helpers. The process directly starts delaying its writes, i.e., writing to primed variables, and reading only from delayed writes, i.e., from primed variables, and behaving the same as the attacker. In order to check that it can extend the sequence of causal dependencies (required by the causal chain definition), it takes a snapshot of the $\_.event$ fields at the beginning of the transaction

$[\![\mathsf{l}_1\colon \mathsf{begin};\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{VH}} =$

// Before joining visibility helpers

$\mathsf{l}_1\colon \mathtt{assume}\ \mathsf{HB} = \perp \wedge (a_{\mathsf{tr_A}} = \perp \vee p.vh = \perp);\ \mathtt{goto}\ \mathsf{l}_{x1};$

$$\mathsf{l}_{x1}\colon \mathsf{begin};\ \mathtt{goto}\ \mathsf{l}_2; \tag{3.11}$$

// Joining visibility helpers

$\mathsf{l}_1\colon \mathtt{assume}\ \mathsf{HB} = p.vh = p.a = \perp \wedge a_{\mathsf{tr_A}} \neq \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x2};$

$\mathsf{l}_{x2}\colon \mathsf{begin};\ \mathtt{goto}\ \mathsf{l}_{x3};$

$\mathsf{l}_{x3}\colon p.vh := \mathsf{false};\ \mathtt{goto}\ \mathsf{l}_{x4};$

$\mathsf{l}_{x4}\colon \mathtt{Foreach}\ x' \in \mathbb{V}.\ x'.event' := x'.event;\ \mathtt{goto}\ \mathsf{l}_2;$

// After joining visibility helpers

$\mathsf{l}_1\colon \mathtt{assume}\ \mathsf{HB} = \perp \wedge a_{\mathsf{tr_A}} \neq \perp \wedge p.vh;\ \mathtt{goto}\ \mathsf{l}_{x5};$

$$\mathsf{l}_{x5}\colon \mathsf{begin};\ \mathtt{goto}\ \mathsf{l}_2; \tag{3.12}$$

$[\![\mathsf{l}_1\colon r := x;\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{VH}} =$

// Before joining visibility helpers

$\mathsf{l}_1\colon \mathtt{assume}\ a_{\mathsf{tr_A}} = \perp \vee (p.vh = p.a = \perp);\ \mathtt{goto}\ \mathsf{l}_{x1};$

$$\mathsf{l}_{x1}\colon r := x;\ \mathtt{goto}\ \mathsf{l}_2; \tag{3.13}$$

// After joining visibility helpers

$\mathsf{l}_1\colon \mathtt{assume}\ p.vh \neq \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x2};$

$$\mathsf{l}_{x2}\colon r := x';\ \mathtt{goto}\ \mathsf{l}_{x3}; \tag{3.14}$$

$\mathsf{l}_{x3}\colon \mathtt{assume}\ x'.event' = \mathsf{st} \wedge\ \neg p.vh;\ \mathtt{goto}\ \mathsf{l}_{x4};$

$$\mathsf{l}_{x4}\colon p.vh := \mathsf{true};\ \mathtt{goto}\ \mathsf{l}_2; \tag{3.15}$$

$\mathsf{l}_{x3}\colon \mathtt{assume}\ x'.event' \neq \mathsf{st} \vee\ p.vh;\ \mathtt{goto}\ \mathsf{l}_2;$

// Last delayed transaction

$\mathsf{l}_1\colon \mathtt{assume}\ x'.event = \perp \wedge p.vh \neq \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x5};$

$$\mathsf{l}_{x5}\colon \mathsf{HB} := \mathsf{true};\ \mathtt{goto}\ \mathsf{l}_{x6}; \tag{3.16}$$

$$\mathsf{l}_{x6}\colon x.event := \mathsf{ld};\ \mathtt{goto}\ \mathsf{l}_{x7}; \tag{3.17}$$

$\mathsf{l}_{x7}\colon r := x';\ \mathtt{goto}\ \mathsf{l}_2;$

$[\![\mathsf{l}_1\colon x := e;\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{VH}} =$

// Before attacker delays transactions

$\mathsf{l}_1\colon \mathtt{assume}\ a_{\mathsf{tr_A}} = \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x1};$

$$\mathsf{l}_{x1}\colon x := e;\ \mathtt{goto}\ \mathsf{l}_2; \tag{3.18}$$

// Before joining visibility helpers

$\mathsf{l}_1\colon \mathtt{assume}\ a_{\mathsf{tr_A}} \neq \perp \wedge p.vh = p.a = \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x2};$

$\mathsf{l}_{x2}\colon x' := e;\ \mathtt{goto}\ \mathsf{l}_{x3};$

$\mathsf{l}_{x3}\colon x := e;\ \mathtt{goto}\ \mathsf{l}_2;$

// After joining visibility helpers

$\mathsf{l}_1\colon \mathtt{assume}\ p.vh \neq \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x4};$

$$\mathsf{l}_{x4}\colon x' := e;\ \mathtt{goto}\ \mathsf{l}_{x5}; \tag{3.19}$$

$$\mathsf{l}_{x5}\colon x'.event := \mathsf{st};\ \mathtt{goto}\ \mathsf{l}_{x6}; \tag{3.20}$$

$\mathsf{l}_{x6}\colon x'.event' := \perp\ ;\ \mathtt{goto}\ \mathsf{l}_2;$

// Last delayed transaction

$\mathsf{l}_1\colon \mathtt{assume}\ x'.event = \perp \wedge p.vh \neq \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x7};$

$$\mathsf{l}_{x7}\colon \mathsf{HB} := \mathsf{true};\ \mathtt{goto}\ \mathsf{l}_{x8}; \tag{3.21}$$

$$\mathsf{l}_{x8}\colon x.event := \mathsf{ld};\ \mathtt{goto}\ \mathsf{l}_{x9}; \tag{3.22}$$

$\mathsf{l}_{x9}\colon x' := e;\ \mathtt{goto}\ \mathsf{l}_2;$

$[\![\mathsf{l}_1\colon \mathsf{com};\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{VH}} =$

// Before joining visibility helpers

$\mathsf{l}_1\colon \mathtt{assume}\ a_{\mathsf{tr_A}} = \perp \vee (a_{\mathsf{tr_A}} \neq \perp \wedge p.vh = \perp);\ \mathtt{goto}\ \mathsf{l}_{x1};$

$\mathsf{l}_{x1}\colon \mathsf{com};\ \mathtt{goto}\ \mathsf{l}_2;$

// After joining visibility helpers

$\mathsf{l}_1\colon \mathtt{assume}\ a_{\mathsf{tr_A}} \neq \perp \wedge p.vh;\ \mathtt{goto}\ \mathsf{l}_{x2};$

$\mathsf{l}_{x2}\colon \mathsf{com};\ \mathtt{goto}\ \mathsf{l}_2;$

// Failed to join visibility helpers

$$\mathsf{l}_1\colon \mathtt{assume}\ a_{\mathsf{tr_A}} \neq \perp \wedge \neg p.vh;\ \mathtt{assume}\ \mathsf{false}; \tag{3.23}$$

Figure 3.15: Instrumentation of the Visibility Helpers.

and stores it to $\_.event'$ fields (line l_{x4} in the instrumentation of begin). This snapshot is necessary to check that it reads from writes made in other transactions (ignoring the writes in the current transaction). When a process choses a first transaction to delay (during the `begin` instruction), it has made a pledge that during this transaction it will read from a variable that was updated by a another delayed transaction from either the attacker or some other visibility helper. This is to ensure that this transaction extends the visibility chain. Hence, the local process flag $p.vh$ will be set to true when the process meets its pledge (line (3.15)). If the process does not keep its pledge (i.e., $p.vh$ is equal to false) at the end of the transaction (i.e., during the end instruction) we block the execution. Thus, when executing the com instruction of the underlying transaction we check whether the flag $p.vh$ is null, if so we block the execution (line (3.23)).

When a process joins the visibility helpers, it delays all writes and reads only from the primed variables (lines (3.19) and (3.14)). Similar to the attacker, a process in the visibility helpers delays a write to a shared variable z by writing to z', it sets the flag $z'.event$ to st (line (3.20)). In order for a process in the visibility helpers to start the happens-before path, it has to either read or write a shared variable x that was not accessed by a delayed transaction (i.e., $x'.event = \perp$). In this case we set the flag HB to true (lines (3.21) and (3.16)) to mark the start of the happens before path and the end of the visibility chains and set the flag $x.event$ to ld (lines (3.22) and (3.17)). When the flag HB is set to true, all processes in the set of visibility helpers stop issuing new transactions because all conditions for executing the begin instruction become false.

3.10.3 Instrumentation of the Happens-Before Helpers

The remaining processes, which are not the attacker or a visibility helper, can become happens-before helpers. Figure 3.16 lists the instrumentation of the instructions of a happens-before helper process. Similar to above, when the flag $a_{\mathsf{tr_A}}$ is set to true by the attacker, other processes enter a phase where they continue executing their instructions, however, when they store a value they write it in both the shared variable z and the primed variable z' (lines (3.25) and (3.26)). However, they only read from the original shared variables (line (4.6)). Once the flag HB is set to true, a process that cannot be the attacker (i.e., the flag $p.a$ is null) or a visibility helper (i.e., the flag $p.vh$ is null) chooses non-deterministically a transaction t (the begin of this transaction is shown in line (3.24)) through which it wants to join the set of happens-before helpers, i.e., continue the happens-before path created by the existing happens-before helpers. Similar to visibility helpers, when a process choses the transaction t, it makes a pledge (while executing the begin instruction) that during this transaction it will either read a variable updated by another happens-before helper or write to a variable that was accessed (read or written) by another happens-before helper (every process that executes a transaction after HB is set to true makes this pledge). When the pledge is met, the process sets the flag $p.hbh$ to true (lines (4.7) and (4.11)). The execution is blocked if a process does

$[\![\mathsf{l}_1 : \mathtt{begin};\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{HbH}} =$

// Before joining happens-before helpers

$\mathsf{l}_1 : \mathtt{assume}\ \mathsf{HB} = p.vh = p.a = \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x1};$

$\mathsf{l}_{x1} : \mathtt{begin};\ \mathtt{goto}\ \mathsf{l}_2;$

// Joining happens-before helpers

$\mathsf{l}_1 : \mathtt{assume}\ \mathsf{HB} \neq \perp \wedge p.hbh = p.vh = p.a = \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x2};$

$\mathsf{l}_{x2} : \mathtt{begin};\ \mathtt{goto}\ \mathsf{l}_{x3};$ (3.24)

$\mathsf{l}_{x3} : \mathtt{Foreach}\ x \in \mathbb{V}.\ x.event' := x.event;\ \mathtt{goto}\ \mathsf{l}_2;$

// After joining happens-before helpers

$\mathsf{l}_1 : \mathtt{assume}\ \mathsf{HB} \neq \perp \wedge p.hbh \neq \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x4};$

$\mathsf{l}_{x4} : \mathtt{begin};\ \mathtt{goto}\ \mathsf{l}_2;$

$[\![\mathsf{l}_1 : x := e;\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{HbH}} =$

// Before the first delayed transaction

$\mathsf{l}_1 : \mathtt{assume}\ \mathsf{HB} = \perp \wedge a_{\mathsf{tr_A}} = \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x1};$

$\mathsf{l}_{x1} : x := e;\ \mathtt{goto}\ \mathsf{l}_2;$

// After the first delayed transaction

$\mathsf{l}_1 : \mathtt{assume}\ \mathsf{HB} = p.vh = p.a = \perp \wedge a_{\mathsf{tr_A}} \neq \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x2};$

$\mathsf{l}_{x2} : x' := e;\ \mathtt{goto}\ \mathsf{l}_{x3};$ (3.25)

$\mathsf{l}_{x3} : x := e;\ \mathtt{goto}\ \mathsf{l}_2;$ (3.26)

// After the last delayed transaction

$\mathsf{l}_1 : \mathtt{assume}\ \mathsf{HB} \neq \perp \wedge p.vh = p.a = \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x4};$

$\mathsf{l}_{x4} : x := e;\ \mathtt{goto}\ \mathsf{l}_{x5};$

$\mathsf{l}_{x5} : x.event := \mathsf{st};\ \mathtt{goto}\ \mathsf{l}_{x6};$ (3.27)

$\mathsf{l}_{x6} : \mathtt{assume}\ x.event' \neq \perp \wedge p.hbh = \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x7};$

$\mathsf{l}_{x7} : p.hbh := \mathsf{true};\ \mathtt{goto}\ \mathsf{l}_2;$ (3.28)

$\mathsf{l}_{x6} : \mathtt{assume}\ x.event' = \perp \vee p.hbh \neq \perp\ ;\ \mathtt{goto}\ \mathsf{l}_2;$

$[\![\mathsf{l}_1 : r := x;\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{HbH}} =$

// Before the last delayed transaction

$\mathsf{l}_1 : \mathtt{assume}\ \mathsf{HB} = \perp \wedge p.vh = p.a = \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x1};$

$\mathsf{l}_{x1} : r := x;\ \mathtt{goto}\ \mathsf{l}_2;$ (3.29)

// After the last delayed transaction

$\mathsf{l}_1 : \mathtt{assume}\ \mathsf{HB} \neq \perp \wedge p.vh = p.a = \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x2};$

$\mathsf{l}_{x2} : r := x;\ \mathtt{goto}\ \mathsf{l}_{x3};$

$\mathsf{l}_{x3} : \mathtt{assume}\ x.event' = \mathsf{st} \wedge\ p.hbh = \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x4};$

$\mathsf{l}_{x4} : p.hbh := \mathsf{true};\ \mathtt{goto}\ \mathsf{l}_2;$ (3.30)

$\mathsf{l}_{x3} : \mathtt{assume}\ x.event = \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x5};$

$\mathsf{l}_{x5} : x.event := \mathsf{ld};\ \mathtt{goto}\ \mathsf{l}_2;$ (3.31)

$\mathsf{l}_{x3} : \mathtt{assume}\ x.event \neq \perp \vee p.hbh \neq \perp\ ;\ \mathtt{goto}\ \mathsf{l}_2;$

$[\![\mathsf{l}_1 : \mathtt{com};\ \mathtt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{HbH}} =$

// Before joining happens-before helpers

$\mathsf{l}_1 : \mathtt{assume}\ \mathsf{HB} = p.vh = p.a = \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x1};$

$\mathsf{l}_{x1} : \mathtt{com};\ \mathtt{goto}\ \mathsf{l}_2;$

// After joining happens-before helpers

$\mathsf{l}_1 : \mathtt{assume}\ \mathsf{HB} \neq \perp \wedge p.hbh \neq \perp\ ;\ \mathtt{goto}\ \mathsf{l}_{x2};$

$\mathsf{l}_{x2} : \mathtt{com};\ \mathtt{goto}\ \mathsf{l}_{x3};$

$\mathsf{l}_{x3} : \tilde{r} := a_{\mathsf{st_A}};\ \mathtt{goto}\ \mathsf{l}_{x4};$ (3.32)

$\mathsf{l}_{x4} : \tilde{r} := \tilde{r}.event;\ \mathtt{goto}\ \mathsf{l}_{x5};$ (3.33)

$\mathsf{l}_{x5} : \mathtt{assume}\ \tilde{r} \neq \perp\ ;\ \mathtt{assert\ false};$ (3.34)

$\mathsf{l}_{x5} : \mathtt{assume}\ \tilde{r} = \perp\ ;\ \mathtt{goto}\ \mathsf{l}_2;$

// Failed to join happens-before helpers

$\mathsf{l}_1 : \mathtt{assume}\ \mathsf{HB} \neq \perp \wedge p.hbh = p.vh = p.a = \perp$

$;\ \mathtt{assume\ false};$ (3.35)

Figure 3.16: Instrumentation of Happens-Before Helpers.

not keep its pledge (i.e., the flag *p.hbh* is null) at the end of the transaction (line (3.35)). We use a flag *x.event* for each variable x to record the type (read ld or write st) of the last access made by a happens-before helper (lines (4.8) and (4.10)). Moreover, once HB is set to true (i.e., there are no more delayed transactions), the process can write and read only the original shared variables, since the primed versions are no longer in use. A particular case is when the transaction t is from the first process trying to join the happens-before helpers, in which the transaction must contain a read accessing the variable x that was read or written to by a transaction from the attacker of a visibility helper.

The happens-before helpers continue executing their instructions, until one of them reads from the shared variable y whose name was stored in $a_{\mathsf{st_A}}$. This establishes a happens-before path between the last delayed transaction and a "fictitious" store event corresponding to the first delayed transaction that could be executed just after this read of y. The execution does not have to contain this store event explicitly since it is always enabled. Therefore, at the end of every transaction, the instrumentation checks whether the transaction read y. If it is the case, then the execution stops and goes to an error state to indicate that this is a robustness violation. The happens-before helpers processes continue executing their instructions, until one of them executes a load that reads from the shared variable y that was stored in $a_{\mathsf{st_A}}$ which implies the existence of a happens-before cycle. Thus, when executing the instruction com at the end of every transaction, we have a conditional check to detect if we have a load or a write accessing the variable y (lines (3.32), (3.33), and (3.34)). When the check detects that the variable y was accessed, the execution goes to an error state (line (3.34)) to indicate that it has produced a robustness violation.

In Figure 3.17, we show an excerpt of the instrumentations of the two transactions of the SB program. In particular, we only give the instructions of the instrumented SB that are reached during the execution that leads to an error state. The attacker instrumentation is applied to the transaction $t1$ of $p1$ and the happens-before helpers instrumentation is applied to the transaction $t2$ of $p2$. The first conflict order from $t1$ to $t2$ (shown in Figure 3.10) is simulated by the fact that at line 3.39, $y.event' = \mathsf{ld}$ (see lines 3.37 and 3.38). Also, the second conflict order from $t2$ to $t1$ is simulated by the fact that at line 3.41 we reach the error state where $a_{\mathsf{st_A}}.event = x.event = \mathsf{ld}$ (see lines 3.36 and 3.40).

3.10.4 Correctness

As we have already mentioned, the role of a process in an execution is chosen non-deterministically at runtime. Therefore, the final instrumentation of a given program $\mathcal{P}$, denoted by $[\![\mathcal{P}]\!]^{\mathsf{P2}}$, is obtained by replacing each labeled instruction $\langle linst \rangle$ with the concatenation of the instrumentations corresponding to the attacker, the visibility helpers, and the happens-before helpers, i.e., $[\![\langle linst \rangle]\!]^{\mathsf{P2}} ::= [\![\langle linst \rangle]\!]_{\mathsf{A}}\ [\![\langle linst \rangle]\!]_{\mathsf{VH}}\ [\![\langle linst \rangle]\!]_{\mathsf{HbH}}$. The instrumented program $[\![\mathcal{P}]\!]^{\mathsf{P2}}$ reaches the error state iff $\mathcal{P}$ admits a violation of the pattern τ_{CCv2}. Let $[\![\mathcal{P}]\!]^{\mathsf{P1}}$ be the instrumented program

$$
\begin{aligned}
&[\![\mathsf{l}_1 : \texttt{begin};\ \texttt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{A}} = \\
&\quad \mathsf{l}_1 : \texttt{assume}\ \mathsf{HB} = \perp \wedge\ a_{\mathsf{tr}_\mathsf{A}} = \perp\ ;\ \texttt{goto}\ \mathsf{l}_{b2}; \\
&\quad \mathsf{l}_{b2} : \texttt{begin};\ \texttt{goto}\ \mathsf{l}_{b3}; \\
&\quad \mathsf{l}_{b3} : p1.a := 1;\ \texttt{goto}\ \mathsf{l}_{b4}; \\
&\quad \mathsf{l}_{b4} : \texttt{Foreach}\ z \in \mathbb{V}.\ z' := z;\ \texttt{goto}\ \mathsf{l}_{b5}; \\
&\quad \mathsf{l}_{b5} : a_{\mathsf{tr}_\mathsf{A}} := \mathsf{true};\ \texttt{goto}\ \mathsf{l}_2;
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathsf{l}_2 : x := 1;\ \texttt{goto}\ \mathsf{l}_3;]\!]_{\mathsf{A}} = \\
&\quad \mathsf{l}_2 : \texttt{assume}\ a_{\mathsf{st}_\mathsf{A}} = x.event = \perp \wedge\ p1.a \neq \perp\ ;\ \texttt{goto}\ \mathsf{l}_{s4}; \\
&\quad \mathsf{l}_{s4} : x' := 1;\ \texttt{goto}\ \mathsf{l}_{s5}; \\
&\quad \mathsf{l}_{s5} : a_{\mathsf{st}_\mathsf{A}} := \text{`}x\text{`};\ \texttt{goto}\ \mathsf{l}_{s6}; \\
&\quad \mathsf{l}_{s6} : x'.event := \mathsf{st};\ \texttt{goto}\ \mathsf{l}_3;
\end{aligned} \tag{3.36}
$$

$$
\begin{aligned}
&[\![\mathsf{l}_3 : r1 := y;\ \texttt{goto}\ \mathsf{l}_4;]\!]_{\mathsf{A}} = \\
&\quad \mathsf{l}_3 : \texttt{assume}\ y'.event = \perp \wedge\ a_{\mathsf{tr}_\mathsf{A}} \neq \perp\ ;\ \texttt{goto}\ \mathsf{l}_{l4}; \\
&\quad \mathsf{l}_{l4} : r1 := y';\ \texttt{goto}\ \mathsf{l}_{l5}; \\
&\quad \mathsf{l}_{l5} : \mathsf{HB} := \mathsf{true};\ \texttt{goto}\ \mathsf{l}_{l6}; \\
&\quad \mathsf{l}_{l6} : y.event := \mathsf{ld};\ \texttt{goto}\ \mathsf{l}_4;
\end{aligned} \tag{3.37}
$$

$$
\begin{aligned}
&[\![\mathsf{l}_4 : \texttt{com};\ \texttt{goto}\ \mathsf{l}_5;]\!]_{\mathsf{A}} = \\
&\quad \mathsf{l}_4 : \texttt{com};\ \texttt{goto}\ \mathsf{l}_5;
\end{aligned}
$$

$$
\begin{aligned}
&[\![\mathsf{l}_1 : \texttt{begin};\ \texttt{goto}\ \mathsf{l}_2;]\!]_{\mathsf{HbH}} = \\
&\quad \mathsf{l}_1 : \texttt{assume}\ \mathsf{HB} \neq \perp \wedge\ p2.hbh = p2.vh = p2.a = \perp\ ;\ \texttt{goto}\ \mathsf{l}_{b2}; \\
&\quad \mathsf{l}_{b2} : \texttt{begin};\ \texttt{goto}\ \mathsf{l}_{b3}; \\
&\quad \mathsf{l}_{b3} : x.event' := x.event;\ y.event' := y.event;\ \texttt{goto}\ \mathsf{l}_2;
\end{aligned} \tag{3.38}
$$

$$
\begin{aligned}
&[\![\mathsf{l}_3 : y := 1;\ \texttt{goto}\ \mathsf{l}_4;]\!]_{\mathsf{HbH}} = \\
&\quad \mathsf{l}_3 : \texttt{assume}\ \mathsf{HB} \neq \perp \wedge\ p2.vh = p2.a = \perp\ ;\ \texttt{goto}\ \mathsf{l}_{s4}; \\
&\quad \mathsf{l}_{s4} : y := 1;\ \texttt{goto}\ \mathsf{l}_{s5}; \\
&\quad \mathsf{l}_{s5} : y.event := \mathsf{st};\ \texttt{goto}\ \mathsf{l}_{s6}; \\
&\quad \mathsf{l}_{s6} : \texttt{assume}\ y.event' \neq \perp \wedge\ p2.hbh = \perp\ ;\ \texttt{goto}\ \mathsf{l}_{s7}; \\
&\quad \mathsf{l}_{s7} : p2.hbh := \mathsf{true};\ \texttt{goto}\ \mathsf{l}_4;
\end{aligned} \tag{3.39}
$$

$$
\begin{aligned}
&[\![\mathsf{l}_2 : r2 := x;\ \texttt{goto}\ \mathsf{l}_3;]\!]_{\mathsf{HbH}} = \\
&\quad \mathsf{l}_2 : \texttt{assume}\ \mathsf{HB} \neq \perp \wedge\ p.vh = p.a = \perp\ ;\ \texttt{goto}\ \mathsf{l}_{l2}; \\
&\quad \mathsf{l}_{l2} : r2 := x;\ \texttt{goto}\ \mathsf{l}_{l3}; \\
&\quad \mathsf{l}_{l3} : \texttt{assume}\ x.event = \perp\ ;\ \texttt{goto}\ \mathsf{l}_{l5}; \\
&\quad \mathsf{l}_{l5} : x.event := \mathsf{ld};\ \texttt{goto}\ \mathsf{l}_3;
\end{aligned} \tag{3.40}
$$

$$
\begin{aligned}
&[\![\mathsf{l}_4 : \texttt{com};\ \texttt{goto}\ \mathsf{l}_5;]\!]_{\mathsf{HbH}} = \\
&\quad \mathsf{l}_4 : \texttt{assume}\ \mathsf{HB} \neq \perp \wedge\ p2.hbh \neq \perp\ ;\ \texttt{goto}\ \mathsf{l}_{e2}; \\
&\quad \mathsf{l}_{e2} : \texttt{com};\ \texttt{goto}\ \mathsf{l}_{e3}; \\
&\quad \mathsf{l}_{e3} : \tilde{r} := a_{\mathsf{st}_\mathsf{A}};\ \texttt{goto}\ \mathsf{l}_{e4}; \\
&\quad \mathsf{l}_{e4} : \tilde{r} := \tilde{r}.event;\ \texttt{goto}\ \mathsf{l}_{e5}; \\
&\quad \mathsf{l}_{e5} : \texttt{assume}\ \tilde{r} \neq \perp\ ;\ \texttt{assert false};
\end{aligned} \tag{3.41}
$$

Figure 3.17: Instrumentation of SB program in Figure 3.2b.

that reaches an error state iff $\mathcal{P}$ admits a violation of the pattern $\tau_{\texttt{CCv1}}$. The instrumentation $[\![\ ]\!]^{\mathsf{P1}}$ does not include the visibility helpers since only a single transaction is delayed in $\tau_{\texttt{CCv1}}$, and it can be obtained in the same manner as $[\![\ ]\!]^{\mathsf{P2}}$. The following theorem states the correctness of the instrumentation.

Theorem 3.7. *A program $\mathcal{P}$ is not robust against* CCv *iff either $[\![\mathcal{P}]\!]^{\mathsf{P1}}$ or $[\![\mathcal{P}]\!]^{\mathsf{P2}}$ reaches the error state.*

The proof of this theorem relies on the explanations given above. One can define a bijection between executions of the instrumentation that reach an error state and executions of the original program that satisfy the constraints in one of the two violation patterns. The former can be rewritten to the latter by roughly, removing all accesses to the auxiliary variables used by the instrumentation, replacing the writes to shared variable copies by writes to the original variables, delivering delayed transactions only to visibility helpers, and appending store events for all the delayed transactions. For the reverse, given a robustness violation $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta_1 \cdot \mathsf{isu}(p_1,t_1) \cdot \beta_2 \cdot (p',t') \cdot \mathsf{del}(p',t) \cdot \gamma_{\mathsf{S}}$ of type τ_{CCv2}, we can build an execution of the instrumentation that reaches an error state, where p is the attacker, the processes delaying transactions in $\beta_1 \cdot \mathsf{isu}(p_1,t_1)$ are visibility helpers, and the processes that issue transactions between $\mathsf{isu}(p_1,t_1)$ and $\mathsf{del}(p',t)$ and that are part of the happens-before path between these two events are the happens-before helpers.

The following result states the complexity of checking robustness for finite-state programs[9] against one of the three variations of causal consistency considered in this work (we use causal consistency as a generic name to refer to all of them). The upper bound is a direct consequence of Theorem 3.7 and of previous results concerning the reachability problem in concurrent programs running over SC, with a fixed [108] or parametric number of processes [146]. For the lower bound, given an instance of the reachability problem under sequential consistency, denoted by $(\mathcal{P},\ell)$[10], we construct a program $\mathcal{P}'$ where each statement s of $\mathcal{P}$ is a different transaction (guarded by a global lock), and where reaching the location ℓ enables the execution of a "gadget" that corresponds to the SB program in Figure 3.2b. Executing each statement under a global lock ensures that every execution of $\mathcal{P}'$ under causal consistency is serializable, and faithfully represents an execution of the original $\mathcal{P}$ under sequential consistency. Moreover, $\mathcal{P}$ reaches ℓ iff $\mathcal{P}'$ contains a robustness violation, which is due to the execution of SB.

Corollary 3.1. *Checking robustness of finite-state programs against causal consistency is PSPACE-complete when the number of processes is fixed and EXPSPACE-complete, otherwise.*

Remark 3.3. *The reduction to reachability does not manipulate transaction identifiers and it is insensitive to the number of transactions executed by one process. Thus, all our results extend to processes that include unbounded loops of transactions. This includes programs where each process can call a statically known set of transactions (with parameters) an arbitrary number of times.*

[9]That is, programs where the number of variables and the data domain are bounded.

[10]That is, whether the program $\mathcal{P}$ reaches the control location ℓ under SC.

3.11 Related Work

Causal consistency is one of the oldest consistency models for distributed systems [116]. Formal definitions of several variants of causal consistency, suitable for different types of applications, have been introduced recently [60, 59, 142, 52]. The definitions in this chapter are inspired from these works and coincide with those given in [52]. In that paper, the authors address the decidability and the complexity of verifying that an implementation of a storage system is causally consistent (i.e., all its computations, for every client, are causally consistent).

While our work focuses on *trace-based* robustness, *state-based robustness* requires that a program is robust if the set of all its reachable states under the weak semantics is the same as its set of reachable states under the strong semantics. While state-robustness is the necessary and sufficient concept for preserving state-invariants, its verification, which amounts in computing the set of reachable states under the weak semantics, is in general a hard problem. The decidability and the complexity of this problem has been investigated in the context of relaxed memory models such as TSO and Power, and it has been shown that it is either decidable but highly complex (non-primitive recursive), or undecidable [37, 38]. Recently, [111] showed that it is also non-primitive recursive for causal convergence. As far as we know, the decidability and complexity of this problem has not been investigated for weak causal consistency and causal memory.

Automatic procedures for approximate reachability/invariant checking have been proposed using either abstractions or bounded analyses, e.g., [39, 28, 72, 24]. Proof methods have also been developed for verifying invariants and portability in the context of weakly consistent models such as [114, 93, 130, 27, 118]. These methods, however, do not provide decision procedures.

Decidability and complexity of trace-based robustness has been investigated for the Release-Aquire (RA) and Partitioned Global Address Space (PGAS) parallel programming models, and the TSO and Power weak memory models [113, 66, 55, 51, 74, 73]. The work we present in this chapter borrows the idea of using minimal violation characterizations for building an instrumentation allowing to obtain a reduction of the robustness checking problem to the reachability checking problem over SC. However, applying this approach to the case of causal consistency is not straightforward and requires different proof techniques. Dealing with causal consistency requires coming up with radically different arguments and proofs, for (1) characterizing in a finite manner the set of violations, (2) showing that this characterization is sound and complete, and (3) using effectively this characterization in the definition of the reduction to the reachability problem.

The robustness reductions defined in [55, 66, 74] are based on the theory of regular languages and do not extend to infinite-state programs like in our case. [55] uses an approach that consists of enumeration of a pair of SC computations that are conflicting, i.e., between reads and writes actions occurring in the pair of SC computations, to simulate a TSO computation that violates SC. [66, 74] define decision procedures based on checking the intersection of a multi-headed automaton introduced in [66] that simulates minimal violations types of computations (called normal form computations) with regular languages. In [75], the authors develop robustness characterizations for store-atomic consistency models called locality and singularity, which state that in a minimal violation only a single process delays writes and only a single write is delayed, respectively. [115] proposes robustness characterizations for weak memory models in terms of program transformations allowed over SC. In particular, they show that instructions reorderings and eliminations are enough to characterize TSO, while they are not enough to characterize neither C11, Power, or ARM.

As far as we know, our work is the first one that establishes results on the decidability and complexity issues of the robustness problem in the context of causal consistency, and taking into account transactions. The existing work on the verification of robustness for distributed systems consider essentially trace-based concepts of robustness and provide either over- or under-approximate analyses for checking it. In [47, 57, 58, 70], static analysis techniques are proposed based on computing an abstraction of the set of computations that is used in searching for robustness violations. These approaches may return false alarms due to the abstractions they consider. In particular, [47] shows that a trace under causal convergence is not admitted by the serializability semantics iff it contains a (transactional) happens-before cycle with a RW dependency, and another RW or WW dependency. This characterization alone is not sufficient to prove our result concerning robustness checking. Our result relies on a characterization of more refined robustness violations and relies on different proof arguments. In [129] a sound (but not complete) bounded analysis for detecting robustness violation is proposed. Our approach is technically different, is precise, and provides a decision procedure for checking robustness when the program is finite-state.

3.12 Conclusion

We studied three variations of causal consistency, showing that they are equivalent for programs without write-write data races. We showed that the problem of verifying that a program is robust against causal consistency relative to serializability can be reduced, modulo a linear-size instrumentation, to a reachability problem in a program running over serializability semantics. This reduction leads to the first decidability result concerning the problem of checking robustness against a weak transactional consistency model. Furthermore, this reduction opens the door to the use of existing methods and tools for the analysis and verification of sequentially consistent concurrent programs, in order to reason about weakly-consistent programs. It can be used for the design of a large spectrum of static/dynamic tools for testing/verifying robustness against causal consistency relative to serializability.

Chapter 4

Robustness Against Snapshot Isolation

4.1 Introduction

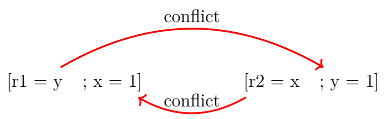
In this chapter, we address the problem of verifying robustness of programs against snapshot isolation (SER) relative to serializability. In §4.2, we outline our approach for tackling this problem. In §4.3, we formally define the semantics of programs under snapshot isolation consistency model. We also define programs executions and traces under this semantics. In §4.5, we present a series of results that characterize the particular shapes of minimal violations under SI. Then, in §4.6, we show a polynomial-time reduction of robustness to reachability problem in a program running under serializability semantics. Using the above reduction, in §4.7, we develop a proof methodology for establishing robustness which builds on Lipton's reduction theory [120]. In particular, we use the theory of movers to establish whether the relaxations allowed by SI are harmless, i.e., they do not introduce new behaviors compared to serializability. Finally, in §4.8, we apply our techniques on 10 challenging applications extracted from previous work [58, 161, 32, 105, 47, 93, 129]. We show that our techniques were enough for proving or disproving the robustness of these applications.

4.2 Overview

In this section, we give an overview of our approach for checking robustness against snapshot isolation. While serializability enforces that transactions are atomic and conflicting transactions,

```
    p1:                  p2:
t1: [r1 = y //0    || t2: [r2 = x //0
     x = 1]                y = 1]
```

(a) Write Skew (WS).



(b) A WS execution trace.

Figure 4.1: Examples of non-robust programs illustrating the difference between `SI` and serializability. *causal dependency* means that a read in a transaction obtains its value from a write in another transaction. *conflict* means that a write in a transaction is not visible to a read in another transaction, but it would affect the read value if it were visible. Here, *happens-before* is the union of the two.

i.e., which read or write to a common location, *cannot* commit concurrently, `SI` [46] allows that conflicting transactions commit in parallel as long as they do not contain a write-write conflict, i.e., write on a common location. Moreover, under `SI`, each transaction reads from a snapshot of the database taken at its start. These relaxations permit the "anomaly" known as Write Skew (WS) shown in Figure 4.1a, where an anomaly is a program execution which is allowed by `SI`, but not by serializability. The execution of Write Skew under `SI` allows the reads of `x` and `y` to return 0 although this cannot happen under serializability. These values are possible since each transaction is executed locally (starting from the initial snapshot) without observing the writes of the other transaction.

Execution trace. Our notion of program robustness is based on an abstract representation of executions called *trace*. Informally, an execution trace is a sequence of events, i.e., accesses to shared variables and transaction begin/commit events, along with several standard dependency relations between events recording the data-flow. The transitive closure of the union of all these dependency relations is called *happens-before*. An execution is an anomaly if the happens-before of its trace is cyclic. Figure 4.1b shows the happens-before of the Write Skew anomaly. Notice that the happens-before order is cyclic in both cases.

Semantically, every transaction execution involves two main events, the issue and the commit. The issue event corresponds to a sequence of reads and/or writes where the writes are visible only to the current transaction. We interpret it as a single event since a transaction starts with a database snapshot that it updates in isolation, without observing other concurrently executing transactions. The commit event is where the writes are propagated and made visible to all processes. Under serializability, the two events coincide, i.e., they are adjacent in the execution. Under `SI`, this is not the case and in between the issue and the commit of the same transaction, we may have issue/commit events from concurrent transactions. When a transaction commit does not occur immediately after its issue, we say that the underlying transaction is *delayed*. For example, the following execution of WS corresponds to the happens-before cycle in Figure 4.1b where the write to x was committed after t_2 finished, hence, t_1 was delayed:

$$\begin{array}{lll} \mathsf{begin}(p_1,t_1)\mathsf{ld}(p_1,t_1,y,0)\mathsf{isu}(p_1,t_1,x,1) & & \mathsf{com}(p_1,t_1) \\ & \mathsf{begin}(p_2,t_2)\mathsf{ld}(p_2,t_2,x,0)\mathsf{isu}(p_2,t_2,y,1)\mathsf{com}(p_2,t_2) & \end{array}$$

Above, $\mathsf{begin}(p_1, t_1)$ stands for starting a new transaction t_1 by process p_1, ld represents read (load) actions, while isu denotes write actions that are visible only to the current transaction (not yet committed). The writes performed during t_1 become visible to all processes once the commit event $\mathsf{com}(p_1, t_1)$ takes place.

Reducing robustness to SC reachability. The above `SI` execution can be mimicked by an execution of the same program under serializability modulo an instrumentation that simulates the delayed transaction. The local writes in the issue event are simulated by writes to auxiliary registers and the commit event is replaced by copying the values from the auxiliary registers to the shared variables (actually, it is not necessary to simulate the commit event; we include it here for presentation reasons). The auxiliary registers are visible only to the delayed transaction. In order that the execution be an anomaly (i.e., not possible under serializability without the instrumentation) it is required that the issue and the commit events of the delayed transaction are linked by a chain of happens-before dependencies. For instance, the above execution for WS can be simulated by:

$$\begin{array}{lll} \mathsf{begin}(p_1,t_1)\mathsf{ld}(p_1,t_1,y,0)\mathsf{we}(p_1,t_1,r_x,1) & & \mathsf{we}(p_1,t_1,x,r_x) \\ & \mathsf{begin}(p_2,t_2)\mathsf{ld}(p_2,t_2,x,0)\mathsf{isu}(p_2,t_2,y,1)\mathsf{com}(p_2,t_2) & \end{array}$$

The write to x was delayed by storing the value in the auxiliary register r_x and the happens-before chain exists because the read on y that was done by t_1 is conflicting with the write on y from t_2 and the read on x by t_2 is conflicting with the write of x in the simulation of t_1's commit event. On the other hand, the following execution of Write-Skew without the read on y in t_1:

$\mathsf{begin}(p_1,t_1)\mathsf{we}(p_1,t_1,r_x,1)$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\mathsf{we}(p_1,t_1,x,r_x)$

$\qquad\qquad\qquad\qquad$ $\mathsf{begin}(p_2,t_2)\mathsf{ld}(p_2,t_2,x,0)\mathsf{isu}(p_2,t_2,y,1)\mathsf{com}(p_2,t_2)$

misses the conflict (happens-before dependency) between the issue event of t_1 and t_2. Therefore, the events of t_2 can be reordered to the left of t_1 and obtain an equivalent execution where $\mathsf{we}(p_1,t_1,x,r_x)$ occurs immediately after $\mathsf{we}(p_1,t_1,r_x,1)$. In this case, t_1 is not anymore delayed and this execution is possible under serializability (without the instrumentation).

If the number of transactions to be delayed in order to expose an anomaly is unbounded, the instrumentation described above may need an unbounded number of auxiliary registers. This would make the verification problem hard or even undecidable. However, we show that it is actually enough to delay a single transaction, i.e., a program admits an anomaly under `SI` iff it admits an anomaly containing a single delayed transaction. This result implies that the number of auxiliary registers needed by the instrumentation is bounded by the number of program variables, and that checking robustness against `SI` can be reduced in linear time to a reachability problem under serializability (the reachability problem encodes the existence of the chain of happens-before dependencies mentioned above). The proof of this reduction relies on a non-trivial characterization of anomalies.

Proving robustness using commutativity dependency graphs. Based on the reduction above, we also devise an approximated method for checking robustness based on the concept of mover in Lipton's reduction theory [120]. An event is a left (resp., right) mover if it commutes to the left (resp., right) of another event (from a different process) while preserving the computation. We use the notion of mover to characterize happens-before dependencies between transactions. Roughly, there exists a happens-before dependency between two transactions in some execution if one does not commute to the left/right of the other one. We define a commutativity dependency graph which summarizes the happens-before dependencies in all executions of a given program between transactions t as they appear in the program, transactions $t \setminus \{w\}$ where the writes of t are deactivated (i.e., their effects are not visible outside the transaction), and transactions $t \setminus \{r\}$ where the reads of t obtain non-deterministic values. The transactions $t \setminus \{w\}$ are used to simulate issue events of delayed transactions (where writes are not yet visible) while the transactions $t \setminus \{r\}$ are used to simulate commit events of delayed transactions (which only write to the

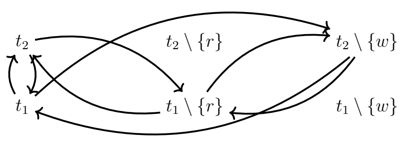


Figure 4.2: Commutativity dependency graph of WS where the read of y is omitted.

shared memory). Two transactions a and b are linked by an edge iff a *cannot* move to the right of b (or b cannot move to the left of a), or if they are related by the program order (i.e., issued in some order in the same process). Then a program is robust if for every transaction t, this graph *does not* contain a path from $t \setminus \{w\}$ to $t \setminus \{r\}$ formed of transactions that do not write to a variable that t writes to (the latter condition is enforced by SI since two concurrent transactions cannot commit at the same time when they write to a common variable). For example, Figure 4.2 shows the commutativity dependency graph of the modified WS program where the read of y is removed from t_1. The fact that it does not contain any path like above implies that it is robust.

4.3 Program Semantics Under Snapshot Isolation

The semantics of a program under SI is defined as follows. The shared variables are stored in a central memory and each process keeps a replicated copy of the central memory. A process starts a transaction by discarding its local copy and fetching the values of the shared variables from the central memory. When a process commits a transaction, it merges its local copy of the shared variables with the one stored in the central memory in order to make its updates visible to all processes. During the execution of a transaction, the process stores the writes to shared variables only in its local copy and reads only from its local copy. When a process merges its local copy with the centralized one, it is required that there were no concurrent updates that occurred after the last fetch from the central memory to a shared variable that was updated by the current transaction. Otherwise, the transaction is aborted and its effects discarded.

A program configuration is a tuple $\mathsf{gs} = (\mathsf{ls}, \mathsf{tstamp}, \mathsf{Log})$ where $\mathsf{ls} : \mathbb{P} \to \mathbb{S}$ associates a local state in $\mathbb{S}$ to each process in $\mathbb{P}$, $\mathsf{tstamp} : \mathbb{V} \to \mathbb{T}$ stores the largest timestamp for each shared variable, and $\mathsf{Log} : \mathbb{V} \to \mathbb{D}$ holds the global valuation of shared variables. A local state is a tuple $\langle \mathsf{pc}, \mathsf{store}, \mathsf{log}, \mathsf{rval} \rangle$ where $\mathsf{pc} \in \mathbb{L}\mathsf{ab}$ is the program counter, i.e., the label of the next instruction to be executed, $\mathsf{store} : \mathbb{V} \to \mathbb{D}$ is the local valuation of the shared variables, $\mathsf{log} : \mathbb{V} \to \{\bot, 1\}$ is a local log which marks shared variables which were updated in a transaction, and $\mathsf{rval} : \mathbb{R} \to \mathbb{D}$ is the valuation of the local registers. For a local state s, we use $s.\mathsf{pc}$ to denote the program counter component of s, and similarly for all the other components of s. Given a transaction $t \in \mathbb{T} \times \mathbb{T}$, we use $t.st$ to denote the start time of transaction t and $t.ct$ to denote the commit time of t. Before merging the updates in store with Log, after locally executing a transaction t, we check for every variable x that t writes to ($\mathsf{log}(x) \neq \bot$) we have that $\mathsf{tstamp}(x) < t.st$ (i.e., there were no concurrent

$$\frac{\mathsf{begin} \in \mathsf{inst}(\mathsf{ls}(p).\mathsf{pc}) \quad \mathsf{img}(\mathsf{ls.tstamp}) < t.st \quad s = \mathsf{ls}(p)[\mathsf{log} \mapsto \epsilon, \mathsf{store} \mapsto \mathsf{Log}, \mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})]}{(\mathsf{ls}, \mathsf{tstamp}, \mathsf{Log}, \mathsf{lk}) \xrightarrow{\mathsf{begin}(p,\, t)} (\mathsf{ls}[p \mapsto s], \mathsf{tstamp}, \mathsf{Log}, \mathsf{lk})}$$

$$\frac{\begin{array}{r} r := x \in \mathsf{inst}(\mathsf{ls}(p).\mathsf{pc}) \quad \mathsf{ls}(p).\mathsf{store}[x] = v \quad rval = \mathsf{ls}(p).\mathsf{rval}[r \mapsto v] \\ s = \mathsf{ls}(p)[\mathsf{rval} \mapsto rval, \mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})] \end{array}}{(\mathsf{ls}, \mathsf{tstamp}, \mathsf{Log}, \mathsf{lk}) \xrightarrow{\mathsf{ld}(p,\, t,\, x,\, v)} (\mathsf{ls}[p \mapsto s], \mathsf{tstamp}, \mathsf{Log}, \mathsf{lk})}$$

$$\frac{\begin{array}{r} x := v \in \mathsf{inst}(\mathsf{ls}(p).\mathsf{pc}) \quad log = \mathsf{ls}(p).\mathsf{log}[x \mapsto 1] \quad store = \mathsf{ls}(p).\mathsf{store}[x \mapsto v] \\ s = \mathsf{ls}(p)[\mathsf{log} \mapsto log, \mathsf{store} \mapsto store, \mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})] \end{array}}{(\mathsf{ls}, \mathsf{tstamp}, \mathsf{Log}, \mathsf{lk}) \xrightarrow{\mathsf{isu}(p,\, t,\, x,\, v)} (\mathsf{ls}[p \mapsto s], \mathsf{tstamp}, \mathsf{Log}, \mathsf{lk})}$$

$$\frac{\begin{array}{r} \mathsf{end} \in \mathsf{inst}(\mathsf{ls}(p).\mathsf{pc}) \quad \forall x \in \mathbb{V}.\ log[x] = \bot \vee \mathsf{tstamp}(x) < t.st \quad \mathsf{img}(\mathsf{ls.tstamp}) < t.ct \\ Log = \mathsf{Log}[x \mapsto \mathsf{store}[x] : x \in \mathbb{V}, log[x] \neq \bot] \\ tstamp = \mathsf{tstamp}[x \mapsto st.ct : x \in \mathbb{V}, log[x] \neq \bot] \quad s = \mathsf{ls}(p)[\mathsf{pc} \mapsto \mathsf{next}(\mathsf{pc})] \end{array}}{(\mathsf{ls}, \mathsf{tstamp}, \mathsf{Log}, \mathsf{lk}) \xrightarrow{\mathsf{com}(p,\, t)} (\mathsf{ls}[p \mapsto s], \mathsf{Log} \mapsto Log, \mathsf{tstamp} \mapsto tstamp, \mathsf{lk})}$$

Figure 4.3: The set of transition rules defining snapshot isolation semantics model. We assume a transaction identifier $t : (st, ct)$ has two components.

writes to x). Then, we store the value of $\mathsf{store}(x)$ for every variable x that t writes to ($\mathsf{log}(x) \neq \bot$) in $\mathsf{Log}(x)$. Also, for every variable x that t writes to, we store $t.ct$ in $\mathsf{tstamp}(x)$.

The semantics of a program $\mathcal{P}$ under $\mathtt{SI}$ is defined as a LTS $[\mathcal{P}]_{\mathtt{SI}} = (\mathbb{C}, \mathbb{E}\mathsf{v}, \mathsf{gs}_0, \mathbb{C}_{\mathbb{F}}, \rightarrow)$ where we assume that any program configuration can be final, i.e., $\mathbb{C}_{\mathbb{F}} = \mathbb{C}$. The set of events under $\mathtt{SI}$ is defined as follow.

$$\mathbb{E}\mathsf{v} = \{\mathsf{begin}(p,t), \mathsf{ld}(p,t,x,v), \mathsf{isu}(p,t,x,v), \mathsf{com}(p,t) : p \in \mathbb{P}, t \in \mathbb{T}^2, x \in \mathbb{V}, v \in \mathbb{D}\}$$

where begin and com label transitions corresponding to the start and the commit of a transaction, respectively. isu and ld label transitions corresponding to writing, resp., reading, a shared variable during some transaction. The transition relation $\rightarrow$ is defined in Figure 4.3. For readability, the events labeling a transition are written on top of $\rightarrow$. A begin transition resets the local valuation of the shared variables and fetches their values from the central memory. A com transition applies the writes performed in a transaction to the central memory by merging the contents of the local copy store with the central memory Log. An ld transition reads the value of a shared-variable from the local copy store while an isu transition applies a new write to the local copy store.

An execution of program $\mathcal{P}$, under snapshot isolation, is a sequence of events $ev_1 \cdot ev_2 \cdot \ldots$ labeling the transitions, such that there exists a sequence of configurations $\mathsf{gs}_0 \cdot \mathsf{gs}_1 \cdot \ldots$ where gs_0 is the initial configuration before $\mathcal{P}$ starts execution and $\mathsf{gs}_{i-1} \xrightarrow{ev_i} \mathsf{gs}_i$ is a valid transition for $i > 1$. The set of executions of $\mathcal{P}$ under $\mathtt{SI}$ is denoted by $\mathbb{E}\mathsf{x}_{\mathtt{SI}}(\mathcal{P})$.

4.3.1 Trace

Formally, the trace of an execution ρ, under snapshot isolation, is obtained by (1) replacing each sub-sequence of transitions in ρ corresponding to the same transaction, but excluding the com transition, with a single "macro-event" $\mathsf{isu}(p,t)$, and (2) adding several standard relations between these macro-events $\mathsf{isu}(p,t)$ and commit events $\mathsf{com}(p,t)$ to record the data-flow in ρ, e.g. which transaction wrote the value read by another transaction. The sequence of $\mathsf{isu}(p,t)$ and $\mathsf{com}(p,t)$ events obtained in the first step is called a *summary of* ρ. We say that a transaction t in ρ performs an *external read* of a variable x if ρ contains an event $\mathsf{ld}(p,t,x,v)$ which is not preceded by a write on x of t, i.e., an event $\mathsf{isu}(p,t,x,v)$. Also, we say that a transaction t *writes* a variable x if ρ contains an event $\mathsf{isu}(p,t,x,v)$, for some v.

The *trace* $\mathsf{tr}(\rho) = (\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{STO})$ of an execution ρ consists of the summary τ of ρ along with the *program order* PO, which relates any two issue events $\mathsf{isu}(p,t)$ and $\mathsf{isu}(p,t')$ that occur in this order in τ, *write-read* relation WR, which relates any two events $\mathsf{com}(p,t)$ and $\mathsf{isu}(p',t')$ that occur in this order in τ such that t' performs an external read of x, and $\mathsf{com}(p,t)$ is the last event in τ before $\mathsf{isu}(p',t')$ that writes to x (to mark the variable x, we may use $\mathsf{WR}(x)$), the *write-write* order WW, which relates any two store events $\mathsf{com}(p,t)$ and $\mathsf{com}(p',t')$ that occur in this order in τ and write to the same variable x (to mark the variable x, we may use $\mathsf{WW}(x)$), the *read-write* relation RW, which relates any two events $\mathsf{isu}(p,t)$ and $\mathsf{com}(p',t')$ that occur in this order in τ such that t reads a value that is overwritten by t', and the *same-transaction* relation STO, which relates the issue event with the commit event of the same transaction. The read-write relation RW is formally defined as $\mathsf{RW}(x) = \mathsf{WR}^{-1}(x); \mathsf{WW}(x)$ and $\mathsf{RW} = \bigcup_{x \in \mathbb{V}} \mathsf{RW}(x)$. If a transaction t reads the initial value of x then $\mathsf{RW}(x)$ relates $\mathsf{isu}(p,t)$ to $\mathsf{com}(p',t')$ of any other transaction t' which writes to x (i.e., $(\mathsf{isu}(p,t), \mathsf{com}(p',t')) \in \mathsf{RW}(x)$) (note that in the above relations, p and p' might designate the same process).

Since we reason about only one trace at a time, to simplify the writing, we may say that a trace is simply a sequence τ as above, keeping the relations PO, WR, WW, RW, and STO implicit. The set of traces of executions of a program $\mathcal{P}$ under $\mathtt{SI}$ is denoted by $\mathbb{T}\mathsf{r}(\mathcal{P})_{\mathtt{SI}}$.

Happens before order. We introduce the *happens-before* relation on the events of a given trace as the transitive closure of the union of all the relations in the trace, i.e., $\mathsf{HB} = (\mathsf{PO} \cup \mathsf{WW} \cup$

$\mathsf{WR} \cup \mathsf{RW} \cup \mathsf{STO})^+$. The happens-before relation between events is extended to transactions as follows: a transaction t_1 *happens-before* another transaction $t_2 \neq t_1$ if the trace *tr* contains an event of transaction t_1 which happens-before an event of t_2. The happens-before relation between transactions is denoted by HB_t and called *transactional happens-before*. For a trace of serializable execution, the transactional happens-before and the happens-before relation coincide.

4.4 Robustness Against SI Relative to SER

Given a trace $tr = (\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{STO})$ such that every event $\mathsf{isu}(p, t)$ in τ is immediately followed by $\mathsf{com}(p, t)$. For simplicity, we write τ as a sequence of "atomic macro-events" (p, t) where (p, t) denotes a sequence $\mathsf{isu}(p, t) \cdot \mathsf{com}(p, t)$. We say that t is *atomic*. Thus, $(\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$ is a trace of a serializable execution as defined in Section 2.3.

Similar to Theorem 3.3, the following result characterizes traces of serializable executions. The transactional happens-before of any trace under SER is acyclic, and conversely, any trace obtained under SI with an acyclic transactional happens-before can be transformed into a trace under SER by successive swaps of consecutive events in its summary, which are not related by happens-before. Since multiple executions/traces can have the same (transactional) happens-before. It is possible that a trace *tr* produced by snapshot isolation has has an acyclic transactional happens-before even though $\mathsf{isu}(p, t)$ events may not be immediately followed by $\mathsf{com}(p, t)$ events. However, *tr* would be equivalent, up to reordering of "independent" (or commutative) transitions, to a trace of serializable execution.

Theorem 4.1 ([25, 156]). *For any trace $tr \in \mathbb{T}r_{\mathtt{SER}}(\mathcal{P})$, the transactional happens-before of tr is acyclic. Moreover, for any trace $tr = (\tau, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{STO}) \in \mathbb{T}r_{\mathtt{SI}}(\mathcal{P})$, if the transactional happens-before of tr is acyclic, then there exists a permutation τ' of τ such that $(\tau', \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}, \mathsf{STO}) \in \mathbb{T}r_{\mathtt{SER}}(\mathcal{P})$.*

As a consequence of Theorem 4.1, we define a trace *tr* to be *serializable* if it *has the same happens-before relations* as a trace of a serializable execution. Let $\mathbb{T}r_{\mathtt{SER}}(\mathcal{P})$ denote the set of serializable traces of a program $\mathcal{P}$.

We now consider the problem of checking whether the snapshot isolation semantics of a program produces only serializable traces.

Definition 4.1. *A program $\mathcal{P}$ is called* robust *against a snapshot isolation relative to serializability iff $\mathbb{T}r_{\mathtt{SI}}(\mathcal{P}) = \mathbb{T}r_{\mathtt{SER}}(\mathcal{P})$.*

By Theorem 4.1, the transactional happens-before HB_t of a robustness violation $tr \in \mathbb{T}r_{\mathtt{SI}}(\mathcal{P}) \setminus \mathbb{T}r_{\mathtt{SER}}(\mathcal{P})$ is cyclic.

4.5 Minimal Violations

A trace which is not serializable must contain at least an issue and a commit event of the same transaction that do not occur one after the other even after reordering of "independent" events. Thus, there must exist an event that occur between the two which is related to both events via the happens-before relation, forbidding the issue and commit to be adjacent. Otherwise, we can build another trace with the same happens-before where events are reordered such that the issue is immediately followed by the corresponding commit. The latter is a serializable trace which contradicts the initial assumption.

We deduce from above, that a violation trace must contain at least an issue and a commit events of the same transaction that are related via the happens-before through relation (Definition 3.3). Otherwise, we can build another trace with the same happens-before where events are reordered such that every issue $\mathsf{isu}(p,t)$ is immediately followed by the corresponding commit $\mathsf{com}(p,t)$. The latter is a serializable trace which contradicts the initial assumption.

Lemma 4.1. *Given a violation τ, there must exist a transaction t such that $\tau = \alpha \cdot \mathsf{isu}(p,t) \cdot \beta \cdot \mathsf{com}(p,t) \cdot \gamma$ and $\mathsf{isu}(p,t)$ happens before $\mathsf{com}(p,t)$ through β where α, β, and γ are sequences of events.*

Given a violation of the from $\tau = \alpha \cdot \mathsf{isu}(p,t) \cdot \beta \cdot \mathsf{com}(p,t) \cdot \gamma$, we call t a *delayed* transaction in the trace τ when $\mathsf{isu}(p,t)$ happens before $\mathsf{com}(p,t)$ through β.

We define the *number of delays* for a robustness violation τ, denoted by $\#(\tau)$, as the total number of *delayed* transactions in τ.

Definition 4.2 (Minimal violation)**.** *For a given program $\mathcal{P}$, a violation τ is called* minimal *if it has the least number of delays among all possible violations.*

The characterization of robustness violations under $\mathtt{SI}$ is stated in the following theorem.

$$\begin{tikzcd}\mathsf{isu}(p1,t1) \arrow[r, bend left, "RW"] & (p2,t2) \arrow[r, bend left, "RW"] & \mathsf{com}(p1,t1)\end{tikzcd}$$

(a) A violation of WS program.

$$\begin{tikzcd}\mathsf{isu}(p3,t3) \arrow[r, bend left, "RW"] & (p1,t1) \arrow[r, bend left, "WR"] & (p2,t2) \arrow[r, bend left, "RW"] & \mathsf{com}(p3,t3)\end{tikzcd}$$

(b) A violation of RWC program.

Figure 4.4: (a) Corresponds to a violation where $\beta = (p2, t2)$, t corresponds to $t1$, and $a = b = (p2, t2)$. (b) Corresponds to a violation where $\beta = (p1, t1) \cdot (p2, t2)$, t corresponds to $t1$, and a and b correspond to $(p1, t1)$ and $(p2, t2)$, respectively.

Theorem 4.2. *A program $\mathcal{P}$ is not robust under* $\mathtt{SI}$ *iff there exists a minimal violation in* $\mathbb{T}r(\mathcal{P})_{\mathtt{SI}}$ *of the following form:*

$$\tau_{\mathtt{SI}} = \alpha \cdot \mathit{isu}(p,t) \cdot \beta \cdot \mathit{com}(p,t) \text{ where:}$$

(a) $\mathit{isu}(p,t)$ is the issue of the only delayed transaction in τ; (Lemmas 4.3);

(b) $\mathit{isu}(p,t)$ happens before $\mathit{com}(p,t)$ through β (Lemma 4.1);

(c) for any event $a \in \beta$, we have that $(\mathit{isu}(p,t), a) \in \mathsf{HB}$ and $(a, \mathit{com}(p,t)) \in \mathsf{HB}$ (Lemma 4.1);

(d) there exist events a and b in β such that $(\mathit{isu}(p,t), a) \in \mathsf{RW}(x)$ and $(b, \mathit{com}(p,t)) \in \mathsf{RW}(y)$ with $x \neq y$ (Lemma 4.5);

(e) all transactions in β do not write to shared variables that t writes to (Lemma 4.4).

Above, $\tau_{\mathtt{SI}}$ contains a single delayed transaction. The theorem above allows $\alpha = \epsilon$ and $\beta = a = b$. Figure 4.4 shows two violations against $\mathtt{SI}$ relative to $\mathtt{SER}$ of the form given in Theorem 4.2.

In the following, we give a series lemmas that constitute Theorem 4.2. For the remainder of the chapter, we write a minimal violation in the shape $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta \cdot \mathsf{com}(p,t) \cdot \gamma$ to say that t is the *first* delayed transaction in τ (w.r.t. the order between issue events of delayed transactions) and all the events in the sequence α_{A} are atomic macro events. The following lemma shows that we can assume w.l.o.g. that γ contains only commit events.

Lemma 4.2. *Let $\tau = \alpha_{\mathsf{A}} \cdot \mathit{isu}(p,t) \cdot \beta \cdot \mathit{com}(p,t) \cdot \gamma$ be a minimal violation such that $\mathit{isu}(p,t)$ happens-before $\mathit{com}(p,t)$ through β. Then, $\tau' = \alpha_{\mathsf{A}} \cdot \mathit{isu}(p,t) \cdot \beta \cdot \mathit{com}(p,t) \cdot \gamma'$, such that γ' contains only commit events from delayed transactions is also a minimal violation.*

Proof. Similar to the proof of Lemma 3.4. □

Next lemma shows that in a minimal violation $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta \cdot \mathsf{com}(p,t) \cdot \gamma$ such that γ contains only commit events, we have β contains no delayed transaction and $\gamma = \epsilon$. We show that if it were to have a delayed transaction t_0 in β, then it is possible to obtain a new violation where either t is not delayed and obtain a new violation with a smaller number of delayed transactions which contradicts the minimality assumption.

Lemma 4.3. *Let $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta \cdot \mathsf{com}(p,t) \cdot \gamma$ be a minimal violation such that γ contains only commit events. Then, β does not contain delayed transactions and $\gamma = \epsilon$.*

Proof. We suppose by contradiction that β contains a delayed transaction t_0 issued by a process p_0.

It is important to notice that there must exist $\beta' \subset \beta$ and $\mathsf{com}(p_0,t_0) \in \beta$ such that $\mathsf{isu}(p_0,t_0)$ happens before $\mathsf{com}(p_0,t_0)$ through β. Otherwise, we can commute the events until $\mathsf{com}(p_0,t_0)$ occurs just after $\mathsf{isu}(p_0,t_0)$ and in this case transaction t_0 is not delayed by p_0. Thus, τ is of the form $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta_1 \cdot \mathsf{isu}(p_0,t_0) \cdot \beta' \cdot \mathsf{com}(p_0,t_0) \cdot \beta_2 \cdot \mathsf{com}(p,t) \cdot \gamma$ ($\beta = \beta_1 \cdot \mathsf{isu}(p_0,t_0) \cdot \beta' \cdot \mathsf{com}(p_0,t_0) \cdot \beta_2$ where we assume w.l.o.g that $\mathsf{com}(p_0,t_0)$ occurs in β).

Since in $\beta_1 \cdot \mathsf{isu}(p_0,t_0) \cdot \beta' \cdot \mathsf{com}(p_0,t_0) \cdot \beta_2$ no event depends on $\mathsf{isu}(p,t)$. Thus, we can safely remove $\mathsf{isu}(p,t)$ and its associated commit event $\mathsf{com}(p,t)$ and obtains: $\tau' = \alpha_{\mathsf{A}} \cdot \beta_1 \cdot \mathsf{isu}(p_0,t_0) \cdot \beta' \cdot \mathsf{com}(p_0,t_0) \cdot \beta_2 \cdot \gamma$. which is an violation because of the transactional happens-before cycle caused by $\mathsf{isu}(p_0,t_0)$ happens-before $\mathsf{com}(p_0,t_0)$ through β'. In τ', transaction t was not delayed, therefore, τ' has less number of delayed transactions than τ. Thus, τ is not a minimal violation, a contradiction to our hypothesis. Therefore, t is the only delayed transaction in τ which implies that $\gamma = \epsilon$. □

An important property of `SI` is that the execution of concurrent transactions that write to the same location is disallowed. Thus, the event $\mathsf{com}(p,t)$ can take place only if there are no concurrent writes that were committed after $\mathsf{isu}(p,t)$ and write to the same variables as $\mathsf{com}(p,t)$.

Lemma 4.4. *Let $\tau = \alpha_{\mathsf{A}} \cdot \mathsf{isu}(p,t) \cdot \beta \cdot \mathsf{com}(p,t)$ be a minimal violation such that $\mathsf{isu}(p,t)$ happens-before $\mathsf{com}(p,t)$ through β. Then, for every $a \in \beta$, a does not write to a shared variable that $\mathsf{com}(p,t)$ writes to.*

The next lemma characterizes the relation between the delayed transaction and the commit of the underlying transaction. It shows the type of the first and last happens-before relations in the happens-before path between the issue event of the delayed transaction and its corresponding commit event.

Lemma 4.5. *Let $\tau = \alpha_{\mathsf{A}} \cdot \mathit{isu}(p,t) \cdot \beta \cdot \mathit{com}(p,t)$ be a minimal anomaly under* `SI`*. Then, the following must hold: there exist (p_0,t_0), $(p_1,t_1) \in \beta$ ((p_0,t_0) and (p_1,t_1) might be identical) where $(\mathit{isu}(p,t),(p_0,t_0)) \in \mathsf{RW}$, and $((p_1,t_1),\mathit{com}(p,t)) \in \mathsf{RW}$.*

Proof. The proof comes from the fact that β does not contain events that write to the same variables that t writes to. □

As a direct consequence of Theorem 4.3, the next corollary states that certain classes of programs are robust against `SI` relative to `SER`.

Corollary 4.1. *Given a program $\mathcal{P}$, if one of the following holds:*

(a) every transaction of $\mathcal{P}$ contains a single instruction (either a read or a write);

(b) every transaction of $\mathcal{P}$ contains only read/write operations that access a single variable (different transactions might read (reps., write) from (resp., to) different variables);

(c) there exists a shared-variable x, every transaction of $\mathcal{P}$ contains a write to x.

then $\mathcal{P}$ is robust against `SI` *relative to* `SER`*.*

4.6 Reducing Robustness to SC Reachability

We define a program instrumentation which mimics the delay of a transaction by doing the writes on auxiliary variables which are not visible to other transactions. After the delay of a transaction, we track happens-before dependencies until we execute a transaction that does a "read" on one of the variables that the delayed transaction writes to (this would expose a read-write dependency to the commit event of the delayed transaction). While tracking happens-before dependencies we cannot execute a transaction that writes to a variable that the delayed transaction writes to since `SI` forbids write-write conflicts between concurrent transactions.

Concretely, given a program $\mathcal{P}$, we define an instrumentation of $\mathcal{P}$ such that $\mathcal{P}$ is not robust against `SI` iff the instrumentation reaches an error state under serializability. The instrumentation uses auxiliary variables in order to simulate a *single* delayed transaction which we prove that it is enough for deciding robustness. Let $\mathsf{isu}(p,t)$ be the issue event of the only delayed transaction. The process p that delayed t is called the *Attacker*. When the attacker finishes executing the delayed transaction it stops. Other processes that execute transactions afterwards are called *Happens-Before Helpers*.

The instrumentation uses two copies of the set of shared variables in the original program to simulate the delayed transaction. We use primed variables x' to denote the second copy. Thus, when a process becomes the attacker, it will only write to the second copy that is not visible to other processes including the happens-before helpers. The writes made by the other processes including the happens-before helpers are made visible to all processes.

When the attacker delays the transaction t, it keeps track of the variables it accessed, in particular, it stores the name of one of the variables it writes to, x, it tracks every variable y that it reads from and every variable z that it writes to. When the attacker finishes executing t, and some other process wants to execute some other transaction, the underlying transaction must contain a write to a variable y that the attacker reads from. Also, the underlying transaction must not write to a variable that t writes to. We say that this process has joined happens-before helpers through the underlying transaction. While executing this transaction, we keep track of each variable that was accessed and the type of operation, whether it is a read or write. Afterward, in order for some other transaction to "join" the happens-before path, it must not write to a variable that t writes to so it does not violate the fact that `SI` forbids write-write conflicts, and it has to satisfy one of the following conditions in order to ensure the continuity of the happens-before dependencies: (1) the transaction is issued by a process that has already another transaction in the happens-before dependency (program order dependency), (2) the transaction is reading from a shared variable that was updated by a previous transaction in the happens-before dependency (write-read dependency), (3) the transaction writes to a shared variable that was read by a previous transaction in the happens-before dependency (read-write dependency), or (4) the transaction writes to a shared variable that was updated by a previous transaction in the happens-before dependency (write-write dependency). We introduce a flag for each shared variable to mark the fact that the variable was read or written by a previous transaction.

Processes continue executing transactions as part of the chain of happens-before dependencies, until a transaction does a read on the variable x that t wrote to. In this case, we reached an error state which signals that we found a cycle in the transactional happens-before relation.

The instrumentation uses four varieties of flags: a) global flags (i.e., HB, $a_{\mathsf{tr_A}}$, $a_{\mathsf{st_A}}$), b) flags local to a process (i.e., $p.a$ and $p.hbh$), and c) flags per shared variable (i.e., $x.event$, $x.event'$, and

$x.eventI$). We will explain the meaning of these flags along with the instrumentation. At the start of the execution, all flags are initialized to null ($\perp$).

Whether a process is an attacker or happens-before helper is not enforced syntactically by the instrumentation. It is set non-deterministically during the execution using some additional process-local flags. Each process chooses to set to true at most one of the flags $p.a$ and $p.hbh$, implying that the process becomes an attacker or happens-before helper, respectively. At most one process can be an attacker, i.e., set $p.a$ to true. In the following, we detail the instrumentation for read and write instructions of the attacker and happens-before helpers.

4.6.1 Instrumentation of the Attacker

Figure 4.5 lists the instrumentation of the write and read instructions of the attacker. Each process passes through an initial phase where it executes transactions that are visible immediately to all the other processes (i.e., they are not delayed), and then non-deterministically it can choose to delay a transaction at which point it sets the flag $a_{\mathsf{tr_A}}$ to true. During the delayed transaction it chooses non-deterministically a write instruction to a variable x and stores the name of this variable in the flag $a_{\mathsf{st_A}}$ (line (4.5)). The values written during the delayed transaction are stored in the primed variables and are visible only to the current transaction, in case the transaction reads its own writes. For example, given a variable z, all writes to z from the original program are transformed into writes to the primed version z' (line (4.3)). Each time, the attacker writes to z, it sets the flag $z.event' = 1$. This flag is used later by transactions from happens-before helpers to avoid writing to variables that the delayed transaction writes to.

A read on a variable, y, in the delayed transaction takes her value from the primed version, y'. In every read in the delayed transaction, we set the flag $y.event$ to ld (line (4.1)) to be used latter in order for a process to join the happens-before helpers. Afterward, the attacker starts the happens-before path, and it sets the variable HB to true (line (4.2)) to mark the start of the happens. When the flag HB is set to true the attacker stops executing new transactions.

4.6.2 Instrumentation of the Happens-Before Helpers

The remaining processes, which are not the attacker, can become a happens-before helper. Figure 4.6 lists the instrumentation of write and read instructions of a happens-before helper. In a first phase, each process executes the original code until the flag $a_{\mathsf{tr_A}}$ is set to true by the attacker. This

```
$[\![ \mathsf{l}_1 \colon r := x;\ \texttt{goto}\ \mathsf{l}_2; ]\!]_{\mathsf{A}} =$
// Read before the delayed transaction
 $\mathsf{l}_1$: assume $a_{\mathsf{tr}_\mathsf{A}} = \bot$ ; goto $\mathsf{l}_{x1}$;
 $\mathsf{l}_{x1}$: $r := x$; goto $\mathsf{l}_2$;
// Read in the delayed transaction
 $\mathsf{l}_1$: assume $a_{\mathsf{tr}_\mathsf{A}} \neq \bot \wedge p.a \neq \bot$ ; goto $\mathsf{l}_{x2}$;
 $\mathsf{l}_{x2}$: $r := x'$; goto $\mathsf{l}_{x3}$;
 $\mathsf{l}_{x3}$: $x.event := \mathsf{ld}$; goto $\mathsf{l}_{x4}$;          (4.1)
 $\mathsf{l}_{x4}$: assume $\mathsf{HB} = \bot$ ; goto $\mathsf{l}_{x5}$;
 $\mathsf{l}_{x5}$: $\mathsf{HB} := \mathsf{true}$; goto $\mathsf{l}_2$;            (4.2)
 $\mathsf{l}_{x4}$: assume $\mathsf{HB} \neq \bot$ ; goto $\mathsf{l}_2$;
```

```
$[\![ \mathsf{l}_1 \colon x := e;\ \texttt{goto}\ \mathsf{l}_2; ]\!]_{\mathsf{A}} =$
// Write before the delayed transaction
 $\mathsf{l}_1$: assume $a_{\mathsf{tr}_\mathsf{A}} = \bot$ ; goto $\mathsf{l}_{x1}$;
 $\mathsf{l}_{x1}$: $x := e$; goto $\mathsf{l}_2$;
// Write in the delayed transaction
 $\mathsf{l}_1$: assume $a_{\mathsf{tr}_\mathsf{A}} \neq \bot \wedge p.a \neq \bot$ ; goto $\mathsf{l}_{x2}$;
 $\mathsf{l}_{x2}$: $x' := e$; goto $\mathsf{l}_{x3}$;                          (4.3)
 $\mathsf{l}_{x3}$: $x.event' := 1$; goto $\mathsf{l}_2$;                      (4.4)
// Special write in the delayed transaction
 $\mathsf{l}_1$: assume $a_{\mathsf{st}_\mathsf{A}} = x.event = \bot \wedge a_{\mathsf{tr}_\mathsf{A}} \neq \bot$ ; goto $\mathsf{l}_{x4}$;
 $\mathsf{l}_{x4}$: $x' := e$; goto $\mathsf{l}_{x5}$;
 $\mathsf{l}_{x5}$: $a_{\mathsf{st}_\mathsf{A}} :=$ '$x$'; goto $\mathsf{l}_{x6}$;                   (4.5)
 $\mathsf{l}_{x8}$: $x.event' := 1$; goto $\mathsf{l}_2$;
```

Figure 4.5: Instrumentation of the Attacker. We use 'x' to denote the name of the shared variable x.

flag signals the "creation" of the secondary copy of the shared-variables, which can be observed only by the attacker. At this point, the flag HB is set to `true`, and the happens-before helper process chooses non-deterministically a first transaction through which it wants to join the set of happens-before helpers, i.e., continue the happens-before dependency created by the existing happens-before helpers. When a process chooses a transaction, it makes a pledge (while executing the `begin` instruction) that during this transaction it will either read from a variable that was written to by another happens-before helper, write to a variable that was accessed (read or written) by another happens-before helper, or write to a variable that was read from in the delayed transaction. When the pledge is met, the process sets the flag $p.hbh$ to `true` (lines (4.7) and (4.11)). The execution is blocked if a process does not keep its pledge (i.e., the flag $p.hbh$ is null) at the end of the transaction. Note that the first process to join the happens-before helper has to execute a transaction t which writes to a variable that was read from in the delayed transaction since this is the only way to build a happens-before between t, and the delayed transaction (PO is not possible since t is not from the attacker, WR is not possible since t does not see the writes of the delayed transaction, and WW is not possible since t cannot write to a variable that the delayed transaction writes to). We use a flag $x.event$ for each variable x to record the type (read ld or write st) of the last access made by

```
$[\![ l_1 : r := x;\ \texttt{goto}\ l_2; ]\!]_{\mathsf{HbH}} =$

// Read before the delayed transaction
 $l_1$: assume $\mathsf{HB} = \perp \wedge p.a = \perp$ ; goto $l_{x1}$;
 $l_{x1}$: $r := x$; goto $l_2$;                                   (4.6)
// Read after the delayed transaction
 $l_1$: assume $\mathsf{HB} \neq \perp$ ; goto $l_{x2}$;
 $l_{x2}$: $r := x$; goto $l_{x3}$;
 $l_{x3}$: assume $x.eventI = \mathsf{st} \wedge p.hbh = \perp$ ; goto $l_{x4}$;
 $l_{x4}$: $p.hbh := \mathsf{true}$; goto $l_2$;                   (4.7)
 $l_{x3}$: assume $x.event = \perp$ ; goto $l_{x5}$;
 $l_{x5}$: $x.event := \mathsf{ld}$; goto $l_2$;                   (4.8)
 $l_{x3}$: assume $x.event \neq \perp \vee p.hbh \neq \perp$ ; goto $l_2$;
```

```
$[\![ l_1 : x := e;\ \texttt{goto}\ l_2; ]\!]_{\mathsf{HbH}} =$

// Write before the delayed transaction
 $l_1$: assume $\mathsf{HB} = \perp \wedge a_{\mathsf{tr_A}} = \perp$ ; goto $l_{x1}$;
 $l_{x1}$: $x := e$; goto $l_2$;
// Write after the delayed transaction
 $l_1$: assume $\mathsf{HB} \neq \perp \wedge p.a = \perp$ ; goto $l_{x2}$;
 $l_{x2}$: assume $x.event' \neq \perp$ ; assume false;             (4.9)
 $l_{x2}$: assume $x.event' = \perp$ ; goto $l_{x3}$;
 $l_{x3}$: $x := e$; goto $l_{x4}$;
 $l_{x4}$: $x.event := \mathsf{st}$; goto $l_{x5}$;                 (4.10)
 $l_{x5}$: assume $x.eventI \neq \perp \wedge p.hbh = \perp$ ; goto $l_{x6}$;
 $l_{x6}$: $p.hbh := \mathsf{true}$; goto $l_2$;                    (4.11)
 $l_{x5}$: assume $x.eventI = \perp \vee p.hbh \neq \perp$ ; goto $l_2$;
```

Figure 4.6: Instrumentation of Happens-Before Helpers.

a happens-before helper (lines (4.8) and (4.10)). During the execution of a transaction that is part of the happens-before dependency, we must ensure that the transaction does not write to variable y where $y.even'$ is set to 1. Otherwise, the execution is blocked (line 4.9).

The happens-before helpers continue executing their instructions, until one of them reads from the shared variable x whose name was stored in $a_{\mathsf{st_A}}$. This establishes a happens-before dependency between the delayed transaction and a “fictitious” store event corresponding to the delayed transaction that could be executed just after this read of x. The execution does not have to contain this store event explicitly since it is always enabled. Therefore, at the end of every transaction, the instrumentation checks whether the transaction read x. If it is the case, then the execution stops and goes to an error state to indicate that this is a robustness violation. Notice that after the attacker stops, the only processes that are executing transactions are happens-before helpers, which is justified since when a process is not from a happens-before helper it implies that we cannot construct a happens-before dependency between a transaction of this process and the delayed transaction which means that the two transactions commute which in turn implies that this process's transactions can be executed before executing the delayed transaction of the attacker.

4.6.3 Correctness

The role of a process in an execution is chosen non-deterministically at runtime. Therefore, the final instrumentation of a given program $\mathcal{P}$, denoted by $[\![\mathcal{P}]\!]$, is obtained by replacing each labeled instruction $\langle linst\rangle$ with the concatenation of the instrumentations corresponding to the attacker and the happens-before helpers, i.e., $[\![\langle linst\rangle]\!] ::= [\![\langle linst\rangle]\!]_{\mathsf{A}}\ [\![\langle linst\rangle]\!]_{\mathsf{HbH}}$

The following theorem states the correctness of the instrumentation.

Theorem 4.3. *$\mathcal{P}$ is not robust against* SI *iff $[\![\mathcal{P}]\!]$ reaches the error state.*

If a program is not robust, this implies that the execution of the program under SI results in a trace where the happens-before is cyclic. Which is possible only if the program contains at least one delayed transaction. In the proof of this theorem, we show that is sufficient to search for executions that contain a single delayed transaction.

Notice that in the instrumentation of the attacker, the delayed transaction must contain a read and write instructions on different variables. Also, the transactions of the happens-before helpers must not contain a write to a variable that the delayed transaction writes to.

Proof. **Soundness.** Suppose that the instrumented program reaches an error state. Then, the execution's trace of the instrumented program is of the form:

$$\tau_\star = \tau_1 \cdot \mathsf{isu}(p,t) \cdot \tau_2 \cdot (p',t')$$

The last transaction, (p',t') performed by a process p'' that does a read accessing the variable $x = a_{\mathsf{st_A}}$ and is part of the happens-before helpers. This is because the conditional check can be performed only by a process (p_{HbH1}) that is one of the happens-before helpers and is currently executing.

In order for p_{HbH1}, to join the set of happens-before helpers, it must have found that the valuation of the flag HB is not null which means there exists some process p that is the attacker that sets the flag HB to true. In τ_1, the attacker, happens-before helpers, and other processes start executing the original instructions without setting any flags or delaying any transactions. Afterwards, the attacker issues the delayed transaction $\mathsf{isu}(p,t)$ and it starts populating the primed variables x' and reading from them and setting the flags $x.event'$ to 1 for every variable x that it writes to and $y.event$ to ld for every variable y that it reads from. During the execution of t, the attacker sets the flag HB to true. Hence, the happens-before helpers start checking at every instruction whether the flags $x.event$ are set to either st or ld. If so, they start populating the flags $x.event$ and $l.event$ as well. When HB is set to true, the attacker stop issuing new transactions. Therefore, all transaction in τ_2 are from the happens-before helpers.

We now transform $\tau_\star$ into the following execution trace:

$$\tau = \tau_1' \cdot \mathsf{isu}(p,t) \cdot \tau_2' \cdot \mathsf{com}(p,t)$$

Here, τ_1' is the subsequence of all τ_1 events that are produced by instructions from $\mathcal{P}$ without the conditionals checking (i.e., the assume statements). The transaction t which is executed by the attacker represents the delayed transactions in τ with the removal of the conditionals checking and the flags setting. τ_2' is the subsequence of all events of τ_2 produced by transactions from $\mathcal{P}$ which are executed only by the happens-before helpers except the conditionals checking and the flags setting. We add the commit event $\mathsf{com}(p_0,t)$ to describe the commit of the delayed transaction that was delayed by the attacker. τ is a possible execution's trace of the program $\mathcal{P}$ because $\tau_\star$ is a result of an execution of the instrumented version of $\mathcal{P}$ and we have removed from τ all the effects of the instrumentation, and replaced the stores to auxiliary variables by issues of stores without changing the dependency between all the events in the execution.

All transactions in τ_2' are from the happens-before helpers. Transactions in τ_2' form a happens-before path between $\mathsf{isu}(p,t)$ and $\mathsf{com}(p,t)$. Also, we have $a,\ b = (p',t') \in \tau_2'$ such that $(\mathsf{isu}(p,t), a) \in \mathsf{RW}(y)$ and $(b, \mathsf{com}(p,t)) \in \mathsf{RW}(x)$. No transaction in τ_2' writes to a variable that t writes to. Hence, τ indeed holds all the properties of the violation described in Theorem 4.2.

Completeness. Suppose we have a violation of a given program $\mathcal{P}$:

$$\tau = \tau_1 \cdot \mathsf{isu}(p,t) \cdot \tau_2 \cdot \mathsf{com}(p,t)$$

such that τ maintains all the properties given in Theorem 4.2. We demonstrate that there is a possible serializable execution based on τ of the instrumented version of the program $\mathcal{P}$ that reaches the error state. We show how to build the instrumented program execution. At the start of the execution, τ_1, the attacker, happens-before helpers, and other processes execute the original transactions with just conditional checks. Afterwards, the attacker delays the transaction $\mathsf{isu}(p,t)$ and starts populating the flags. In $\mathsf{isu}(p,t)$, the attacker issues a store to the shared variable 'x' $= a_{\mathsf{st}_\mathsf{A}}$ and $\exists\ b \in \tau_2$ such that $(b, \mathsf{com}(p,t)) \in \mathsf{RW}(x)$. All writes that were executed in t by the attacker are invisible to the remaining processes which includes the happens-before helpers. While executing t, the attacker sets the content of the flag $y.event$ to ld for every variable y that it reads from and it sets the flag HB to true.

On the other hand, the processes which are executing their transactions without delaying them will attempt to join the happens-before helpers by checking if the flag HB is set to true. If so, they start the attempt of joining the happens-before helpers and when it succeed they joining the happens-before helpers and start executing their transactions which constitute τ_2. The first executed transaction by the happens-before helpers is a described above which signals the start of τ_2 and the happen before dependency. Thus, in τ_2, we have only transactions form the happens-before helpers (because the attacker stop when the flag HB is set to true) such that they are related by the happen before dependency that started from $\mathsf{isu}(p,t)$ until it reaches $\mathsf{com}(p,t)$ through τ_2. We know that there must exist $b \in \tau_2$ such that $(b, \mathsf{com}(p,t)) \in \mathsf{RW}(`x` = a_{\mathsf{st_A}})$. b is equivalent to the last executed transaction by the happens-before helpers that accesses the shared variable x. Thus, the underlying happens-before helper will set the content of the flag $x.event$ to ld. Hence, when the underlying process executes the com instruction of this transaction, it will go to the error state (lines (3.32), (3.33), and (3.34)) and in this case the instrumented version of the program $\mathcal{P}$ has reached the desired error state. $\square$

The following corollary states the complexity of checking robustness for finite-state programs[1] against snapshot isolation relative to serializability. The upper bound is a direct consequence of Theorem 4.3 and of previous results concerning the reachability problem in concurrent programs running over a sequentially-consistent memory, with a fixed [108] or parametric number of processes [146]. For the lower bound, given an instance of the reachability problem under sequential consistency, denoted by $(\mathcal{P}, \ell)$, we construct a program $\mathcal{P}'$ where each statement s of $\mathcal{P}$ is a different transaction (guarded by a global lock), and where reaching the location ℓ enables the execution of a "gadget" that corresponds to the WS program in Figure 4.1a. Executing each statement under a global lock ensures that every execution of $\mathcal{P}'$ under snapshot isolation is serializable, and faithfully represents an execution of the original $\mathcal{P}$ under sequential consistency. Moreover, $\mathcal{P}$ reaches ℓ iff $\mathcal{P}'$ contains a robustness violation, which is due to the execution of WS.

Corollary 4.2. *Checking robustness of finite-state programs against snapshot isolation is PSPACE-complete when the number of processes is fixed and EXPSPACE-complete, otherwise.*

[1]Programs with a bounded number of variables taking values from a bounded domain.

4.7 Proving Program Robustness

As a more pragmatic alternative to the reduction in the previous section, we define an approximated method for proving robustness which is inspired by Lipton's reduction theory [120].

Movers. Given an execution $\tau = ev_1 \cdot \ldots \cdot ev_n$ of a program $\mathcal{P}$ under serializability (where each event ev_i corresponds to executing an entire transaction), we say that the event ev_i *moves right (resp., left)* in τ if $ev_1 \cdot \ldots \cdot ev_{i-1} \cdot ev_{i+1} \cdot ev_i \cdot ev_{i+2} \cdot \ldots \cdot ev_n$ (resp., $ev_1 \cdot \ldots \cdot ev_{i-2} \cdot ev_i \cdot ev_{i-1} \cdot ev_{i+1} \cdot \ldots \cdot ev_n$) is also a valid execution of $\mathcal{P}$, the process of ev_i is different from the process of ev_{i+1} (resp., ev_{i-1}), and both executions reach to the same end state σ_n. For an execution τ, let $\mathsf{instOf}_\tau(ev_i)$ denote the transaction that generated the event ev_i. A transaction t of a program $\mathcal{P}$ is a *right (resp., left) mover* if for all executions τ of $\mathcal{P}$ under serializability, the event ev_i with $\mathsf{instOf}(ev_i) = t$ moves right (resp., left) in τ.

If a transaction t is not a right mover, then there must exist an execution τ of $\mathcal{P}$ under serializability and an event ev_i of τ with $\mathsf{instOf}(ev_i) = t$ that does not move right. This implies that there must exist another ev_{i+1} of τ which caused ev_i to not be a right mover. Since ev_i and ev_{i+1} do not commute, then this must be because of either a write-read, write-write, or a read-write dependency. If $t' = \mathsf{instOf}(ev_{i+1})$, we say that t is not a right mover because of t' and some dependency that is either write-read, write-write, or read-write. Notice that when t is not a right mover because of t' then t' is not a left mover because of t.

We define M_{WR} as a binary relation between transactions such that $(t, t') \in \mathsf{M}_{\mathsf{WR}}$ when t is *not* a right mover because of t' and a write-read dependency. We define the relations M_{WW} and M_{RW} corresponding to write-write and read-write dependencies in a similar way.

Read/Write-free transactions. Given a transaction t, we define $t \setminus \{r\}$ as a variation of t where all the reads from shared variables are replaced with non-deterministic reads, i.e., $\langle reg \rangle := \langle var \rangle$ statements are replaced with $\langle reg \rangle := \star$ where $\star$ denotes non-deterministic choice. We also define $t \setminus \{w\}$ as a variation of t where all the writes to shared variables in t are disabled. Intuitively, recalling the reduction to SC reachability in §4.6, $t \setminus \{w\}$ simulates the delay of a transaction by the Attacker, i.e., the writes are not made visible to other processes, and $t \setminus \{r\}$ approximates the commit of the delayed transaction which only applies a set of writes.

Commutativity dependency graph. Given a program $\mathcal{P}$, we define the commutativity dependency graph as a graph where vertices represent transactions and their read/write-free variations.

Two vertices which correspond to the original transactions in $\mathcal{P}$ are related by a program order edge, if they belong to the same process. The other edges in this graph represent the "non-mover" relations M_{WR}, M_{WW}, and M_{RW}.

Given a program $\mathcal{P}$, we say that the commutativity dependency graph of $\mathcal{P}$ contains a *non-mover cycle* if there exist a set of transactions $t_0, t_1, \ldots, t_n$ of $\mathcal{P}$ such that the following hold:

(a) $(t_0'', t_1) \in \mathsf{M}_{\mathsf{RW}}$ where t_0'' is the write-free variation of t_0 and t_1 does not write to a variable that t_0 writes to;

(b) for all $i \in [1, n]$, $(t_i, t_{i+1}) \in (\mathsf{PO} \cup \mathsf{M}_{\mathsf{WR}} \cup \mathsf{M}_{\mathsf{WW}} \cup \mathsf{M}_{\mathsf{RW}})$, t_i and t_{i+1} do not write to a shared variable that t_0 writes to;

(c) $(t_n, t_0') \in \mathsf{M}_{\mathsf{RW}}$ where t_0' is the read-free variation of t_0 and t_n does not write to a variable that t_0 writes to.

A non-mover cycle approximates an execution of the instrumentation defined in §4.6 in between the moment that the Attacker delays a transaction t_0 (which here corresponds to the write-free variation t_0'') and the moment where t_0 gets committed (the read-free variation t_0').

The following theorem shows that the acyclicity of the commutativity dependency graph of a program implies the robustness of the program. Actually, the notion of robustness in this theorem relies on a slightly different notion of trace where store-order and write-order dependencies take into account values, i.e., store-order relates only writes writing different values and the write-order relates a read to the oldest write (w.r.t. execution order) writing its value. This relaxation helps in avoiding some harmless robustness violations due to for instance, two transactions writing the same value to some variable.

Theorem 4.4. *For a program $\mathcal{P}$, if the commutativity dependency graph of $\mathcal{P}$ does not contain non-mover cycles, then $\mathcal{P}$ is robust.*

Proof. We prove the contrapositive, i.e., $\neg(2) \Rightarrow \neg(1)$. In the proof, we use the result of Theorem 4.3.

Assuming that the program $\mathcal{P}$ is not robust. Then, based on Theorem 4.3 there must exist an execution of the instrumentation of $\mathcal{P}$ that reaches the error state. We suppose that t is the delayed transaction, t_{ins} is the instrumentation of t (writes are stored in auxiliary registers), and p is the attacker process. Therefore, the execution of the instrumentation of $\mathcal{P}$ that reaches the error state is of the form $\tau = \alpha \cdot (p, t_{ins}) \cdot a \cdot \beta \cdot b$ where a writes to a variable that t reads from and b reads from a variable that t writes to. We assume that b is the first event that does read that accesses a variable that t writes to. In the following we show that the commutativity dependency graph of $\mathcal{P}$ contains a non-mover cycle where t is t_0. We consider two cases, first case when $a = b$ and $\beta = \epsilon$, and second case is when $a \neq b$.

First case: $\tau = \alpha \cdot (p, t_{ins}) \cdot a$ where a writes to a variable that t reads from, reads from a variable that t writes to, and does not write to a variable that t writes to. Assume that $a = (p_1, t_1)$. Thus, we can obtain that $\tau_0 = \alpha \cdot (p_1, t_1) \cdot (p, t')$ is a trace of serializable execution of $\mathcal{P}$ where t' is the reads free instantiation of t. Since $((p_1, t_1), \mathsf{com}(p, t)) \in \mathsf{RW}$ then t_1 reads a value that t' is overwriting with a different value. Therefore, $\tau_0' = \alpha \cdot (p, t') \cdot (p_1, t_1)$ is either a trace of serializable execution with a different end state than τ_0 has or it is not a trace of serializable execution. Thus, $(t_1, t') \in \mathsf{M}_{\mathsf{RW}}$ and t_1 does not write to a variable that t writes to. Similarly, we can obtain that $\tau_n = \alpha \cdot (p, t'') \cdot (p_1, t_1)$ is a trace of serializable execution of $\mathcal{P}$ where t'' is the writes free instantiation of t. Since $(\mathsf{isu}(p, t), (p_1, t_1)) \in \mathsf{RW}$ then t'' reads a value that t_1 is overwriting with a different value. Therefore, $\tau_n' = \alpha \cdot (p_1, t_1) \cdot (p, t'')$ is either a trace of serializable execution with a different end state than τ_n has or it is not a trace of serializable execution. Thus, $(t'', t_1) \in \mathsf{M}_{\mathsf{RW}}$ and t_1 does not write to a variable that t writes to.

Second case: $\tau = \alpha \cdot (p, t_{ins}) \cdot a \cdot \beta \cdot b$ where a writes to a variable that t reads from, b reads from a variable that t writes to, and every transaction in $a \cdot \beta \cdot b$ does not write to a variable that t writes to. Assume that $a = (p_1, t_1)$ and $b = (p_n, t_n)$. Since the transactions in $(p_1, t_1) \cdot \beta \cdot (p_n, t_n)$ constitute the happens-before path in the trace τ. Then, for every (p_i, t_i), $(p_{i+1}, t_{i+1}) \in (p_1, t_1) \cdot \beta \cdot (p_n, t_n)$ we have that $((p_i, t_i), (p_{i+1}, t_{i+1})) \in (\mathsf{PO} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW})$. In the case $((p_i, t_i), (p_{i+1}, t_{i+1})) \in (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW})$, we can obtain that $\tau_i = \alpha \cdot \gamma \cdot (p_i, t_i) \cdot (p_{i+1}, t_{i+1})$ is a trace of serializable execution of $\mathcal{P}$ where γ either empty (i.e., ϵ) or $\gamma = (p_1, t_1) \cdot \cdots \cdot (p_{i-1}, t_{i-1})$. Since, $((p_i, t_i), (p_{i+1}, t_{i+1})) \in (\mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW})$, then swapping t_i and $t_i + 1$ will result in either reordering of writes or write overwrites a read, or read obtains a different value. Therefore, $\tau_i' = \alpha \cdot \gamma \cdot (p_{i+1}, t_{i+1}) \cdot (p_i, t_i)$ is either a trace of serializable execution with a different end state than τ_i has or it is not a trace of serializable execution. Thus, $(t_i, t_{i+1}) \in \mathsf{M}_{\mathsf{RW}}$. Also, we have that t_i and t_{i+1} do not write to a variable that t writes to. Similar to the first case, we can obtain that $\tau_0 = \alpha \cdot (p_1, t_1) \cdot \beta \cdot (p_n, t_n) \cdot (p, t')$ is a trace of serializable execution of $\mathcal{P}$ where t' is the reads free instantiation of t. Since $((p_n, t_n), \mathsf{com}(p, t)) \in \mathsf{RW}$ then t_n reads a value that t' is overwriting with a different one. Therefore, $\tau_0' = \alpha \cdot (p_1, t_1) \cdot \beta \cdot (p, t') \cdot (p_n, t_n)$ is either a trace of serializable execution with a different end state than τ_0 has or it is not a trace of serializable execution. Thus, $(t_n, t') \in \mathsf{M}_{\mathsf{RW}}$ and t_n does not write to a variable that t writes to. Furthermore, we can obtain that $\tau_n = \alpha \cdot (p, t'') \cdot (p_1, t_1)$ is a trace of serializable execution of $\mathcal{P}$ where t'' is the writes free instantiation of t. Since $(\mathsf{isu}(p, t), (p_1, t_1)) \in \mathsf{RW}$ then t'' reads a value that t_1 is overwriting with a different one. Then, $\tau'_n = \alpha \cdot (p_1, t_1) \cdot (p, t'')$ is either a trace of serializable execution with a different end state than τ_n has or it is not a trace of serializable execution. Thus, $(t'', t_1) \in \mathsf{M}_{\mathsf{RW}}$ and t_1 does not write to a variable that t writes to. □

4.8 Experiments

To test the applicability of our robustness checking algorithms, we have considered a benchmark of 10 applications extracted from the literature related to weakly consistent databases in general. Each application consists of a set of SQL transactions that can be called an arbitrary number of times from an arbitrary number of processes. A first set of applications are open source projects that were implemented to be run over the Cassandra database, that were used in [58]. The set is constituted of:

Cassieq-Core[2]: A distributed queue. It manipulates data stored on a single table: USERACCOUNTS. It has eight transactions: 1) AddNewAccount for adding a new account; 2) DeleteAnAccount for deleting an account; 3) AddNewKey for adding a new key to an existing account; 4) DeleteAKey for removing a key from an existing account; 5) GetAnAccount to check whether there exist an account with a given identifier; 6) GetAccounts to display all existing accounts; 7) GetAccountKeys for inspecting all the keys of a certain account; 8) GetAccountKey to check whether a certain account does hold a certain key.

Currency-Exchange[3]: A trading service. It manipulates data stored on a single table: TRADES. It has six transactions: 1) SaveTrade for registering a new trade; 2) ViewListTrades for viewing the trades that occurred before a given timestamp; 3) ViewTrade for inspecting a given trade; 4) ViewTradeUser for looking for a user who carried out a given trade; 5) GetNbTrades for inspecting the number of trades; 6) GetTradeTimeStamp for inspecting the timestamp of a given trade.

Shopping-Cart[4]: An on-line shopping service. It manipulates data stored on two tables: USERS and PRODUCTS. It has four transactions: 1) GetUser for querying whether a user exist; 2) Get-

[2]https://github.com/paradoxical-io/cassieq

[3]https://github.com/Haiyan2/Trade

[4]https://github.com/nikhilswagle/Shopping_Cart_Angular_Cassandra

ProductsByCategory for finding products that belong to a given category; 3) GetProductByUPC for finding a product by its UPC identification; 4) GetCategories for displaying all available categories.

Playlist[5]: An on-line music service. It manipulates data stored on three tables: USERS, TRACKS, and ARTISTS. It has fourteen transactions: 1) AddTrack for adding a new track; 2) GetTrack for inspecting a certain track; 3) AddUser for adding a new user; 4) GetUser for querying whether a user exist; 5) CreatePlayList for creating a new playlist for a a given user; 6) ListArtistByLetter for listing artists by the first letters of their names; 7) ListSongsByArtist for listing tracks produced by a certain artist; 8) ListSongsByGenre for listing tracks in a certain genre group; 9) AddTrackToPlaylist for adding an existing track to an existing user playlist; 10) DeleteTrackFromPlaylist for removing a track from a user playlist; 11) GetPlaylistForUser for displaying the contents of a playlist of a certain user; 12) GetPlaylistNames for displaying all the playlists of a user; 13) DeletePlayListForUser for deleting a user's playlist; 14) DeleteUser for deleting a user.

RoomStore[6]: A messages bot service. It manipulates data stored on a single table: MESSAGES. It has five transactions: 1) AddMessage for adding a new message; 2) GetLastMessage for getting the messages of a user; 3) GetMessages for displaying messages that were added in a certain date; 4) GetSpecificMessage for displaying a specific message that was added in a certain date and time; 5) GetTopicMessages for displaying messages that are in a certain topic group.

The second set of applications is extracted from the literatures and is constituted of:

TPC-C [161]: An on-line transaction processing benchmark widely used in the database community. It manipulates data stored on nine tables: WAREHOUSE, DISTRICT, STOCK, ITEMS, CUSTOMERS, HISTORY, ORDER, NEWORDER, and ORDERLINE. It has five transactions: 1) NewOrder for placing a new order on a set of items; 2) Delivery for delivering a withstanding order to a given warehouse; 3) Payment for a given customer paying a withstanding amount of credit; 4) OrderStatus for inspecting certain orders and the associated order lines; 5) StockLevel for inspecting stocks at a given warehouse and the withstanding orders at this warehouse.

SmallBank [32]: A simplified representation of a banking application. It manipulates data stored on three tables: ACCOUNT, SAVING, and CHECKING. It has five transactions: 1) Balance for inspecting both the saving and checking balances of a given user account; 2) DepositChecking for depositing a certain amount into the checking balance; 3) TransactSaving for depositing or with-

[5]https://github.com/DataStaxDocs/playlist

[6]https://github.com/mebigfatguy/roomstore

drawing into/form the saving balance; 4) Amalgamate (Amg) for moving the saving and checking balances of an account to another account checking balance; 5) WriteCheck for withdrawing from a given account's checking balance.

FusionTicket [105]: A movie ticketing application. It manipulates data stored on a single table: EVENTS. It has four transactions: 1) AddEvent for adding new event in a given venue; 2) ViewEvent for inspecting an event and the number of tickets available for this event; 3) Browse for viewing events that are planned in a given venue; 4) Purchase for buying a ticket at a certain event.

Auction [47]: An online auction application It manipulates data stored on three tables: BIDS, ITEMS, and USERS. It has five transactions: 1) RegBid for placing a bid on an item; 2) RegUser for user's registration; 3) ViewItem for viewing the number of bids for an item; 4) ViewUser for inspecting a user's name; 5) ViewUsers for displaying all registered users.

Courseware [93, 129]: A course registration service. It manipulates data stored on three tables: STUDENT, COURSE, and ENROLED. It has five transactions: 1) RegisterStudent for registering a new student; 2) AddCourse for adding a new course; 3) EnrollStudent for enrolling a given registered student in a given course; 4) RemoveCourse for removing a given course; 5) QueryCourses for inspecting available courses.

Our first experiment concerns the reduction of robustness checking to SC reachability. For each application, we have constructed a client (i.e., a program composed of transactions defined within that application) with a fixed number of processes (at most 3) and a fixed number of transactions (between 3 and 7 transactions per process). We have encoded the instrumentation of this client, defined in §4.6, in the Boogie programming language [41] and used the Civl verifier [99] in order to check whether the assertions introduced by the instrumentation are violated (which would represent a robustness violation). Note that since clients are of fixed size, this requires no additional assertions/invariants (it is an instance of bounded model checking). We model tables as unbounded maps in Boogie and SQL queries as first-order formulas over these maps (that may contain existential or universal quantifiers). To model the uniqueness of primary keys we use Boogie linear types. The results are reported in Table 5.2. We have found two of the applications, Courseware and SmallBank, to *not* be robust against snapshot isolation. The violation in Courseware is caused by transactions RemoveCourse and EnrollStudent that execute concurrently, RemoveCourse removing a course that has no registered student and EnrollStudent registering a student to the same course. We get an invalid state where a student is registered for a course that was removed. SmallBank's violation contains transactions Balance, TransactSaving, and WriteCheck. One process executes WriteCheck where it withdraws an amount from the checking account after checking that the sum of the checking and savings accounts is bigger than this amount. Concurrently, a second process executes TransactSaving where it withdraws an amount from the saving account after checking that it is smaller than the amount in the savings account. Afterwards, the second process checks the contents of both the checking and saving accounts. We get an invalid state where the sum of the checking and savings accounts is negative.

Since in the first experiment we consider fixed clients, the lack of assertion violations does not imply that the application is robust (this instantiation of our reduction can only be used to reveal robustness violations). Thus, a second experiment concerns the robustness proof method based on commutativity dependency graphs (§4.7). For the applications that were not identified as non-robust by the previous method, we have used Civl to construct their commutativity dependency graphs, i.e., identify the "non-mover" relations M_{WR}, M_{WW}, and M_{RW} (Civl allows to check whether some code fragment is a left/right mover). In all cases, the graph didn't contain non-mover cycles, which allows to conclude that the applications are robust.

The experiments show that our results can be used for finding violations and proving robustness, and that they apply to a large set of interesting examples. Note that the reduction to SC and the proof method based on commutativity dependency graphs are valid for programs with SQL (select/update) queries.

4.9 Related Work

Similar to the previous chapter, in this chapter we borrow some high-level principles from [51] which addresses the robustness against TSO. We reuse the high-level methodology of characterizing minimal violations according to some measure and defining reductions to SC reachability using a program instrumentation. Instantiating this methodology in `SI` context is however very different, several fundamental differences being:

- `SI` and TSO admit different sets of relaxations and `SI` is a model of transactional databases.
- We use a different notion of measure: the measure in [51] counts the number of events between a write issue and a write commit while our notion of measure counts the number of delayed transactions. This is a first reason for which the proof techniques in [51] do not extend to our

Table 4.1: An overview of the analysis results. CDG stands for commutativity dependency graph. The columns PO and PT show the number of proof obligations and proof time in second, respectively. ⊤ stands for trivial when the application has only read-only transactions.

Application	#T	Robustness	Reachability Analysis		CDG Analysis	
			PO	PT	PO	PT
Auction	4	yes	70	0.3	20	0.5
Courseware	5	no	59	0.37	na	na
FusionTicket	4	yes	72	0.3	34	0.5
SmallBank	5	no	48	0.28	na	na
TPC-C	5	yes	54	0.7	82	3.7
Cassieq-Core	8	yes	173	0.55	104	2.9
Currency-Exchange	6	yes	88	0.35	26	3.5
PlayList	14	yes	99	4.63	236	7.3
RoomStore	5	yes	85	0.3	22	0.5
Shopping-Cart	4	yes	58	0.25	⊤	⊤

context.

– Transactions induce more complex traces: two transactions can be related by several dependency relations since each transaction may contain multiple reads and writes to different locations. In TSO, each action is a read or a write to some location, and two events are related by a single dependency relation. Also, the number of dependencies between two transactions depends on the execution since the set of reads/writes in a transaction evolves dynamically.

The existing work on the verification of robustness for transactional programs provide either over- or under-approximate analyses. Our commutativity dependency graphs are similar to the static dependency graphs used in [33, 47, 57, 58, 70], but they are more precise, i.e., reducing the number of false alarms. The static dependency graphs record happens-before dependencies between transactions based on a syntactic approximation of the variables accessed by a transaction. For example, our techniques are able to prove that the program in Figure

```
      p1:                          p2:
t1: [ if (x > y)          || t2: [ if (y > x)
        r1 = x - y                   r2 = y - x
        x  = y ]                     y  = x ]
```

Figure 4.7: A robust program.

4.7 is robust, while this is not possible using static dependency graphs. The latter would contain a dependency from transaction t_1 to t_2 and one from t_2 to t_1 just because syntactically, each of the two transactions reads both variables and may write to one of them. Our dependency graphs take into account the semantics of these transactions and do not include this happens-before cycle. Other over- and under-approximate analyses have been proposed in [129]. They are based on encoding executions into first order logic, bounded-model checking for the under-approximate analysis, and a sound check for proving a cut-off bound on the size of the happens-before cycles possible in the executions of a program, for the over-approximate analysis. The latter is less precise than our method based on commutativity dependency graphs. For instance, extending the TPC-C application with additional transactions will make the method in [129] fail while our method will succeed in proving robustness (the three transactions are for adding a new product, adding a new warehouse based on the number of customers and warehouses, and adding a new customer, respectively).

Finally, the idea of using Lipton's reduction theory for checking robustness has been also used in the context of the TSO memory model [54], but the techniques are completely different, e.g., the TSO technique considers each update in isolation and does not consider non-mover cycles like in our commutativity dependency graphs.

4.10 Conclusion

We studied the robustness against snapshot isolation relative to serializability. We proposed a characterization for a class of traces that are possible under snapshot isolation but are not possible under serializability, called *minimal violations*. Using this characterization, we showed that the robustness problem is polynomial time reducible to reachability under serializability (the size of the program increases linearly). We also used this characterization to develop an approximated method for proving robustness based on Lipton's reduction theory. We evaluated our techniques on a benchmark of distributed applications extracted from the literature and open source Github projects. The evaluation showed the effectiveness of our techniques for proving (non-)robustness of practical distributed applications.

Chapter 5

Robustness Between Weak Consistency Models

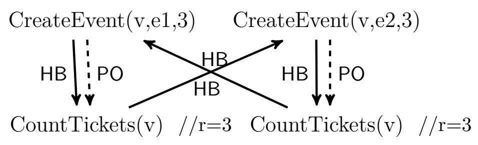
5.1 Introduction

In this chapter, we consider the sequence of increasingly strong consistency models, causal consistency (`CC`), prefix consistency (`PC`), and snapshot isolation (`SI`), and investigate the problem of checking robustness for a given program against weakening the consistency model to one in this range. In §5.2, we outline the robustness problems we study in this chapter: robustness against substituting `SI` with `PC` and `PC` with `CC`, respectively. Robustness against substituting `SI` with `PC` can be obtained as the conjunction of these two cases. In §5.3, we formally define programs traces under the above consistency models. In §5.4, we show that checking robustness against substituting `PC` with `CC` is reduced to the problem of checking robustness against substituting `SER` with `CC` that we studied in Chapter 3. In §5.5, we show that checking robustness for a program P is reduced to a reachability (assertion checking) problem in a composition of P under `PC` with a monitor that checks whether a `PC` behavior is an "anomaly", i.e., admitted by P under `PC`, but not under `SI`. In §5.7, we present a more pragmatic approach for establishing robustness, which avoids a non-reachability proof under `SER`, that builds on the concept of commutativity dependency graph introduced in Chapter 4. We give sufficient conditions for robustness in all the cases mentioned above, which characterize the commutativity dependency graph associated to a given program. Finally, in §5.8, we tested the applicability of the proposed techniques on a benchmark containing 7 challenging applications extracted from previous work [76, 105, 58]. These techniques are precise enough for proving or disproving the robustness of all of these applications, for all combinations of the consistency models discussed above.

```
      Process 1
CreateEvent(v, e1, 3):
[ Tickets[v][e1] := 3 ]
CountTickets(v):
[ r := $\sum_e$Tickets[v][e] ]
```

```
      Process 2
CreateEvent(v, e2, 3):
[ Tickets[v][e2] := 3 ]
CountTickets(v):
[ r := $\sum_e$Tickets[v][e] ]
```

(a) FusionTicket.

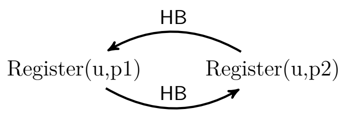


(b) A CC trace of FusionTicket.

```
      Process 1
Register(u, p1):
[ r := RegisteredUsers[u]
  assume r == 0
  RegisteredUsers[u] := 1
  Password[u] := p1 ]
```

```
      Process 2
Register(u, p2):
[ r := RegisteredUsers[u]
  assume r == 0
  RegisteredUsers[u] := 1
  Password[u] := p2 ]
```

(c) Twitter.

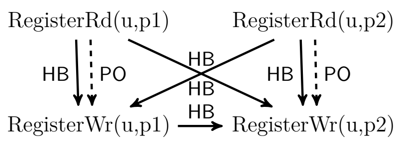


(d) A CC and PC trace of Twitter.

```
      Process 1
RegisterRd(u, p1):
[ r := RegisteredUsers[u]
  assume r == 0 ]
RegisterWr(u, p1):
[ RegisteredUsers[u] := 1
  Password[u] := p1 ]
```

```
      Process 2
RegisterRd(u, p2):
[ r := RegisteredUsers[u]
  assume r == 0 ]
RegisterWr(u, p2):
[ RegisteredUsers[u] := 1
  Password[u] := p2 ]
```

(e) Transformed Twitter.



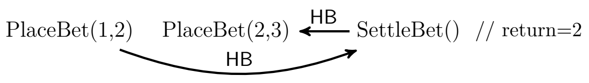
(f) A CC and SER trace of transformed Twitter.

```
      Process 1
PlaceBet(1,2):
[ assume time < EXPIRY_time
    Bets[1] := 2 ]
```

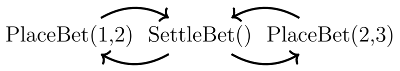
```
      Process 2
PlaceBet(2,3):
[ assume time < EXPIRY_time
    Bets[2] := 3 ]
```

```
      Process 3
SettleBet():
[ Bets' := Bets
  n := Bets'.Length
  assume time > EXPIRY_time & n > 0
  select i s.t. Bets'[i] $\neq \perp$
  return := Bets'[i] ]
```

(g) Betting.



(h) A PC and SI trace of Betting.



(i) The commutativity dependency graph of Betting.

Figure 5.1: Programs and traces under different consistency models.

5.2 Overview

We give an overview of the robustness problems we investigate in this chapter, discussing first the case `PC` vs. `CC`, and then `SI` vs `PC`. We end with an example that illustrates the more pragmatic robustness checking technique based on commutativity arguments.

Robustness `PC` vs `CC`. We illustrate the robustness against substituting `PC` with `CC` using the FusionTicket and the Twitter programs in Figure 5.1a and Figure 5.1c, respectively. FusionTicket manages tickets for a number of events, each event being associated with a venue. Its state consists of a two-dimensional map that stores the number of tickets for an event in a given venue (r is a local variable, and the assignment in `CountTickets` is interpreted as a read of the shared state). The program has two processes and each process contains two transactions. The first transaction creates an event e in a venue v with a number of tickets n, and the second transaction computes the total number of tickets for all the events in a venue v. A possible candidate for a specification of this program is that the values computed in `CountTickets` are monotonically increasing since each such value is computed after creating a new event. Twitter provides a transaction for registering a new user with a given username and password, which is executed by two parallel processes. Its state contains two maps that record whether a given username has been registered (0 and 1 stand for non-registered and registered, respectively) and the password for a given username. Each transaction first checks whether a given username is free (see the `assume` statement). The intended specification is that the user must be registered with the given password when the registration transaction succeeds.

A program is robust against substituting `PC` with `CC` if its set of behaviors under the two models coincide. We model behaviors of a given program as *traces*, which record standard control-flow and data-flow dependencies between transactions, e.g., the order between transactions in the same session and whether a transaction reads the value written by another (read-from). The transitive closure of the union of all these dependency relations is called *happens-before*. Figure 5.1b pictures a trace of FusionTicket where the concrete values which are read in a transaction are written under comments. In this trace, each process registers a different event but in the same venue and with the same number of tickets, and it ignores the event created by the other process when computing the sum of tickets in the venue.

Figure 5.1b pictures a trace of FusionTicket under `CC`, which is a witness that FusionTicket is *not* robust against substituting `PC` with `CC`. This trace is also a violation of the intended specification since the number of tickets is not increasing (the sum of tickets is 3 in both processes). The happens-before dependencies (pictured with HB labeled edges) include the program-order PO (the order between transactions in the same process), and read-write dependencies, since an instance of CountTickets(v) does not observe the value written by the CreateEvent transaction in the other process (the latter overwrites some value that the former reads). This trace is allowed under CC because the transaction CreateEvent(v, e1, 3) executes concurrently with the transaction CountTickets(v) in the other process, and similarly for CreateEvent(v, e2, 3). However, it is not allowed under PC since it is impossible to define a total commit order between CreateEvent(v, e1, 3) and CreateEvent(v, e2, 3) that justifies the reads of both CountTickets(v) transactions (these reads should correspond to the updates in a prefix of this order). For instance, if we assume that CreateEvent(v, e1, 3) commits before CreateEvent(v, e2, 3), then CountTickets(v) in the second process must observe the effect of CreateEvent(v, e1, 3) as well since it observes the effect of CreateEvent(v, e2, 3). However, this contradicts the fact that CountTickets(v) computes the sum of tickets as being 3.

On the other hand, Twitter is robust against substituting PC with CC. For instance, Figure 5.1d pictures a trace of Twitter under CC, where the `assume` in both transactions pass. In this trace, the transactions Register(u,p1) and Register(u,p2) execute concurrently and are unaware of each other's writes (they are not causally related). The HB dependencies include write-write dependencies since both transactions write on the same location (we consider the transaction in Process 2 to be the last one writing to the `Password` map), and read-write dependencies since each transaction reads `RegisteredUsers` that is written by the other. This trace is also allowed under PC since the commit order can be defined such that Register(u,p1) is ordered before Register(u,p2), and then both transactions read from the initial state (the empty prefix). Note that this trace has a cyclic happens-before which means that it is not allowed under serializability.

Checking robustness PC vs CC. We reduce the problem of checking robustness against substituting PC with CC to the robustness problem against substituting SER with CC (the latter reduces to a reachability problem under SER [45]). This reduction relies on a syntactic program transformation that rewrites PC behaviors of a given program P to SER behaviors of another program P'. The program P' is obtained by splitting each transaction t of P into two transactions: the first transaction performs all the reads in t and the second performs all the writes in t (the two are related by program order). Figure 5.1e shows this transformation applied on Twitter. The trace in Figure 5.1f is a serializable execution of the transformed Twitter which is "observationally" equivalent to the trace in Figure 5.1d of the original Twitter, i.e., each read of the shared state returns the same value and the writes on the shared state are applied in the same order (the acyclicity of the happens-before shows that this is a serializable trace). The transformed FusionTicket coincides with the original version because it contains no transaction that both reads and writes on the shared state.

We show that PC behaviors and SER behaviors of the original and transformed program, respectively, are related by a bijection. In particular, we show that any PC vs. CC robustness violation of the original program manifests as a SER vs. CC robustness violation of the transformed program, and vice-versa. For instance, the CC trace of the original Twitter in Figure 5.1d corresponds to the CC trace of the transformed Twitter in Figure 5.1f, and the acyclicity of the latter (the fact that it is admitted by SER) implies that the former is admitted by the original Twitter under PC. On the other hand, the trace in Figure 5.1b is also a CC of the transformed FusionTicket and its cyclicity implies that it is not admitted by FusionTicket under PC, and thus, it represents a robustness violation.

Robustness SI vs PC. We illustrate the robustness against substituting SI with PC using Twitter and the Betting program in Figure 5.1g. Twitter is *not* robust against substituting SI with PC, the trace in Figure 5.1d being a witness violation. This trace is also a violation of the intended specification since one of the users registers a password that is overwritten in a concurrent transaction. This PC trace is not possible under SI because Register(u,p1) and Register(u,p2) observe the same prefix of the commit order (i.e., an empty prefix), but they write to a common memory location Password[u] which is not allowed under SI.

On the other hand, the Betting program in Figure 5.1g, which manages a set of bets, is robust against substituting SI with PC. The first two processes execute one transaction that places a bet of a value v with a unique bet identifier id, assuming that the bet expiration time is not yet reached (bets are recorded in the map `Bets`). The third process contains a single transaction that settles the betting assuming that the bet expiration time was reached and at least one bet has been placed. This transaction starts by taking a snapshot of the `Bets` map into a local variable `Bets'`, and then selects a random non-null value (different from $\perp$) in the map to correspond to the winning bet. The intended specification of this program is that the winning bet corresponds to a genuine bet that was placed. Figure 5.1g pictures a PC trace of Betting where SettleBet observes only the bet of the first process PlaceBet(1,2). The HB dependency towards the second process denotes a read-write dependency (SettleBet reads a cell of the map `Bets` which is overwritten by the second process).

This trace is allowed under SI because no two transactions write to the same location.

Checking robustness SI vs PC. We reduce robustness against substituting SI with PC to a reachability problem under SER. This reduction is based on a characterization of happens-before cycles[1] that are possible under PC but not SI, and the transformation described above that allows to simulate the PC semantics of a program on top of SER. The former is used to define an instrumentation (monitor) for the transformed program that reaches an error state iff the original program is not robust. Therefore, we show that the happens-before cycles in PC traces that are not admitted by SI must contain a transaction that (1) overwrites a value written by another transaction in the cycle and (2) reads a value overwritten by another transaction in the cycle. For instance, the trace of Twitter in Figure 5.1d is not allowed under SI because Register(u,p2) overwrites a value written by Register(u,p1) (the password) and reads a value overwritten by Register(u,p1) (checking whether the username u is registered). The trace of Betting in Figure 5.1g is allowed under SI because its happens-before is acyclic.

Checking robustness using commutativity arguments. Based on the reductions above, we propose an approximated method for proving robustness based on the concept of mover in Lipton's reduction theory [120]. A transaction is a left (resp., right) mover if it commutes to the left (resp., right) of another transaction (by a different process) while preserving the computation. We use the notion of mover to characterize the data-flow dependencies in the happens-before. Roughly, there exists a data-flow dependency between two transactions in some execution if one does not commute to the left/right of the other one.

We define a commutativity dependency graph which summarizes the happens-before dependencies in all executions of a transformed program (obtained by splitting the transactions of the original program as explained above), and derive a proof method for robustness which inspects paths in this graph. Two transactions t_1 and t_2 are linked by a directed edge iff t_1 *cannot* move to the right of t_2 (or t_2 cannot move to the left of t_1), or if they are related by the program order. Moreover, two transactions t_1 and t_2 are linked by an undirected edge iff they are the result of splitting the same transaction.

A program is robust against substituting PC with CC if roughly, its commutativity dependency graph does *not* contain a *simple* cycle of directed edges with two distinct transactions t_1 and t_2,

[1]Traces with an acyclic happens-before are not robustness violations because they are admitted under serializability, which implies that they are admitted under the weaker model SI as well.

such that t_1 does not commute left because of another transaction t_3 in the cycle that reads a variable that t_1 writes to, and t_2 does not commute right because of another transaction t_4 in the cycle (t_3 and t_4 can coincide) that writes to a variable that t_2 either reads from or writes to[2]. For instance, Figure 5.1i shows the commutativity dependency graph of the transformed Betting program, which coincides with the original Betting because PlaceBet(1,2) and PlaceBet(2,3) are write-only transactions and SettleBet() is a read-only transaction. Both simple cycles in Figure 5.1i contain just two transactions and therefore do not meet the criterion above which requires at least 3 transactions. Therefore, Betting is robust against substituting `PC` with `CC`.

A program is robust against substituting `SI` with `PC`, if roughly, its commutativity dependency graph does *not* contain a *simple* cycle with two successive transactions t_1 and t_2 that are linked by an undirected edge, such that t_1 does not commute left because of another transaction t_3 in the cycle that writes to a variable that t_1 writes to, and t_2 does not commute right because of another transaction t_4 in the cycle (t_3 and t_4 can coincide) that writes to a variable that t_2 reads from[3]. Betting is also robust against substituting `SI` with `PC` for the same reason (simple cycles of size 2).

5.3 Consistency Models

Syntax. We assume w.l.o.g. that every transaction is written as a sequence of reads or `assume` statements followed by a sequence of writes (a single `goto` statement from the sequence of read/`assume` instructions transfers the control to the sequence of writes). In the context of the consistency models we study in this chapter, every program can be equivalently rewritten as a set of transactions of this form.

Semantics. We consider the serializability (`SER`), causal consistency (we focus on causal convergence) (`CC`), and snapshot isolation (`SI`) consistency models semantics as described in Chapters 2, 3, and 4, respectively.

In the semantics of a program under prefix consistency (`PC`), shared variables are stored in a central memory and each process keeps a local valuation of these variables. When a process starts a new transaction, it fetches a consistent snapshot of the shared variables from the central memory and stores it in its local valuation of these variables. During the execution of a transaction in a process, writes to shared variables are stored in the local valuation of these variables, and in a

[2]The transactions t_1, t_2, t_3, and t_4 correspond to t_1, t_i, t_n, and t_{i+1}, respectively, in Theorem 5.7.

[3]The transactions t_1, t_2, t_3, and t_4 correspond to t_1, t_2, t_n, and t_3, respectively, in Theorem 5.8.

transaction log. To read a shared variable, a process takes its own valuation of the shared variable. A process commits a transaction by applying the updates in the transaction log on the central memory in an atomic way (to make them visible to all processes).

We use the standard model of executions of a program, i.e., *trace*. We assume that each transaction in a program is identified uniquely using a transaction identifier from a set $\mathbb{T}$. $f : \mathbb{T} \to 2^{\mathbb{S}}$ denotes a mapping that associates each transaction in $\mathbb{T}$ with a sequence of read and write events from the set

$$\mathbb{S} = \{\mathsf{re}(t, x, v), \mathsf{we}(t, x, v) : t \in \mathbb{T}, x \in \mathbb{V}, v \in \mathbb{D}\}$$

where $\mathsf{re}(t, x, v)$ is a read of x returning v, and $\mathsf{we}(t, x, v)$ is a write of v to x.

Definition 5.1. *A* trace *is a tuple* $tr = (\rho, f, \mathsf{TO}, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$ *where* $\rho \subseteq \mathbb{T}$ *is a set of transaction identifiers, and*

- TO *is a mapping giving the order between events in each transaction, i.e., it associates each transaction* t *in* ρ *with a total order* $\mathsf{TO}(t)$ *on* $f(t) \times f(t)$.

- PO *is the program order relation, a strict partial order on* $\rho \times \rho$ *that orders every two transactions issued by the same process.*

- WR *is the read-from relation between distinct transactions* $(t1, t2) \in \rho \times \rho$ *representing the fact that* $t2$ *reads a value written by* $t1$.

- WW *is the store order relation on* $\rho \times \rho$ *between distinct transactions that write to the same shared variable.*

- RW *is the conflict order relation between distinct transactions, defined by* $\mathsf{RW} = \mathsf{WR}^{-1}; \mathsf{WW}$ *(*$;$ *denotes the sequential composition of two relations).*

For simplicity, for a trace $tr = (\rho, f, \mathsf{TO}, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, we write $t \in tr$ instead of $t \in \rho$. We also assume that each trace contains a fictitious transaction that writes the initial values of all shared variables, and which is ordered before any other transaction in program order. Also, $\mathbb{T}\mathsf{r}_{\mathsf{X}}(\mathcal{P})$ is the set of traces representing executions of program $\mathcal{P}$ under a consistency model X.

For each $\mathsf{X} \in \{\mathtt{CC}, \mathtt{PC}, \mathtt{SI}, \mathtt{SER}\}$, the set of traces $\mathbb{T}\mathsf{r}_{\mathsf{X}}(\mathcal{P})$ can be described using the set of properties in Table 5.1. A trace tr is possible under causal consistency iff there exist two relations CO a partial order (causal order) and ARB a total order (arbitration order) that includes CO, such that the properties $\mathsf{AxCausal}$, AxArb, and $\mathsf{AxRetVal}$ hold [71, 52]. $\mathsf{AxCausal}$ guarantees that the program order and the read-from relation are included in the causal order, and AxArb guarantees that the causal order and the store order are included in the arbitration order. $\mathsf{AxRetVal}$ guarantees that a read returns the value written by the last write in the last transaction that contains a write to the same variable and that is ordered by CO before the read's transaction. We use AxCC to denote the conjunction of these three properties. A trace tr is possible under prefix consistency iff there exist a causal order CO and an arbitration order ARB such that AxCC holds and the property $\mathsf{AxPrefix}$ holds as well [71]. $\mathsf{AxPrefix}$ guarantees that every transaction observes a prefix of transactions that are ordered by ARB before it. We use AxPC to denote the conjunction of AxCC and $\mathsf{AxPrefix}$. A trace tr is possible under snapshot isolation iff there exist a causal order CO and an arbitration order ARB such that AxPC holds and the property $\mathsf{AxConflict}$ holds [71]. $\mathsf{AxConflict}$ guarantees that if two transactions write to the same variable then one of them must observe the other. We use AxSI to denote the conjunction of AxPC and $\mathsf{AxConflict}$. A trace tr is serializable iff there exist a causal order CO and an arbitration order ARB such that the property AxSer holds which implies that the two relations CO and ARB coincide. Note that for any given program $\mathcal{P}$, $\mathbb{T}\mathsf{r}_{\mathtt{SER}}(\mathcal{P}) \subseteq \mathbb{T}\mathsf{r}_{\mathtt{SI}}(\mathcal{P}) \subseteq \mathbb{T}\mathsf{r}_{\mathtt{PC}}(\mathcal{P}) \subseteq \mathbb{T}\mathsf{r}_{\mathtt{CC}}(\mathcal{P})$.

For a given trace $tr = (\rho, f, \mathsf{TO}, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, the happens before order is the transitive closure of the union of all the relations in the trace, i.e., $\mathsf{HB} = (\mathsf{PO} \cup \mathsf{WR} \cup \mathsf{WW} \cup \mathsf{RW})^{+}$. A classic result states that a trace tr is serializable iff HB is acyclic [25, 156]. Note that HB is acyclic implies that WW is a total order between transactions that write to the same variable, and $(\mathsf{PO} \cup \mathsf{WR})^{+}$ and $(\mathsf{PO} \cup \mathsf{WR} \cup \mathsf{WW})^{+}$ are acyclic.

5.3.1 Robustness

In this chapter, we investigate the problem of checking whether a program $\mathcal{P}$ under a semantics $\mathsf{Y} \in \{\mathtt{PC},\ \mathtt{SI}\}$ produces the same set of traces as under a weaker semantics $\mathsf{X} \in \{\mathtt{CC},\ \mathtt{PC}\}$.

We illustrate the notion of robustness on the programs in Figure 5.2, which are commonly used in the literature. In all programs, transactions of the same process are aligned vertically and ordered from top to bottom. Each read instruction is commented with the value it reads in some execution.

The store buffering (SB) program in Figure 5.2a contains four transactions that are issued by two distinct processes. We emphasize an execution where t_2 reads 0 from y and t_4 reads 0 from x. This execution is allowed under $\mathtt{CC}$ since the two writes by t_1 and t_3 are not causally dependent.

AxCausal	$\mathsf{CO}_0^+ \subseteq \mathsf{CO}$
AxArb	$ARB_0^+ \subseteq ARB$
AxCC	AxRetVal $\wedge$ AxCausal $\wedge$ AxArb
AxPrefix	$ARB; \mathsf{CO} \subseteq \mathsf{CO}$
AxPC	AxPrefix $\wedge$ AxCC
AxConflict	$\mathsf{WW} \subseteq \mathsf{CO}$
AxSI	AxConflict $\wedge$ AxPC
AxSer	AxRetVal $\wedge$ AxCausal $\wedge$ AxArb $\wedge$ $\mathsf{CO} = ARB$

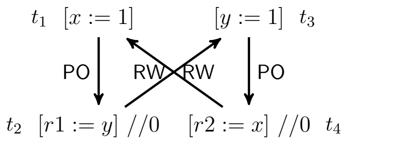
where

$\mathsf{CO}_0 = \mathsf{PO} \cup \mathsf{WR}$ and $ARB_0 = \mathsf{PO} \cup \mathsf{WR} \cup \mathsf{WW}$

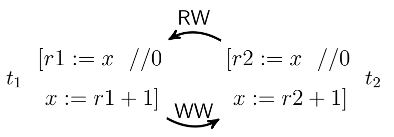
AxRetVal $= \forall\, t \in tr.\ \forall\, \mathsf{re}(t, x, v) \in f(t)$ we have that

- there exist a transaction $t_0 = Max_{ARB}(\{t' \in tr \mid (t', t) \in \mathsf{CO} \wedge \exists\, \mathsf{we}(t', x, \cdot) \in f(t')\})$ and an event $\mathsf{we}(t_0, x, v) = Max_{\mathsf{TO}(t_0)}(\{\mathsf{we}(t_0, x, \cdot) \in f(t_0)\})$.

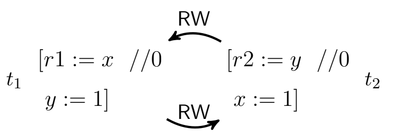
Table 5.1: Declarative definitions of consistency models. For an order relation $\leq$, $a = Max_{\leq}(A)$ iff $a \in A \wedge \forall\, b \in A.\ b \leq a$.



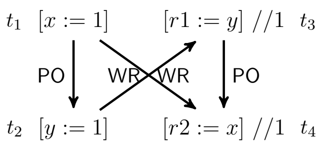
(a) Store Buffering (SB).



(b) Lost Update (LU).



(c) Write Skew (WS).



(d) Message Passing (MP).

Figure 5.2: Litmus programs

Thus, t_2 and t_4 are executed without seeing the writes from t_3 and t_1, respectively. However, this execution is not feasible under `PC` (which implies that it is not feasible under both `SI` and `SER`). In particular, we cannot have neither $(t_1, t_3) \in ARB$ nor $(t_3, t_1) \in ARB$ which contradicts the fact that ARB is total order. For example, if $(t_1, t_3) \in ARB$, then $(t_1, t_4) \in \mathsf{CO}$ (since $ARB; \mathsf{CO} \subset \mathsf{CO}$) which contradicts the fact that t_4 does not see t_1. Similarly, $(t_3, t_1) \in ARB$ implies that $(t_3, t_2) \in \mathsf{CO}$ which contradicts the fact that t_2 does not see t_3. Thus, SB is not robust against `CC` relative to `PC`.

The lost update (LU) program in Figure 5.2b has two transactions that are issued by two distinct processes. We highlight an execution where both transactions read 0 from x. This execution is allowed under `PC` since both transactions are not causally dependent and can be executed in parallel by the two processes. However, it is not allowed under `SI` since both transactions write to a common variable (i.e., x). Thus, they cannot be executed in parallel and one of them must see the write of the other. Thus, LU is not robust against `PC` relative to `SI`.

The write skew (WS) program in Figure 5.2c has two transactions that are issued by two distinct processes. We highlight an execution where t_1 reads 0 from x and t_2 reads 0 from y. This execution is allowed under `SI` since both transactions are not causally dependent, do not write to a common variable, and can be executed in parallel by the two processes. However, this execution is not allowed under `SER` since one of the two transactions must see the write of the other. Thus, WS is not robust against `SI` relative to `SER`.

The message passing (MP) program in Figure 5.2d has four transactions issued by two processes. Because t_1 and t_2 are causally dependent, under any semantics $\mathsf{X} \in \{$`CC`, `PC`, `SI`, `SER`$\}$ we only have three possible executions of MP, which correspond to either t_3 and t_4 not observing the writes of t_1 and t_2, or t_3 and t_4 observe the writes of both t_1 and t_2, or t_4 observes the write of t_1 (we highlight the values read in the second case in Figure 5.2d). Therefore, the executions of this program under the four consistency models coincide. Thus, MP is robust against `CC` relative to any other model.

5.4 Robustness Against `CC` Relative to `PC`

We show that checking robustness against `CC` relative to `PC` can be reduced to checking robustness against `CC` relative to `SER`. The crux of this reduction is a program transformation that allows to simulate the `PC` semantics of a program $\mathcal{P}$ using the `SER` semantics of a program $\mathcal{P}_\clubsuit$. Checking robustness against `CC` relative to `SER` can be reduced in polynomial time to reachability under `SER` [45].

Given a program $\mathcal{P}$ with a set of transactions $\mathsf{Tr}(\mathcal{P})$, we define a program $\mathcal{P}_\clubsuit$ such that every transaction $t \in \mathsf{Tr}(\mathcal{P})$ is split into a transaction $t[r]$ that contains all the read/`assume` statements in t (in the same order) and another transaction $t[w]$ that contains all the write statements in t (in the same order). In the following, we establish the following result:

Theorem 5.1. *A program $\mathcal{P}$ is robust against* `CC` *relative to* `PC` *iff $\mathcal{P}_\clubsuit$ is robust against* `CC` *relative to* `SER`*.*

The proof of this theorem relies on several intermediate results concerning the relationship between traces of $\mathcal{P}$ and $\mathcal{P}_{\clubsuit}$. Let $tr = (\rho, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathbb{T}\mathsf{r}_{\mathsf{X}}(\mathcal{P})$ be a trace of a program $\mathcal{P}$ under a semantics X. We define the trace $tr_{\clubsuit} = (\rho_{\clubsuit}, \mathsf{PO}_{\clubsuit}, \mathsf{WR}_{\clubsuit}, \mathsf{WW}_{\clubsuit}, \mathsf{RW}_{\clubsuit})$ where every transaction $t \in tr$ is split into two transactions $t[r] \in tr_{\clubsuit}$ and $t[w] \in tr_{\clubsuit}$, and the dependency relations are straightforward adaptations, i.e.,

- $\mathsf{PO}_{\clubsuit}$ is the smallest transitive relation that includes $(t[r], t[w])$ for every t, and $(t[w], t'[r])$ if $(t, t') \in \mathsf{PO}$,

- $(t'[w], t[r]) \in \mathsf{WR}_{\clubsuit}$, $(t'[w], t[w]) \in \mathsf{WW}_{\clubsuit}$, and $(t'[r], t[w]) \in \mathsf{RW}_{\clubsuit}$ if $(t', t) \in \mathsf{WR}$, $(t', t) \in \mathsf{WW}$, and $(t', t) \in \mathsf{RW}$, respectively.

For instance, Figure 5.3 pictures the trace $tr_{\clubsuit}$ for the LU trace tr given in Figure 5.2b. For traces tr of programs that contain singleton transactions, e.g., SB in Figure 5.2a, $tr_{\clubsuit}$ coincides with tr.

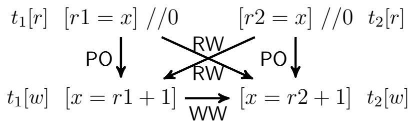


Figure 5.3: A trace of the transformed LU program ($\mathsf{LU}_{\clubsuit}$).

Conversely, for a given trace $tr_{\clubsuit} = (\rho_{\clubsuit}, \mathsf{PO}_{\clubsuit}, \mathsf{WR}_{\clubsuit}, \mathsf{WW}_{\clubsuit}, \mathsf{RW}_{\clubsuit}) \in \mathbb{T}\mathsf{r}_{\mathsf{X}}(\mathcal{P}_{\clubsuit})$ of a program $\mathcal{P}_{\clubsuit}$ under a semantics X, we define the trace $tr = (\rho, \mathsf{PO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$ where every two components $t[r]$ and $t[w]$ are merged into a transaction $t \in tr$. The dependency relations are defined in a straightforward way, e.g., if $(t'[w], t[w]) \in \mathsf{WW}_{\clubsuit}$ then $(t', t) \in \mathsf{WW}$.

The following lemma shows that for any semantics $\mathsf{X} \in \{\mathtt{CC}, \mathtt{PC}, \mathtt{SI}\}$, if $tr \in \mathbb{T}\mathsf{r}_{\mathsf{X}}(\mathcal{P})$ for a program $\mathcal{P}$, then $tr_{\clubsuit}$ is a valid trace of $\mathcal{P}_{\clubsuit}$ under X, i.e., $tr_{\clubsuit} \in \mathbb{T}\mathsf{r}_{\mathsf{X}}(\mathcal{P}_{\clubsuit})$. Intuitively, this lemma shows that splitting transactions in a trace and defining dependency relations appropriately cannot introduce cycles in these relations and preserves the validity of the different consistency axioms. The proof of this lemma relies on constructing a causal order $\mathsf{CO}_{\clubsuit}$ and an arbitration order $ARB_{\clubsuit}$ for the trace $tr_{\clubsuit}$ starting from the analogous relations in tr. In the case of $\mathtt{CC}$, these are the smallest transitive relations such that:

- $\mathsf{PO}_{\clubsuit} \subseteq \mathsf{CO}_{\clubsuit} \subseteq ARB_{\clubsuit}$, and

- if $(t_1, t_2) \in \mathsf{CO}$ then $(t_1[w], t_2[r]) \in \mathsf{CO}_{\clubsuit}$, and if $(t_1, t_2) \in ARB$ then $(t_1[w], t_2[r]) \in ARB_{\clubsuit}$.

For $\mathtt{PC}$ and $\mathtt{SI}$, $\mathsf{CO}_{\clubsuit}$ must additionally satisfy: if $(t_1, t_2) \in ARB$, then $(t_1[w], t_2[w]) \in \mathsf{CO}_{\clubsuit}$. This is required in order to satisfy the axiom $\mathsf{AxPrefix}$, i.e., $ARB_{\clubsuit}; \mathsf{CO}_{\clubsuit} \subset \mathsf{CO}_{\clubsuit}$, when $(t_1[w], t_2[r]) \in ARB_{\clubsuit}$ and $(t_2[r], t_2[w]) \in \mathsf{CO}_{\clubsuit}$.

This construction ensures that $\mathsf{CO}_{\clubsuit}$ is a partial order and $ARB_{\clubsuit}$ is a total order because CO is a partial order and ARB is a total order. Also, based on the above rules, we have that: if

$(t_1[w], t_2[r]) \in \mathsf{CO}_{\clubsuit}$ then $(t_1, t_2) \in \mathsf{CO}$, and similarly, if $(t_1[w], t_2[r]) \in \mathit{ARB}_{\clubsuit}$ then $(t_1, t_2) \in \mathit{ARB}$.

Lemma 5.1. *If $tr \in \mathbb{T}r_{\mathsf{X}}(\mathcal{P})$, then $tr_{\clubsuit} \in \mathbb{T}r_{\mathsf{X}}(\mathcal{P}_{\clubsuit})$.*

Proof. We start with the case $\mathsf{X} = \mathtt{CC}$. We first show that $tr_{\clubsuit}$ satisfies $\mathsf{AxCausal}$ and AxArb. For $\mathsf{AxCausal}$, let $t'_1 \in \{t_1[r], t_1[w]\}$ and $t'_2 \in \{t_2[r], t_2[w]\}$, such that $(t'_1, t'_2) \in (\mathsf{PO}_{\clubsuit} \cup \mathsf{WR}_{\clubsuit})^+$. By the definition of $\mathsf{CO}_{\clubsuit}$, we have that either $(t'_1 = t_1[r], t'_2 = t_2[w]) \in \mathsf{PO}_{\clubsuit}$ and $t_1 = t_2$ or $(t_1, t_2) \in (\mathsf{PO} \cup \mathsf{WR})^+$, which implies that $(t_1, t_2) \in \mathsf{CO}$. In both cases we obtain that $(t'_1, t'_2) \in \mathsf{CO}_{\clubsuit}$. The axiom $\mathsf{AxCausal}$ can be proved in a similar way.

Next, we show that $tr_{\clubsuit}$ satisfies the property $\mathsf{AxRetVal}$. Let t be a transaction in tr that contains a read event $\mathsf{re}(t, x, v)$. Let t_0 be the transaction in tr such that

$$t_0 = \mathit{Max}_{\mathit{ARB}}(\{t' \in tr \mid (t', t) \in \mathsf{CO} \wedge \exists\ \mathsf{we}(t', x, \cdot) \in f(t')\}).$$

The read value v must have been written by t_0 since tr satisfies $\mathsf{AxRetVal}$. Thus, the read $\mathsf{re}(t, x, v)$ in $t[r]$ of $tr_{\clubsuit}$ must return the value written by $t_0[w]$. From the definitions of $\mathsf{CO}_{\clubsuit}$ and $\mathit{ARB}_{\clubsuit}$, we get

$$t_1[w] \in \{t'[w] \in tr_{\clubsuit} \mid (t'[w], t[r]) \in \mathsf{CO}_{\clubsuit} \wedge \exists\ \mathsf{we}(t'[w], x, \cdot) \in f(t'[w])\}$$

iff

$$t_1 \in \{t' \in tr \mid (t', t) \in \mathsf{CO} \wedge \exists\ \mathsf{we}(t', x, \cdot) \in f(t')\}$$

because $(t_1[w], t_2[r]) \in \mathsf{CO}_{\clubsuit}$ implies $(t_1, t_2) \in \mathsf{CO}$. Since $(t_1[w], t_2[w]) \in \mathit{ARB}_{\clubsuit}$ implies $(t_1, t_2) \in \mathit{ARB}$, we also obtain that

$$t_0[w] = \mathit{Max}_{\mathit{ARB}_{\clubsuit}}(\{t'[w] \in tr_{\clubsuit} \mid (t'[w], t[r]) \in \mathsf{CO}_{\clubsuit} \wedge \exists\ \mathsf{we}(t'[w], x, \cdot) \in f(t'[w])\})$$

and since the read $\mathsf{re}(t, x, v)$ in $t[r]$ of $tr_{\clubsuit}$ returns the value written by $t_0[w]$, $tr_{\clubsuit}$ satisfies $\mathsf{AxRetVal}$.

For the case $\mathsf{X} = \mathtt{PC}$, we show that $tr_{\clubsuit}$ satisfies the property $\mathsf{AxPrefix}$ (the other axioms are proved as in the case of $\mathtt{CC}$). Suppose we have $(t'_1, t'_2) \in \mathit{ARB}_{\clubsuit}$ and $(t'_2, t'_3) \in \mathsf{CO}_{\clubsuit}$ where $t'_1 \in \{t_1[r], t_1[w]\}$, $t'_2 \in \{t_2[r], t_2[w]\}$, and $t'_3 \in \{t_3[r], t_3[w]\}$. The are five cases to be discussed:

1. $(t'_1 = t_1[r], t'_2 = t_2[w]) \in \mathsf{PO}_{\clubsuit}$ and $t_1 = t_2$ and $(t_2, t_3) \in \mathsf{CO}$,

2. $(t_1, t_2) \in \mathit{ARB}$ and $(t_2, t_3) \in \mathsf{CO}$,

3. $(t_1, t_2) \in \mathit{ARB}$ and $(t'_2 = t_2[r], t'_3 = t_3[w]) \in \mathsf{PO}_{\clubsuit}$ and $t_2 = t_3$,

4. $(t'_1 = t_1[r], t'_2 = t_2[w]) \in \mathsf{PO}_{\clubsuit}$ and $t_1 = t_2$ and $(t_2, t_3) \in \mathit{ARB}$ and $t'_3 = t_3[w]$,

5. $(t_1, t_2) \in ARB$ and $(t_2, t_3) \in ARB$ and $t'_3 = t_3[w]$.

Cases (a) and (b) imply that $(t_1, t_3) \in \mathsf{CO}$ since $ARB; \mathsf{CO} \subset ARB$, which implies that $(t'_1, t'_3) \in \mathsf{CO}_\clubsuit$. Cases (c), (d), and (e) imply that $(t_1, t_3) \in ARB$ and $t'_3 = t_3[w]$ then we get that $(t_1[w], t_3[w]) \in \mathsf{CO}_\clubsuit$ and $t'_3 = t_3[w]$ which means that $(t'_1, t'_3) \in \mathsf{CO}_\clubsuit$.

Note that the rule $(t_1[w], t_2[w]) \in \mathsf{CO}_\clubsuit$ if $(t_1, t_2) \in ARB$ cannot change the fact that

$$t_1[w] \in \{t'[w] \in tr_\clubsuit \mid (t'[w], t[r]) \in \mathsf{CO}_\clubsuit \wedge \exists\ \mathsf{we}(t'[w], x, \cdot) \in f(t'[w])\}$$

iff

$$t_1 \in \{t' \in tr \mid (t', t) \in \mathsf{CO} \wedge \exists\ \mathsf{we}(t', x, \cdot) \in f(t')\}$$

Thus, the proof of $\mathsf{AxRetVal}$ follows as in the previous case.

For the case $\mathsf{X} = \mathtt{SI}$, we show that $tr_\clubsuit$ satisfies $\mathsf{AxConflict}$. If $(t_1[w], t_2[w]) \in \mathsf{WW}_\clubsuit$, then $(t_1, t_2) \in \mathsf{WW} \subset \mathsf{CO}$, which implies that $(t_1[w], t_2[r]) \in \mathsf{CO}_\clubsuit$. Therefore, $(t_1[w], t_2[w]) \in \mathsf{CO}_\clubsuit$, which concludes the proof. The axiom $\mathsf{AxRetVal}$ can be proved as in the previous cases. $\square$

Before presenting a strengthening of Lemma 5.1 when X is $\mathtt{CC}$, we give an important characterization of $\mathtt{CC}$ traces. This characterization is stated in terms of acyclicity properties.

Lemma 5.2. *tr is a trace under $\mathtt{CC}$ iff ARB_0^+ and $\mathsf{CO}_0^+; \mathsf{RW}$ are acyclic (ARB_0 and CO_0 are defined in Table 5.1).*

Proof. ($\Rightarrow$) Let tr be a trace under $\mathtt{CC}$. From $\mathsf{AxCausal}$ and AxArb we get that $ARB_0^+ \subset ARB$, and ARB_0^+ is acyclic because ARB is total order. Assume by contradiction that $\mathsf{CO}_0^+; \mathsf{RW}$ is cyclic which implies that $\mathsf{CO}; \mathsf{RW}$ is cyclic since $\mathsf{CO}_0^+ \subset \mathsf{CO}$, which means that there exist t_1 and t_2 such that $(t_1, t_2) \in \mathsf{CO}$ and $(t_2, t_1) \in \mathsf{RW}$. $(t_2, t_1) \in \mathsf{RW}$ implies that there exists t_3 such that $(t_3, t_1) \in \mathsf{WW}$ and $(t_3, t_2) \in \mathsf{WR}$. Based on the definition of $\mathsf{AxRetVal}$, t_3 has two possible instances:

- t_3 corresponds to the "fictional" transaction that wrote the initial values which cannot be the case when $(t_1, t_2) \in \mathsf{CO}$ and t_1 writes to the same variable that t_2 reads from,
- t_3 is the last transaction that occurs before t_2 that writes the value read by t_2, which means that $(t_1, t_3) \in ARB$ which contradicts the fact that $(t_3, t_1) \in \mathsf{WW}$ since $\mathsf{WW} \subset ARB$.

($\Leftarrow$) Let tr be a trace such that ARB_0^+ and $\mathsf{CO}_0^+; \mathsf{RW}$ are acyclic. Then, we define the relations CO and ARB such that $\mathsf{CO} = \mathsf{CO}_0^+$ and ARB is any total order that includes ARB_0^+. Then, we obtain that $(\mathsf{CO} \cup \mathsf{WW})^+ \subset ARB$ and $\mathsf{CO}; \mathsf{RW}$ is acyclic. Thus, tr satisfies the properties $\mathsf{AxCausal}$ and AxArb. Next, we will show that tr satisfies $\mathsf{AxRetVal}$. Let t be a transaction in tr that contains a read event $\mathsf{re}(t, x, v)$. Let t_0 be transaction in tr such that

$$t_0 = Max_{ARB}(\{t' \in tr \mid (t', t) \in \mathsf{CO} \land \exists\ \mathsf{we}(t', x, \cdot) \in f(t')\})$$

then the read must return a value written by t_0. Assume by contradiction that there exists some other transaction $t_1 \neq t_0$ such that $(t_1, t) \in \mathsf{WR}$. Then, we get that $(t_1, t_0) \in ARB$ and both write to x, therefore, $(t_1, t_0) \in \mathsf{WW}$ since $\mathsf{WW} \subset ARB$. Combining $(t_1, t) \in \mathsf{WR}$ and $(t_1, t_0) \in \mathsf{WW}$ we obtain $(t, t_0) \in \mathsf{RW}$ and since $(t_0, t) \in \mathsf{CO}$ then we obtain that $(t, t) \in \mathsf{CO}; \mathsf{RW}$ which contradicts the fact that $\mathsf{CO}; \mathsf{RW}$ is acyclic. Therefore, the read value was written by t_0 and tr satisfies $\mathsf{AxRetVal}$. $\square$

Next we show that a trace tr of a program $\mathcal{P}$ is $\mathtt{CC}$ iff the corresponding trace $tr_{\clubsuit}$ of $\mathcal{P}_{\clubsuit}$ is $\mathtt{CC}$ as well. This result is based on the observation that cycles in ARB_0^+ or $\mathsf{CO}_0^+; \mathsf{RW}$ cannot be broken by splitting transactions.

Lemma 5.3. *A trace tr of $\mathcal{P}$ is $\mathtt{CC}$ iff the corresponding trace $tr_{\clubsuit}$ of $\mathcal{P}_{\clubsuit}$ is $\mathtt{CC}$.*

Proof. The only-if direction follows from Lemma 5.1. For the if direction: consider a trace $tr_{\clubsuit}$ which is $\mathtt{CC}$. We prove by contradiction that tr must be $\mathtt{CC}$ as well. Assume that tr is not $\mathtt{CC}$ then it must contain a cycle in either ARB_0^+ or $\mathsf{CO}_0^+; \mathsf{RW}$ (based on Lemma 5.2). In the rest of the proof when we mention a cycle we implicitly refer to a cycle in either ARB_0^+ or $\mathsf{CO}_0^+; \mathsf{RW}$.

Splitting every transaction $t \in tr$ in a trace to a pair of transactions $t[r]$ and $t[w]$ that occur in this order might not maintain a cycle of tr. However, we prove that this is not possible and our splitting conserves the cycle. Assume we have a vertex t as part of the cycle. We show that t can be split into two transactions $t[r]$ and $t[w]$ while maintaining the cycle. Note that t is part of a cycle iff either

1. $(t_1, t) \in ARB_0$ and $(t, t_2) \in ARB_0$ or
2. $(t_1, t) \in \mathsf{CO}_0$ and $(t, t_2) \in \mathsf{CO}_0$ or
3. $(t_1, t) \in \mathsf{CO}_0$ and $(t, t_2) \in \mathsf{RW}$ or
4. $(t_1, t) \in \mathsf{RW}$ and $(t, t_2) \in \mathsf{CO}_0$

where t_1 and t_2 might refer to the same transaction. Thus, by splitting t to $t[r]$ and $t[w]$, the above four cases imply that:

1. if $(t_1, t) \in \mathsf{CO}_0$ and $(t, t_2) \in ARB_0$ then $(t'_1, t[r]) \in (\mathsf{PO}_♣ \cup \mathsf{WR}_♣)$ and $(t[w], t'_2) \in (\mathsf{PO}_♣ \cup \mathsf{WR}_♣ \cup \mathsf{WW}_♣)$ where $t'_1 \in \{t_1[r], t_1[w]\}$ and $t'_2 \in \{t_2[r], t_2[w]\}$. This maintains the vertices t'_1 and t'_2 connected in the cycle formed by the dependency relations of $tr_♣$ since $(t[r], t[w]) \in \mathsf{PO}_♣$;

2. if $(t_1, t) \in \mathsf{WW}$ and $(t, t_2) \in ARB_0$ then $(t'_1, t[w]) \in \mathsf{WW}_♣$ and $(t[w], t'_2) \in (\mathsf{PO}_♣ \cup \mathsf{WR}_♣ \cup \mathsf{WW}_♣)$ which maintains the vertices t'_1 and t'_2 connected in the cycle formed by the dependency relations of $tr_♣$;

3. $(t_1, t) \in \mathsf{CO}_0$ and $(t_2, t) \in \mathsf{RW}$ then $(t'_1, t[r]) \in (\mathsf{PO}_♣ \cup \mathsf{WR}_♣)$ and $(t[r], t'_2) \in \mathsf{RW}_♣$ maintains the vertices t'_1 and t'_2 connected in the cycle formed by the dependency relations of $tr_♣$;

4. $(t_1, t) \in \mathsf{RW}$ and $(t_2, t) \in \mathsf{CO}_0$ then $(t'_1, t[w]) \in \mathsf{RW}_♣$ and $(t[w], t'_2) \in (\mathsf{PO}_♣ \cup \mathsf{WR}_♣)$ which maintains the vertices t'_1 and t'_2 connected in the cycle formed by the dependency relations of $tr_♣$ as well.

Therefore, doing the splitting creates a cycle in either $(\mathsf{PO}_♣ \cup \mathsf{WR}_♣ \cup \mathsf{WW}_♣)^+$ or $(\mathsf{PO}_♣ \cup \mathsf{WR}_♣)^+ ; \mathsf{RW}_♣$ which implies that $tr_♣$ is not `CC`, a contradiction. $\square$

The following lemma shows that a trace tr is `PC` iff the corresponding trace $tr_♣$ is `SER`. The if direction in the proof is based on constructing a causal order CO and an arbitration order ARB for the trace tr from the arbitration order $ARB_♣$ in $tr_♣$ (since $tr_♣$ is a trace under serializability $\mathsf{CO}_♣$ and $ARB_♣$ coincide). These are the smallest transitive relations such that:

- if $(t_1[w], t_2[r]) \in ARB_♣$ then $(t_1, t_2) \in \mathsf{CO}$,
- if $(t_1[w], t_2[w]) \in ARB_♣$ then $(t_1, t_2) \in ARB$[4].

The only-if direction is based on the fact that any cycle in the dependency relations of tr that is admitted under `PC` (characterized in Lemma 5.7) is "broken" by splitting transactions. Also, splitting transactions cannot introduce new cycles that do not originate in tr.

Lemma 5.4. *A trace tr is* `PC` *iff $tr_♣$ is* `SER`

The lemmas above are used to prove Theorem 5.1 as follows:

Proof of Theorem 5.1: For the if direction, assume by contradiction that $\mathcal{P}$ is not robust against `CC` relative to `PC`. Then, there must exist a trace $tr \in \mathbb{T}\mathsf{r}_{\mathsf{CC}}(\mathcal{P}) \setminus \mathbb{T}\mathsf{r}_{\mathsf{PC}}(\mathcal{P})$. Lemmas 5.3 and 5.4 imply

[4]If $t_1[w]$ is empty (t_1 is read-only), then we set $(t_1, t_2) \in ARB$ if $(t_1[r], t_2[w]) \in \mathsf{CO}_♣$. If $t_2[w]$ is empty, then $(t_1, t_2) \in ARB$ if $(t_1[w], t_2[r]) \in \mathsf{CO}_♣$. If both $t_1[w]$ and $t_2[w]$ are empty, then $(t_1, t_2) \in ARB$ if $(t_1[r], t_2[r]) \in \mathsf{CO}_♣$.

that the corresponding trace $tr_{\clubsuit}$ of $\mathcal{P}_{\clubsuit}$ is CC and not SER. Thus, $\mathcal{P}_{\clubsuit}$ is not robust against CC relative to SER. The only-if direction is proved similarly. □

Robustness against CC relative to SER has been shown to be reducible in polynomial time to the reachability problem under SER [45]. Given a program $\mathcal{P}$ and a control location ℓ, the reachability problem under SER asks whether there exists an execution of $\mathcal{P}$ under SER that reaches ℓ. Therefore, as a corollary of Theorem 5.1, we obtain the following:

Corollary 5.1. *Checking robustness against* CC *relative to* PC *is reducible to the reachability problem under* SER *in polynomial time.*

In the following we discuss the complexity of this problem in the case of finite-state programs (bounded data domain). The upper bound follows from Corollary 5.1 and standard results about the complexity of the reachability problem under sequential consistency, which extend to SER, with a bounded [108] or parametric number of processes [146]. For the lower bound, given an instance $(\mathcal{P}, \ell)$ of the reachability problem under sequential consistency, we construct a program $\mathcal{P}'$ where each statement s of $\mathcal{P}$ is executed in a different transaction that guards[5] the execution of s using a global lock (the lock can be implemented in our programming language as usual, e.g., using a busy wait loop for locking), and where reaching the location ℓ enables the execution of a "gadget" that corresponds to the SB program in Figure 5.2a. Executing each statement under a global lock ensures that every execution of $\mathcal{P}'$ under CC is serializable, and faithfully represents an execution of $\mathcal{P}$ under sequential consistency. Moreover, $\mathcal{P}$ reaches ℓ iff $\mathcal{P}'$ contains a robustness violation, which is due to the SB execution.

Corollary 5.2. *Checking robustness of a program with a fixed number of variables and bounded data domain against* CC *relative to* PC *is PSPACE-complete when the number of processes is bounded and EXPSPACE-complete, otherwise.*

5.5 Robustness Against PC Relative to SI

In this section, we show that checking robustness against PC relative to SI can be reduced in polynomial time to a reachability problem under the SER semantics. We reuse the program transformation from the previous section that allows to simulate PC behaviors on top of SER, and additionally, we

[5]That is, the transaction is of the form [lock; s; unlock]

provide a characterization of traces that distinguish the PC semantics from SI. We use this characterization to define an instrumentation (monitor) that is able to detect if a program under PC admits such traces.

We show that the happens-before cycles in a robustness violation (against PC relative to SI) must contain a WW dependency followed by a RW dependency, and they should not contain two successive RW dependencies. This follows from the fact that every happens-before cycle in a PC trace must contain either two successive RW dependencies, or a WW dependency followed by a RW dependency. Otherwise, the happens-before cycle will imply a cycle in the arbitration order. Then, any trace under PC where all its simple happens-before cycles contain two successive RW dependencies is possible under SI. As a first step, we prove the following theorem.

Theorem 5.2. *A program $\mathcal{P}$ is robust against* PC *relative to* SI *iff every happens-before cycle in a trace of $\mathcal{P}$ under* PC *contains two successive* RW *dependencies.*

Before giving the proof of the above theorem, we state several intermediate results that characterize cycles in PC or SI traces. First, we show that every PC trace in which all simple happens-before cycles contain two successive RW is also a SI trace.

Lemma 5.5. *If a trace tr is* PC *and all happens-before cycles in tr contain two successive* RW *dependencies, then tr is* SI.

Proof. Let ARB_1 be a total order that includes ARB_0^+ and $ARB_0^+;\mathsf{RW};ARB_0^*$ (ARB_0^* is the reflexive closure of ARB_0). This is well defined because there exists no cycle between tuples in these two relations. Indeed, if $(t_1,t_2) \in ARB_0^+$ and there exist t_3 and t_4 such that $(t_2,t_3) \in ARB_0^+$, $(t_3,t_4) \in \mathsf{RW}$, and $(t_4,t_1) \in ARB_0^*$, then we have a cycle in $ARB_0^+;\mathsf{RW}$ that does not contain two successive RW dependencies, which contradicts the hypothesis. Also, for every pair of transactions (t_1,t_2) there cannot exist t_3 and t_4 such that

$$(t_2,t_3) \in ARB_0^+,\ (t_3,t_4) \in \mathsf{RW} \text{ and } (t_4,t_1) \in ARB_0^*$$

and t_3' and t_4' such that

$$(t_1,t_3') \in ARB_0^+,\ (t_3',t_4') \in \mathsf{RW} \text{ and } (t_4',t_2) \in ARB_0^*$$

This will imply a cycle in $ARB_0^+;\mathsf{RW};ARB_0^+;\mathsf{RW}$ which again contradicts the hypothesis. Also, let CO_1 be the smallest transitive relation that includes ARB_0^+ and $ARB_1;ARB_0^+$. We show that CO_1 and ARB_1 are causal and arbitration orders of tr that satisfy all the axioms of SI.

$\mathsf{AxCausal}$ and AxArb hold trivially. Since $\mathsf{WW} \subseteq \mathsf{CO}_1$, $\mathsf{AxConflict}$ holds as well. AxPC holds because $ARB_1; \mathsf{CO}_1 = ARB_1; (ARB_0^+ \cup ARB_1; ARB_0^+)^+ = ARB_1; ARB_0^+ \subset \mathsf{CO}_1$.

The axiom $\mathsf{AxRetVal}$ is equivalent to the acyclicity of $\mathsf{CO}_1; \mathsf{RW}$ when $\mathsf{AxCausal}$ and AxArb hold. Assume by contradiction that $\mathsf{CO}_1; \mathsf{RW}$ is cyclic. From the definition of CO_1 and the fact that ARB_1 is total order we obtain that either:

- $ARB_0^+; \mathsf{RW}$ is cyclic, which implies that there exists a happens-before cycle that does not contain two successive RW, which contradicts the hypothesis, or
- $ARB_1; ARB_0^+; \mathsf{RW}$ is cyclic, which implies that there exist t_1, t_2, and t_3 such that $(t_2, t_3) \in ARB_0^+$, $(t_3, t_1) \in \mathsf{RW}$ and $(t_1, t_2) \in ARB_1$. This contradicts the fact that $(t_2, t_3) \in ARB_0^+$ and $(t_3, t_1) \in \mathsf{RW}$ implies $(t_2, t_1) \in ARB_1$.

Therefore, *tr* satisfies $\mathsf{AxRetVal}$ for CO_1 and ARB_1, which concludes the proof. □

The proof of Theorem 5.2 also relies on the following lemma that characterizes happens-before cycles permissible under `SI`.

Lemma 5.6. *[65, 47] If a trace tr is* `SI`*, then all its happens-before cycles must contain two successive* RW *dependencies.*

PROOF of Theorem 5.2: For the only-if direction, if $\mathcal{P}$ is robust against `PC` relative to `SI` then every trace *tr* of $\mathcal{P}$ under `PC` is `SI` as well. Therefore, by Lemma 5.6, all cycles in *tr* contain two successive RW which concludes the proof of this direction. For the reverse, let *tr* be a trace of $\mathcal{P}$ under `PC` such that all its happens-before cycles contain two successive RW. Then, by Lemma 5.5, we have that *tr* is `SI`. Thus, every trace *tr* of $\mathcal{P}$ under `PC` is `SI`. □

Next, we present an important lemma that characterizes happens before cycles possible under the `PC` semantics. This is a strengthening of a result in [47] which shows that all happens before cycles under `PC` must have two successive dependencies in $\{\mathsf{RW}, \mathsf{WW}\}$ and at least one RW. We show that the two successive dependencies cannot be RW followed WW, or two successive WW.

Lemma 5.7. *If a trace tr is* `PC` *then all happens-before cycles in tr must contain either two successive* RW *dependencies or a* WW *dependency followed by a* RW *dependency.*

Proof. It was shown in [47] that all happens-before cycles under `PC` must contain two successive dependencies in $\{\mathsf{RW}, \mathsf{WW}\}$ and at least one RW. Assume by contradiction that there exists a cycle with RW dependency followed by WW dependency or two successive WW dependencies. This cycle must contain at least one additional dependency. Otherwise, the cycle would also have a WW dependency followed by a RW dependency, or it would imply a cycle in WW, which is not possible (since $\mathsf{WW} \subset ARB$ and ARB is a total order). Then, we get that the dependency just before RW is either PO or WR (i.e., CO_0) since we cannot have RW or WW followed by RW. Also, the relation after WW is either PO or WR or WW (i.e., ARB_0) since we cannot have WW followed by RW. Thus, the cycle has the following shape:

$$\begin{tikzcd}
t_1 \arrow[r, "\mathsf{RW}"] & t_2 \arrow[r, "\mathsf{WW}"] & t_3 \arrow[r, "ARB_0"] & t_4 & \cdots & t_i \arrow[r, "\mathsf{CO}_0"] & t_{i+1} \arrow[r, "\mathsf{RW}"] & t_{i+2} \arrow[r, "\mathsf{WW}"] & t_{i+3} & \cdots & t_{n-4} \arrow[r, "\mathsf{CO}_0"] & t_{n-3} \arrow[r, "\mathsf{RW}"] & t_{n-2} \arrow[r, "\mathsf{WW}"] & t_{n-1} \arrow[r, "ARB_0"] & t_n \arrow[llllllllllllll, bend left, "\mathsf{CO}_0"]
\end{tikzcd}$$

Since $\mathsf{CO}_0; \mathsf{RW} \subseteq ARB$ is a consequence of the $\texttt{PC}$ axioms [71], we get that $(t_n, t_2) \in ARB$, $(t_i, t_{i+2}) \in ARB$ and $(t_{n-4}, t_{n-2}) \in ARB$, which allows to "short-circuit" the cycle. Using the fact that $\mathsf{WW} \subset ARB$, $\mathsf{CO}_0 \subset ARB$, and $ARB_0 \subset ARB$, and applying the short-circuiting process multiple times, we obtain a cycle in the arbitration order ARB which contradicts the fact that ARB is a total order. $\square$

Combining the results of Theorem 5.2 and Lemmas 5.4 and 5.7, we obtain the following characterization of traces which violate robustness against $\texttt{PC}$ relative to $\texttt{SI}$.

Theorem 5.3. *A program $\mathcal{P}$ is not robust against* $\texttt{PC}$ *relative to* $\texttt{SI}$ *iff there exists a trace $tr_\clubsuit$ of $\mathcal{P}_\clubsuit$ under* $\texttt{SER}$ *such that the trace tr obtained by merging*[6] *read and write transactions in $tr_\clubsuit$ contains a happens-before cycle that does not contain two successive* RW *dependencies, and it contains a* WW *dependency followed by a* RW *dependency.*

The results above enable a reduction from checking robustness against $\texttt{PC}$ relative to $\texttt{SI}$ to a reachability problem under the $\texttt{SER}$ semantics. For a program $\mathcal{P}$, we define an instrumentation denoted by $[\![\mathcal{P}]\!]$, such that $\mathcal{P}$ is not robust against $\texttt{PC}$ relative to $\texttt{SI}$ iff $[\![\mathcal{P}]\!]$ violates an assertion under $\texttt{SER}$. The instrumentation consists in rewriting every transaction of $\mathcal{P}$ as shown in Figure 5.5[7].

The instrumentation $[\![\mathcal{P}]\!]$ running under $\texttt{SER}$ simulates the $\texttt{PC}$ semantics of $\mathcal{P}$ using the same idea of decoupling the execution of the read part of a transaction from the write part. It violates an assertion

$$\begin{tikzcd}
\alpha & t_\# \arrow[r, bend left, "\mathsf{RW}", "\beta"'] & t_0 \arrow[r, bend left, "\mathsf{HB}", "\gamma"'] & t \arrow[ll, bend left, "\mathsf{WW}"]
\end{tikzcd}$$

Figure 5.4: Execution simulating a violation to robustness against $\texttt{PC}$ relative to $\texttt{SI}$.

[6]This transformation has been defined at the beginning of §5.4.

[7]The instrumentation uses program constructs which can be defined as syntactic sugar from the syntax presented in §2.2, e.g., if-then-else statements (outside transactions).

when it simulates a `PC` trace containing a happens-before cycle as in Theorem 5.3. The execution corresponding to this trace has the shape given in Figure 5.4, where $t_{\#}$ is the transaction that occurs between the WW and the RW dependencies, α, β, and γ are sequences of transactions, and every transaction executed after $t_{\#}$ (this can be a full transaction in $\mathcal{P}$, or only the read or write part of a transaction in $\mathcal{P}$) is related by a happens-before path to $t_{\#}$ (otherwise, the execution of this transaction can be reordered to occur before $t_{\#}$). A transaction in $\mathcal{P}$ can have its read part included in α and the write part included in β or γ. Also, β and γ may contain transactions in $\mathcal{P}$ that executed only their read part. It is possible that $t_0 = t$, $\beta = \gamma = \epsilon$, and $\alpha = \epsilon$ (the LU program shown in Figure 5.2b is an example where this can happen). The instrumentation uses auxiliary variables to track happens-before dependencies, which are explained below.

The instrumentation executes (incomplete) transactions without affecting the auxiliary variables (without tracking happens-before dependencies) until a non-deterministically chosen point in time when it declares the current transaction as the candidate for $t_{\#}$. Only one candidate for $t_{\#}$ can be chosen during the execution. This transaction executes only its reads and it chooses non-deterministically a variable that it could write as a witness for the WW dependency. The name of this variable is stored in a global variable `varW` (see the definition of $\mathcal{I}_{\#}($ x := e $))$. The writes are *not* applied on the shared memory. Intuitively, $t_{\#}$ should be thought as a transaction whose writes are delayed for later, after transaction t in Figure 5.4 executed. The instrumentation checks that $t_{\#}$ and t can be connected by some happens-before path that includes the RW and WW dependencies, and that does not contain two consecutive RW dependencies. If it is the case, it violates an assertion at the commit point of t. Since the write part of $t_{\#}$ is intuitively delayed to execute after t, the process executing $t_{\#}$ is disabled all along the execution (see the `assume false`).

After choosing the candidate for $t_{\#}$, the instrumentation uses the auxiliary variables for tracking happens-before dependencies. Therefore, `rdSet` and `wrSet` record variables read and written, respectively, by transactions that are connected by a happens-before path to $t_{\#}$ (in a trace of $\mathcal{P}$). This is ensured by the assume at line 7. During the execution, the variables read or written by a transaction[8] that writes a variable in `rdSet` (see line 33), or reads or writes a variable in `wrSet` (see line 21 and 28), will be added to these sets (see lines 17 and 18). Since the variables that $t_{\#}$ writes in $\mathcal{P}$ are not recorded in `wrSet`, these happens-before paths must necessarily start with a RW dependency (from $t_{\#}$). When the assertion fails (line 8), the condition `varW` $\in$ `wrSet`' ensures that

[8]These are stored in the local variables `rdSet'` and `wrSet'` while the transaction is running.

Transaction "**begin** ⟨read⟩* ⟨test⟩* ⟨write⟩* **commit**" is rewritten to:

```
if ( !done$_\#$ )
   if ( * )
      begin <read>$^*$ <test>$^*$ commit
      if ( !done$_\#$ )
         begin <write>$^*$ commit
      else
         $\mathcal{I}$(begin) ($\mathcal{I}$(<write>))$^*$ $\mathcal{I}$(commit)
   else
      begin ($\mathcal{I}_\#$(<read>))$^*$ <test>$^*$ ($\mathcal{I}_\#$(<write>))$^*$ $\mathcal{I}_\#$(commit)
      assume false;
else if ( * )
   rdSet' := ∅;
   wrSet' := ∅;
   $\mathcal{I}$(begin) ($\mathcal{I}$(<read>))$^*$ <test>$^*$ $\mathcal{I}$(commit)
   $\mathcal{I}$(begin) ($\mathcal{I}$(<write>))$^*$ $\mathcal{I}$(commit)
```

$\mathcal{I}_\#$(r := x):

```
r := x;
hbR['x'] := 0;
rdSet := rdSet ∪ { 'x' };
```

$\mathcal{I}_\#$(x := e):

```
if ( varW == ⊥ and * )
  varW := 'x';
```

$\mathcal{I}_\#$(commit):

```
assume ( varW != ⊥ )
done$_\#$ := true
```

$\mathcal{I}$(begin):

```
begin
hb := ⊥
if ( hbP != ⊥ and hbP < 2 )
  hb := 0;
else if ( hbP = 2 )
  hb := 2;
```

$\mathcal{I}$(commit):

```
assume ( hb != ⊥ )
assert ( hb == 2 or varW ∉ wrSet' );
if ( hbP == ⊥ or hbP > hb )
  hbP = hb;
for each 'x' ∈ wrSet'
  if ( hbW['x'] == ⊥ or hbW['x'] > hb )
    hbW['x'] = hb;
for each 'x' ∈ rdSet'
  if ( hbR['x'] == ⊥ or hbR['x'] > hb )
    hbR['x'] = hb;
rdSet := rdSet ∪ rdSet';
wrSet := wrSet ∪ wrSet';
commit
```

$\mathcal{I}$(r := x):

```
r := x;
rdSet' := rdSet' ∪ { 'x' };
if ( 'x' ∈ wrSet )
  if ( hbW['x'] != 2 )
    hb := 0
  else if ( hb == ⊥ )
    hb := hbW['x']
```

$\mathcal{I}$(x := e):

```
x := e;
wrSet' := wrSet' ∪ { 'x' };
if ( 'x' ∈ wrSet )
  if ( hbW['x'] != 2 )
    hb := 0
  else if ( hb == ⊥ )
    hb := hbW['x']
if ( 'x' ∈ rdSet )
  if ( hb = ⊥ or hb > hbR['x'] + 1 )
    hb := min(hbR['x'] + 1,2)
```

Figure 5.5: A program instrumentation for checking robustness against PC relative to SI. The auxiliary variables used by the instrumentation are shared variables, except for **hbP**, **rdSet'**, and **wrSet'**, which are process-local variables, and they are initially set to ⊥.

the current transaction has a WW dependency towards the write part of $t_\#$ (the current transaction plays the role of t in Figure 5.4).

The rest of the instrumentation checks that there exists a happens-before path from $t_\#$ to t that does not include two consecutive RW dependencies, called a `SI`$_\neg$ path. This check is based on the auxiliary variables whose name is prefixed by `hb` and which take values in the domain $\{\bot, 0, 1, 2\}$ ($\bot$ represents the initial value). Therefore,

- `hbR['x']` (resp., `hbW['x']`) is 0 iff there exists a transaction t' that reads `x` (resp., writes to `x`), such that there exists a `SI`$_\neg$ path from $t_\#$ to t' that ends with a dependency which is *not* RW,

- `hbR['x']` (resp., `hbW['x']`) is 1 iff there exists a transaction t' that reads `x` (resp., writes to `x`) that is connected to $t_\#$ by a `SI`$_\neg$ path, and *every* `SI`$_\neg$ path from $t_\#$ to a transaction t'' that reads `x` (resp., writes to `x`) ends with an RW dependency,

- `hbR['x']` (resp., `hbW['x']`) is 2 iff there exists no `SI`$_\neg$ path from $t_\#$ to a transaction t' that reads `x` (resp., writes to `x`).

The local variable `hbP` has the same interpretation, except that t' and t'' are instantiated over transactions in the same process (that already executed) instead of transactions that read or write a certain variable. Similarly, the variable `hb` is a particular case where t' and t'' are instantiated to the current transaction. The violation of the assertion at line 8 implies that `hb` is 0 or 1, which means that there exists a `SI`$_\neg$ path from $t_\#$ to t.

During each transaction that executes after $t_\#$, the variable `hb` characterizing happens-before paths that end in this transaction is updated every time a new happens-before dependency is witnessed (using the values of the other variables). For instance, when witnessing a WR dependency (line 21), if there exists a `SI`$_\neg$ path to a transaction that writes to `x`, then the path that continues with the WR dependency towards the current transaction is also a `SI`$_\neg$ path, and the last dependency of this path is not RW. Therefore, `hb` is set to 0 (see line 23). Otherwise, if every path to a transaction that writes to `x` is not a `SI`$_\neg$ path, then every path that continues to the current transaction (by taking the WR dependency) remains a non `SI`$_\neg$ path, and `hb` is set to the value of `hbW['x']`, which is 2 in this case (see line 25). Before ending a transaction, the value of `hb` can be used to modify the `hbR`, `hbW`, and `hbP` variables, but only if those variables contain bigger values (see lines 9–16).

The correctness of the instrumentation is stated in the following theorem.

Theorem 5.4. *A program $\mathcal{P}$ is robust against* `PC` *relative to* `SI` *iff the instrumentation in Figure 5.5 does not violate an assertion when executed under* `SER`*.*

Theorem 5.4 implies the following complexity result for finite-state programs. The lower bound is proved similarly to the case `CC` vs `PC`.

Corollary 5.3. *Checking robustness of a program with a fixed number of variables and bounded data domain against* `PC` *relative to* `SI` *is PSPACE-complete when the number of processes is bounded and EXPSPACE-complete, otherwise.*

Checking robustness against `CC` relative to `SI` can be also shown to be reducible (in polynomial time) to a reachability problem under `SER` by combining the results of checking robustness against `CC` relative to `PC` and `PC` relative to `SI`.

Theorem 5.5. *A program $\mathcal{P}$ is robust against* `CC` *relative to* `SI` *iff $\mathcal{P}$ is robust against* `CC` *relative to* `PC` *and $\mathcal{P}$ is robust against* `PC` *relative to* `SI`*.*

Remark 5.1. *Our reductions of robustness checking to reachability apply to an extension of our programming language where the number of processes is unbounded and each process can execute an arbitrary number of times a statically known set of transactions. This holds because the instrumentation in Figure 5.5 and the one in [45] (for the case* `CC` *vs.* `SER`*) consist in adding a set of instructions that manipulate a fixed set of process-local or shared variables, which do not store process or transaction identifiers. These reductions extend also to SQL queries that access unbounded size tables. Rows in a table can be interpreted as memory locations (identified by primary keys in unbounded domains, e.g., integers), and SQL queries can be interpreted as instructions that read-/write a set of locations in one shot. These possibly unbounded sets of locations can be represented symbolically using the conditions in the SQL queries (e.g., the condition in the WHERE part of a SELECT). The instrumentation in Figure 6 needs to be adapted so that read and write sets are updated by adding sets of locations for a given instruction (represented symbolically as mentioned above).*

5.6 Robustness Against `CC` relative to `SI`

Checking robustness against `CC` relative to `SI` can be also shown to be reducible (in polynomial time) to a reachability problem under `SER` by combining the results in the previous two sections.

Theorem 5.6. *A program $\mathcal{P}$ is robust against* `CC` *relative to* `SI` *iff $\mathcal{P}$ is robust against* `CC` *relative to* `PC` *and $\mathcal{P}$ is robust against* `PC` *relative to* `SI`*.*

5.7 Proving Robustness Using Commutativity Dependency Graphs

We describe an approximated technique for proving robustness, which leverages the concept of left/right mover in Lipton's reduction theory [120]. This technique reasons on the *commutativity dependency graph*, introduced in Chapter 4, associated to the transformation $\mathcal{P}_\clubsuit$ of an input program $\mathcal{P}$ that allows to simulate the `PC` semantics under serializability (we use a slight variation of the original definition of this class of graphs). We characterize robustness against `CC` relative to `PC` and `PC` relative to `SI` in terms of certain properties that (simple) cycles in this graph must satisfy.

We recall the concept of movers and the definition of commutativity dependency graphs. Given a program $\mathcal{P}$ and a trace $tr = t_1 \cdot \ldots \cdot t_n \in \mathbb{T}\mathrm{r}_{\mathtt{SER}}(\mathcal{P})$ of $\mathcal{P}$ under serializability, we say that $t_i \in tr$ *moves right (resp., left)* in tr if $t_1 \cdot \ldots \cdot t_{i-1} \cdot t_{i+1} \cdot t_i \cdot t_{i+2} \cdot \ldots \cdot t_n$ (resp., $t_1 \cdot \ldots \cdot t_{i-2} \cdot t_i \cdot t_{i-1} \cdot t_{i+1} \cdot \ldots \cdot t_n$) is also a valid execution of $\mathcal{P}$, t_i and t_{i+1} (resp., t_{i-1}) are executed by distinct processes, and both traces reach the same end state. A transaction $t \in \mathsf{Tr}(\mathcal{P})$ is not a right (resp., left) mover iff there exists a trace $tr \in \mathbb{T}\mathrm{r}_{\mathtt{SER}}(\mathcal{P})$ such that $t \in tr$ and t does not move right (resp., left) in tr. Thus, when a transaction t is *not* a right mover then there must exist another transaction $t' \in tr$ which caused t to not be permutable to the right (while preserving the end state). Since t and t' do not commute, then this must be because of either a write-read, write-write, or a read-write dependency relation between the two transactions. We say that t is not a right mover because of t' and a dependency relation that is either write-read, write-write, or read-write. Notice that when t is not a right mover because of t' then t' is not a left mover because of t.

We define M_{WR} as a binary relation between transactions such that $(t, t') \in \mathsf{M}_{\mathsf{WR}}$ when t is *not* a right mover because of t' and a write-read dependency (t' reads some value written by t). We define the relations M_{WW} and M_{RW} corresponding to write-write and read-write dependencies in a similar way. We call M_{WR}, M_{WW}, and M_{RW}, *non-mover* relations.

The *commutativity dependency graph* of a program $\mathcal{P}$ is a graph where vertices represent transactions in $\mathcal{P}$. Two vertices are linked by a program order edge if the two transactions are executed by the same process. The other edges in this graph represent the "non-mover" relations M_{WR}, M_{WW}, and M_{RW}. Two vertices that represent the two components $t[w]$ and $t[r]$ of the same transaction t (already linked by PO edge) are also linked by an undirected edge labeled by STO (same-transaction relation).

Our results about the robustness of a program $\mathcal{P}$ are stated over a slight variation of the commutativity dependency graph of $\mathcal{P}_\clubsuit$ (where a transaction is either read-only or write-only). This graph contains additional undirected edges that link every pair of transactions $t[r]$ and $t[w]$ of $\mathcal{P}_\clubsuit$ that were originally components of the same transaction t in $\mathcal{P}$. Given such a commutativity dependency graph, the robustness of $\mathcal{P}$ is implied by the absence of cycles of specific shapes. These cycles can be seen as an abstraction of potential robustness violations for the respective semantics (see Theorem 5.7 and Theorem 5.8). Figure 5.6 pictures the commutativity dependency graph for the MP program. Since every transaction in MP is singleton, the two programs MP and $\mathsf{MP}_\clubsuit$ coincide.

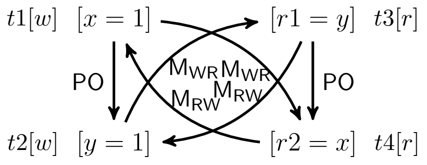


Figure 5.6: The commutativity dependency graph of the $\mathsf{MP}_\clubsuit$ program.

Using the characterization of robustness violations against CC relative to SER from [45] and the reduction in Theorem 5.1, we obtain the following result concerning the robustness against CC relative to PC.

Theorem 5.7. *Given a program $\mathcal{P}$, if the commutativity dependency graph of the program $\mathcal{P}_\clubsuit$ does not contain a simple cycle formed by $t_1 \cdots t_i \cdots t_n$ such that:*

- $(t_n, t_1) \in \mathsf{M}_{\mathsf{RW}}$*;*
- $(t_j, t_{j+1}) \in (\mathsf{PO} \cup \mathsf{WR})^*$*, for* $j \in [1, i-1]$*;*
- $(t_i, t_{i+1}) \in (\mathsf{M}_{\mathsf{RW}} \cup \mathsf{M}_{\mathsf{WW}})$*;*
- $(t_j, t_{j+1}) \in (\mathsf{M}_{\mathsf{RW}} \cup \mathsf{M}_{\mathsf{WW}} \cup \mathsf{M}_{\mathsf{WR}} \cup \mathsf{PO})$*, for* $j \in [i+1, n-1]$.

then $\mathcal{P}$ is robust against CC *relative to* PC.

Proof. It is enough to show: if $\mathcal{P}$ is not robust against CC relative to PC then we have a simple cycle in the commutativity dependency graph of $\mathcal{P}_\clubsuit$ of the form above. Assume $\mathcal{P}$ is not robust against CC relative to PC. Then, from Theorem 5.1, we obtain $\mathcal{P}_\clubsuit$ is not robust against CC relative to SER. Also it was shown in [45] that if a program is not robust then there must exist a robustness violation trace (CC relative to SER) $tr_\clubsuit$ of the shape $tr_\clubsuit = \alpha \cdot t_1 \cdot \beta \cdot t_i \cdot t_{i+1} \cdot \gamma \cdot t_n$ where $(t_1, t_i) \in (\mathsf{PO} \cup \mathsf{WR})^+$, $(t_i, t_{i+1}) \in (\mathsf{WW} \cup \mathsf{RW})$, $(t_{i+1}, t_n) \in \mathsf{HB}$, and $(t_n, t_1) \in \mathsf{RW}$. Note that since transactions in the trace $tr_\clubsuit$ can either be read-only or write-only. Then, $(t_i, t_{i+1}) \in (\mathsf{WW} \cup \mathsf{RW})$ and $(t_n, t_1) \in \mathsf{RW}$

imply that t_1 and t_{i+1} must be a write-only transactions and t_n must be a read-only transaction. Note that we may have $\beta = \gamma = \epsilon$ as the case for the trace of the SB program given in Figure 5.2a.

We consider first the general case when $t_1 \not\equiv t_2$. The other case can be proved in the same way.

Consider the prefix tr_p of $tr_\clubsuit$: $tr_p = \alpha \cdot t_1 \cdot \beta \cdot t_i$ where $(t_1, t_i) \in (\mathsf{PO} \cup \mathsf{WR})^+$ which is a $\mathtt{SER}$ trace of $\mathcal{P}_\clubsuit$. Then, we have a sequence of transactions from t_1 to t_i that are related by either PO or WR. In the case two transactions are only related by WR, then the first transaction is not a right mover because of the second transaction reads from a write in the first transaction. Thus, we can relate the two transactions using the relation M_{WR} in the commutativity dependency graph.

Similarly consider the following trace tr_s extracted from $tr_\clubsuit$: $tr_s = \alpha \cdot t_{i+1} \cdot \gamma \cdot t_n$ where $(t_{i+1}, t_n) \in \mathsf{HB}$ which is a $\mathtt{SER}$ trace of $\mathcal{P}_\clubsuit$. Similar to before, we have a sequence of transactions from t_{i+1} to t_n that are related by either PO, WR, WW, or RW. For any two transactions that are related only by either WR, WW, or RW, this implies that the first transaction is not a right mover because of the second transaction and a write-read, write-write, or read-write dependency between the two, respectively. Thus, we can relate the two transactions using either M_{WR}, M_{WW}, or M_{RW}, respectively.

Now consider the following trace tr_1 extracted from $tr_\clubsuit$: $tr_1 = \alpha \cdot t_1 \cdot \beta \cdot t_i \cdot t_{i+1}$ where $(t_i, t_{i+1}) \in (\mathsf{WW} \cup \mathsf{RW})$ is a $\mathtt{SER}$ trace of $\mathcal{P}_\clubsuit$. Because t_i and t_{i+1} are related by either WW or RW, then t_i is not a right mover because of t_{i+1} and a write-write or read-write dependency between the two, respectively. Thus, we can relate the two transactions using either M_{WW} or M_{RW}, respectively.

Finally, consider the following trace tr_2 extracted from $tr_\clubsuit$: $tr_2 = \alpha \cdot t_{i+1} \cdot \gamma \cdot t_n \cdot t_1$ where $(t_n, t_1) \in \mathsf{RW}$ is a $\mathtt{SER}$ trace of $\mathcal{P}_\clubsuit$. Because t_n and t_1 are related by RW, then t_n is not a right mover because of t_1 and a read-write dependency between the two. Thus, we can relate the two transactions using M_{RW}. □

Next we give the characterization of commutativity dependency graphs required for proving robustness against $\mathtt{PC}$ relative to $\mathtt{SI}$.

Theorem 5.8. *Given a program $\mathcal{P}$, if the commutativity dependency graph of the program $\mathcal{P}_\clubsuit$ does not contain a simple cycle formed by $t_1 \cdots t_n$ such that:*

- $(t_n, t_1) \in \mathsf{M}_{\mathsf{WW}}$*,* $(t_1, t_2) \in \mathsf{STO}$*, and* $(t_2, t_3) \in \mathsf{M}_{\mathsf{RW}}$*;*
- $(t_j, t_{j+1}) \in (\mathsf{M}_{\mathsf{RW}} \cup \mathsf{M}_{\mathsf{WW}} \cup \mathsf{M}_{\mathsf{WR}} \cup \mathsf{PO} \cup \mathsf{STO})^*$*, for* $j \in [3, n-1]$*;*
- $\forall\ j \in [2, n-2]$.

  – *if* $(t_j, t_{j+1}) \in \mathsf{M_{RW}}$ *then* $(t_{j+1}, t_{j+2}) \in (\mathsf{M_{WR}} \cup \mathsf{PO} \cup \mathsf{M_{WW}})$*;*

  – *if* $(t_{j+1}, t_{j+2}) \in \mathsf{M_{RW}}$ *then* $(t_j, t_{j+1}) \in (\mathsf{M_{WR}} \cup \mathsf{PO})$.

- $\forall\ j \in [3, n-3]$. *if* $(t_{j+1}, t_{j+2}) \in \mathsf{STO}$ *and* $(t_{j+2}, t_{j+3}) \in \mathsf{M_{RW}}$ *then* $(t_j, t_{j+1}) \in \mathsf{M_{WW}}$.

then $\mathcal{P}$ *is robust against* PC *relative to* SI.

Proof. Similar to before it is enough to show: if $\mathcal{P}$ is not robust against PC relative to SI then we have a simple cycle in the commutativity dependency graph of $\mathcal{P}_\clubsuit$ of the form above. Assume $\mathcal{P}$ is not robust against PC relative to SI. Then, from Theorem 5.4, we obtain that if $[\![\mathcal{P}]\!]$ reaches an error state under SER then we will have the following trace *tr* under SER: $tr = \alpha \cdot t_\#[r] \cdot t_3 \cdot \beta \cdot t_n \cdot t_\#[w]$[9] where $(t_\#[r], t_3) \in \mathsf{RW}$, $(t_3, t_n) \in \mathsf{HB}$, $(t_n, t_\#[w]) \in \mathsf{WW}$, and we do not have two successive RW in the happens before between t_3 and t_n. In *tr*, $t_\#[w]$ (resp., $t_\#[r]$) represents t_1 (resp., t_2) in the theorem statement. Note that we may have $\alpha = \beta = \epsilon$ as is the case of the transformed LU program given in Figure 5.3. The construction of the cycle in the commutativity dependency graph follows the same procedure taken in the proof of Theorem 5.7. The only difference is that for every two transactions of *tr* that are part of the happens before between t_3 and t_n, if the two are not connected by either PO, WR, WW, or RW then they must be the reads and writes of the same original transaction in $\mathcal{P}$. In this case, in the commutativity dependency graph we have the two transactions related by STO. □

In Figure 5.6, we have three simple cycles in the graph:

- $(t1[w], t4[r]) \in \mathsf{M_{WR}}$ and $(t4[r], t1[w]) \in \mathsf{M_{RW}}$,
- $(t2[w], t3[r]) \in \mathsf{M_{WR}}$ and $(t3[r], t2[w]) \in \mathsf{M_{RW}}$,
- $(t1[w], t2[w]) \in \mathsf{PO}$, $(t2[w], t3[r]) \in \mathsf{M_{WR}}$, $(t3[r], t4[r]) \in \mathsf{PO}$, and $(t4[r], t1[w]) \in \mathsf{M_{RW}}$.

Notice that none of the cycles satisfies the properties in Theorems 5.7 and 5.8. Therefore, MP is robust against CC relative to PC and against PC relative to SI.

Remark 5.2. *For programs that contain an unbounded number of processes, an unbounded number of instantiations of a fixed number of process "templates", or unbounded loops with bodies that contain entire transactions, a sound robustness check consists in applying Theorem 5.7 and Theorem 5.8 to (bounded) programs that contain two copies of each process template, and where each*

[9]For simplicity, we assume here that after reaching the error state we execute the writes of $t_\#$, i.e., $t_\#[w]$

loop is unfolded exactly two times. This holds because the mover relations are "static", they do not depend on the context in which the transactions execute, and each cycle requiring more than two process instances or more than two loop iterations can be short-circuited to a cycle that exists also in the bounded program. Every outgoing edge from a third instance/iteration can also be taken from the second instance/iteration. Two copies/iterations are necessary in order to discover cycles between instances of the same transaction (the cycles in Theorem 5.7 and Theorem 5.8 are simple and cannot contain the same transaction twice). These results extend easily to SQL queries as well because the notion of mover is independent of particular classes of programs or instructions.

5.8 Experimental Evaluation

We evaluated our approach for checking robustness on 7 applications extracted from the literature on databases and distributed systems, and an application Betting designed by ourselves. Two applications were extracted from the OLTP-Bench benchmark [76]: a vote recording application (Vote) and a consumer review application (Epinions). Three applications were obtained from Github projects (used also in [44, 58]): a distributed lock application for the Cassandra database (CassandraLock [67]), an application for recording trade activities (SimpleCurrencyExchange [162]), and a micro social media application (Twitter [3]). The last two applications are a movie ticketing application (FusionTicket) [105], and a user subscription application inspired by the Twitter application (Subscription). Each application consists of a set of SQL transactions that can be called an arbitrary number of times from an arbitrary number of processes. For instance, Subscription provides an AddUser transaction for adding a new user with a given username and password, and a RemoveUser transaction for removing an existing user. (The examples in Figure 5.1 are particular variations of FusionTicket, Twitter, and Betting.) We considered five variations of the robustness problem: the three robustness problems we studied in this chapter along with robustness against `SI` relative to `SER` and against `CC` relative to `SER`. The artifacts are available in a GitHub repository [79].

In the first part of the experiments, we check for robustness violations in bounded-size executions of a given application. For each application, we have constructed a client program with a fixed number of processes (2) and a fixed number of transactions of the corresponding application (at most 2 transactions per process). For each program and pair of consistency models, we check for robustness violations using the reductions to reachability under `SER` presented in §5.4 and §5.5

in the case of pairs of weak consistency models, and the reductions in [44, 45] when checking for robustness relative to `SER`. We check for reachability (assertion violations) using the Boogie program verifier [41]. We model tables as unbounded maps in Boogie and SQL queries as first-order formulas over these maps (that may contain existential or universal quantifiers). To model the uniqueness of primary keys we use Boogie linear types.

Table 5.2 reports the results of this experiment (cells filled with "no"). Five applications are not robust against at least one of the semantics relative to some other stronger semantics. FusionTicket and Twitter were not robust against both `CC` relative to `PC` and `PC` relative to `SI`. Epinions, Subscription, and Vote, were not robust against `CC` relative to `PC`, `PC` relative to `SI`, and `SI` relative to `SER`, respectively. The columns for robustness against `CC` relative to `SI` and against `CC` relative to `SER` can be obtained as the conjunction of results in the other three columns. The wall-clock times for the robustness checks are all under one second, and the memory consumption is around 50 Megabytes.

All the robustness violations we report correspond to violations of the intended specifications. For instance: (1) the robustness violation of Epinions against `CC` relative to `PC` allows two users to update their ratings for a given product and then when each user queries the overall rating of this product they do not observe the latest rating that was given by the other user, (2) the robustness violation of Subscription against `PC` relative to `SI` allows two users to register new accounts with the same identifier, and (3) the robustness violation of Vote against `SI` relative to `SER` allows the same user to vote twice. The specification violation in Twitter was reported in [58]. However, it was reported as violation of a different robustness property (`CC` relative to `SER`) while our work shows that the violation persists when replacing a weak consistency model (e.g., `SI`) with a weaker one (e.g. `CC`). This implies that this specification violation is not present under `SI` (since it appears in the difference between `CC` and `SI` behaviors), which cannot be deduced from previous work.

In the second part of the experiments, we used the technique described in Section 5.7, based on commutativity dependency graphs, to prove robustness. For each application (set of transactions) we considered a program that for each ordered pair of (possibly identical) transactions in the application, contains two processes executing that pair of transactions. Following Remark 5.2, the robustness of such a program implies the robustness of a *most general client* of the application that executes each transaction an arbitrary number of times and from an arbitrary number of processes. We focused on the cases where we could not find robustness violations in the first part. To build

Table 5.2: Results of the experiments. The columns titled X-Y stand for the result of applications robustness against X relative to Y.

Application	Transactions	Robustness				
		`CC-PC`	`PC-SI`	`CC-SI`	`SI-SER`	`CC-SER`
Betting	2	yes	yes	yes	yes	yes
CassandraLock	3	yes	yes	yes	yes	yes
Epinions	8	no	yes	no	yes	no
FusionTicket	3	no	no	no	yes	no
SimpleCurrencyExchange	4	yes	yes	yes	yes	yes
Subscription	2	yes	no	no	yes	no
Twitter	3	no	no	no	yes	no
Vote	1	yes	yes	yes	no	no

the "non-mover" relations M_{WR}, M_{WW}, and M_{RW} for the commutativity dependency graph, we use the left/right mover check provided by the CIVL verifier [99]. The results are reported in Table 5.2, the cells filled with "yes". We showed that the three applications Betting, CassandraLock and SimpleCurrencyExchange are robust against any semantics relative to some other stronger semantics. Epinions, FusionTicket, Subscription and Twitter are robust against `SI` relative to `SER`. Furthermore, Epinions is robust against `PC` relative to `SI` while Subscription is robust against `CC` relative to `PC`. Finally, the Vote application is robust both against `CC` relative to `PC` and against `PC` relative to `SI`. As mentioned earlier, all these robustness results are established for arbitrarily large executions and clients with an arbitrary number of processes. For instance, the robustness of SimpleCurrencyExchange ensures that when the exchange market owner observes a trade registered by a user, they observe also all the other trades that were done by this user in the past.

In conclusion, our experiments show that the robustness checking techniques we present are effective in proving or disproving robustness of concrete applications. Moreover, it shows that the robustness property for different combinations of consistency models is a relevant design principle, that can help in choosing the right consistency model for realistic applications, i.e., navigating the tradeoff between consistency and performance (in general, weakening the consistency leads to better performance).

5.9 Related Work

The consistency models in this chapter were studied in several recent works [60, 59, 69, 142, 52, 145, 50]. Most of them focused on their operational and axiomatic formalizations. The formal definitions we use in this chapter are based on those given in [69, 52]. Biswas and Enea [50] shows that checking whether an execution is `CC` is polynomial time while checking whether it is `PC` or `SI` is NP-complete.

The robustness problem we study in this chapter has been investigated in the context of weak memory models, but only relative to sequential consistency, against RA and PGAS parallel programming models, and TSO and Power weak memory models [113, 66, 55, 51, 74, 73]. In this work, we study the robustness problem between two weak consistency models, which poses different non-trivial challenges. In particular, previous work proposed reductions to reachability under sequential consistency (or `SER`) that relied on a concept of minimal robustness violations (w.r.t. an operational semantics), which does not apply in our case. The relationship between `PC` and `SER` is similar in spirit to the one given by Biswas and Enea [50] in the context of checking whether an execution is `PC`. However, that relationship was proven in the context of a "weaker" notion of trace (containing only program order and read-from), and it does not extend to our notion of trace. For instance, that result does not imply preserving WW dependencies which is crucial in our case.

Some works describe various over- or under-approximate analyses for checking robustness relative to `SER`. The works in [33, 47, 57, 58, 70, 129] propose static analysis techniques based on computing an abstraction of the set of computations, which is used for proving robustness. In particular, [58, 129] encode program executions under the weak consistency model using FOL formulas to describe the dependency relations between actions in the executions. These approaches may return false alarms due to the abstractions they consider in their encoding. Note that in this chapter, we prove a strengthening of the results of [47] with regard to the shape of happens before cycles allowed under `PC`.

5.10 Conclusion

We proposed polynomial time reductions of the problems of checking robustness between weak consistency models to reachability, and therefore, we showed that checking robustness is in principle as hard as checking reachability. We considered three popular weak consistency models: causal consistency, prefix consistency, and snapshot isolation. We also gave a pragmatic technique for proving robustness based on the notion of non-mover relations that can be constructed automatically using SMT solvers. We tested our techniques on realistic programs that model the most intricate parts of distributed applications that are obtained from the standard OLTP benchmark and open source Github projects.

Chapter 6

Behavioral Simulations for Smart Contracts

6.1 Introduction

In this chapter, we are interested in the verification of smart contracts that are not annotated with formal specifications. We propose a technique for verifying unannotated smart contracts via automated semantic comparison against annotated canonical smart contracts. With a notion of comparison that implies substitutability, we can thus amortize the cost of manually annotating the canonical contracts by verifying a vast number of unannotated contracts. Our notion of behavioral refinement relates the input-output behavior of contracts' transactions, i.e., parameters and effects on storage, ignoring internal details like local memory and control flow. By proving that a given contract is a behavioral refinement of another, we guarantee the inheritance of behavioral properties, and in particular that the effects of any sequence of transactions obeys its canonical counterpart's functional properties. Establishing behavioral refinement for unbounded transaction sequences relies on induction. Akin to inductive invariants for safety properties, proofs of behavioral refinement use induction hypotheses called *simulation relations* [127]. Essentially, a behavioral simulation relation identifies states of two contracts such that initial states are related; the same transaction applied to related states yields related states and identical effects; and related states are *observationally equivalent*, i.e., any function applied to both yields identical values.

In §6.2, we outline our approach to synthesize behavioral simulation relations between smart contracts. In §6.3-6.4, we demonstrate an application of behavioral simulation to smart contracts.

Then, in §6.5-6.6, we develop an algorithm for synthesizing behavioral simulation relations. To generate candidate simulation relations, we adopt a paradigm of learning from examples [128]. To verify candidate simulation relations, we adopt a notion of product programs inspired by relational verification [43]. In §6.8, we develop a smart contract benchmark suite including variations of canonical contracts. Finally, in §6.9, we evaluate our approach, verifying functional properties for dozens of unannotated smart contracts. The Empirical evaluation validates our approach by synthesizing simulation relations from multiple variations of each class of canonical contracts. Our implementation can correctly synthesize nontrivial simulation relations for many classes, and integrates off-the-shelf tools for example-guided learning and Solidity smart contracts verification.

6.2 Overview

In this section, we overview the methodology formalized in §6.3 and §6.6 for synthesizing behavioral simulations. We illustrate behavioral refinement on a running example (§6.2.1), describe behavioral simulation for proving refinement (§6.2.2), and demonstrate synthesis on the running example (§6.2.3).

6.2.1 Motivation

We illustrate the concept of behavioral refinement on two contracts implementing an auction (written in the Solidity language of Ethereum), which are partially listed in Figure 6.1. These excerpts focus on the initialization and the bidding parts of an auction. The contract `RefAuction`[1] plays the role of an annotated canonical implementation of an auction (we omit the exact postconditions for brevity) while `Auction` is a particular variation. We generally refer to canonical implementations as *reference (smart) contracts* while variations like `Auction` are called simply *(smart) contracts.*

The fields of `RefAuction` store information about the beneficiary and the ending time of the auction, the current highest bidder and its bid, and the bids of previous highest bidders (the owners of these bids have the right to reclaim them at any point during the auction – for brevity, this functionality is excluded from these excerpts). While the constructor initializes the beneficiary and the ending time of the auction, the `bid` function allows a participant to pose a new bid which is accepted only if it is bigger than the current highest bid and the timeout did not expire. Otherwise, the `bid` function has no effect on the state of the contract – if the condition inside a `require`

[1]Extracted from the documentation page of Solidity [17].

```
// @notice invariant ...
contract RefAuction {
  uint public auctionEnd, highestBid;
  address payable public beneficiary;
  address public highestBidder;
  mapping(address => uint) pendingReturns;
  // @notice postcondition ...
  constructor(uint _bidTime, address payable _benefic) public {
    beneficiary = _benefic;
    auctionEnd = now + _bidTime;
  }
  // @notice postcondition ...
  function bid() public payable {
    require(now <= auctionEnd && msg.value > highestBid);
    if (highestBid != 0)
      pendingReturns[highestBidder] += highestBid;
    highestBidder = msg.sender;
    highestBid = msg.value;
  }
  // @notice postcondition ...
  function PendingReturns() public view returns (uint) {
    return pendingReturns[msg.sender];
  }
  // @notice postcondition ...
  function HighestBid() public view returns (uint) {
    return highestBid;
  }
}
```

Figure 6.1: A canonical auction contract, omitting withdrawal, auction-ending, and other view functions. Implicit variables `now`, `msg.value`, and `msg.sender` yield block timestamps, Ether sent, and callers' addresses.

statement fails, the invocation is *reverted* and is semantically equivalent to skip. This contract also contains several functions that allow to read its fields, in particular a bid that has been superseded by a higher one (function `PendingReturns`) and the highest bid.

The contract `Auction` is a variation that changes the representation of the auction ending time decomposing it into an auction start time and a bidding duration. The handling of revert conditions in the `bid` function is syntactically distinct, but semantically equivalent to the `require` in `RefAuction`.

```
contract Auction {
  uint public auctionStart, biddingTime, highestBid;
  address payable public beneficiary;
  address public highestBidder;
  mapping(address => uint) pendingReturns;

  constructor(uint _bidTime, address payable _benefic) public {
    beneficiary = _benefic;
    auctionStart = now;
    biddingTime = _bidTime;
  }
  function bid() public payable {
    if (now > auctionStart + biddingTime || msg.value <= highestBid)
      revert();
    if (highestBidder != address(0))
      pendingReturns[highestBidder] += highestBid;
    highestBidder = msg.sender;
    highestBid = msg.value;
  }
  function PendingReturns() public view returns (uint) {
    return pendingReturns[msg.sender];
  }
  function HighestBid() public view returns (uint) {
    return highestBid;
  }
}
```

Figure 6.2: A variation of the canonical auction contract in Figure 6.1.

Despite syntactic and state representation differences, every sequence of transactions calling methods of `Auction` has the same effect as if they were calling `RefAuction` instead. This relationship can be stated as `Auction` being a *behavioral refinement* of `RefAuction`, i.e., that its behaviors are subsumed by `RefAuction`. We use the term *behavior* to refer to a summary of the inputs and outcomes, e.g., return values, of a sequence of invocations.

Behavioral refinement is consistent with Liskov's substitutability principle [122], i.e., any contract can be replaced with any of its refinements in any context, as long as a behavior records all the outcomes (effects) which are observable in a context. For the sake of this example, we will focus on return values. Other observable effects which are relevant in a Blockchain environment, e.g., changes on the state of other contracts or Blockchain global variables like the balances of external accounts, are discussed in §6.3.

For instance[2],

$$\texttt{constructor}(5, \texttt{a}) \cdot \texttt{bid}(\texttt{b}, 20) \cdot \texttt{bid}(\texttt{c}, 30) \cdot \texttt{HighestBid}() \Rightarrow 30$$
$$\texttt{constructor}(5, \texttt{a}) \cdot \texttt{bid}(\texttt{b}, 20) \cdot \texttt{bid}(\texttt{c}, 10) \Rightarrow \bot \cdot \texttt{HighestBid}() \Rightarrow 20$$

are two possible behaviors of `Auction` which are also possible when calling methods of `RefAuction` instead (we use the ⊥ return value to signal a reverted `bid` invocation). More generally, refinement holds because the conditions under which a new bid is accepted are semantically the same even though the two contracts use different representations of the ending time. The constructors of these contracts ensure that the two representations are "consistent" in the sense that

$$\texttt{auctionEnd} = \texttt{auctionStart} + \texttt{biddingTime} \tag{6.1}$$

which implies that the timing conditions in function `bid` are equivalent. Note that even though `bid` has no return value, using different conditions for accepting a bid would have been "observable" because of the "getter" method that allows to read the highest bid at any point during an execution.

6.2.2 Behavioral Simulation Relations

Establishing refinement usually relies on an induction argument based on a *(behavioral) simulation relation*, which in our context, is a relation between the states of the two contracts supporting a proof that the reference contract mimics every method invocation of the other contract. The simulation relation supporting such a proof is defined as follows (the fields of `Auction` are prefixed by # to distinguish them from fields of the reference auction having the same name):

$$\begin{aligned}\mathsf{Sim} &\overset{\text{def}}{=} \texttt{auctionStart} + \texttt{biddingTime} = \texttt{auctionEnd} \\ &\wedge\ \texttt{\#beneficiary} = \texttt{beneficiary} \wedge \texttt{\#highestBidder} = \texttt{highestBidder} \\ &\wedge\ \texttt{\#highestBid} = \texttt{highestBid} \wedge \texttt{\#pendingReturns} = \texttt{pendingReturns}\end{aligned} \tag{6.2}$$

The initial states of the two contracts (produced after executing the constructor) are obviously related by Sim and also, given any two states (of `Auction` and `RefAuction`, respectively) related by

[2]For `bid` invocations, the caller identity and the amount of Ether it sends are written as explicit arguments, and the return value of an invocation (e.g., to `HighestBid()`) is written after ⇒. Also, we use small cap letters `a`, `b`, `c` to represent values of type `address`.

Sim, executing an arbitrary invocation in `Auction` can be mimicked by an invocation in `RefAuction` of the same method with the same arguments and return values. Moreover, the states reached at the end of the two invocations are again related by Sim. The latter enables an extension of this proof to an arbitrary number of invocations.

The existence of this simulation relation implies that `Auction` is a behavioral refinement of `RefAuction`, which implies that it satisfies any property of `RefAuction` characterizing its behaviors. Even more, since the simulation relates the states of the two contracts, it also supports deriving valid-by-construction inductive invariants or pre/post-condition annotations for methods. For instance, an inductive invariant of a reference contract (that holds before and after every method invocation) can be used to define a valid-by-construction inductive invariant for any contract that it simulates. In the context of our running example, the following inductive invariant of the reference auction

$$\mathsf{Inv} \overset{\text{def}}{=} \forall i.\mathtt{pendingReturns}[i] \leq \mathtt{highestBid}$$

implies that $\mathsf{Sim} \wedge \mathsf{Inv}$ is an inductive invariant for `Auction`.

6.2.3 Simulation Relation Synthesis

We propose methodology for synthesizing such simulation relations automatically that consists of two parts: a learning procedure for guessing simulation relation candidates from examples (§6.2.3), and using deductive verification for establishing the validity of the inferred candidates (§6.2.3).

Learning Simulations From Examples

To generate candidate simulation relations we use a procedure based on learning from examples, where the goal is learning a (first-order) formula that "separates" a set of positive examples from a set of negative examples, i.e., satisfied by all positive examples and falsified by all negative ones. In our context, examples are pairs of states of the contract and reference contract, respectively. The positive examples must be included in any simulation relation while the negative ones must be excluded from any simulation. Classifying examples as positive or negative enables the re-use of any existing learning algorithm that can produce formulas separating between the two, e.g., [85, 86, 139, 149, 155].

The positive examples are pairs of states obtained by executing the *same* sequence of invocations (with the same arguments) from the initial state of both the contract and the reference contract. Such pairs of states are necessarily included in every simulation under the assumption that contracts are deterministic, which roughly, means that the state reached by a contract when executing a sequence of invocations is unique. These two auction contracts satisfy this determinism assumption (this is rather straightforward when the global variable `now` is assumed to be a constant; otherwise, it is required that any modification of the environment variable `now` is modeled explicitly as an invocation to a fictitious method – see §6.3 for more details). For instance, the following pair of states is obtained by running $\texttt{constructor}(2, \texttt{a}) \cdot \texttt{bid}(\texttt{b}, 10) \cdot \texttt{bid}(\texttt{c}, 20)$ in both contracts (we write only the keys of `pendingReturns` that changed with respect to the initial state):

$$\left(\begin{bmatrix} \texttt{beneficiary} = \texttt{a} \\ \texttt{now} = 0 \\ \texttt{auctionStart} = 0 \\ \texttt{biddingTime} = 2 \\ \texttt{highestBid} = 20 \\ \texttt{highestBidder} = \texttt{c} \\ \texttt{pendingReturns}[\texttt{b}] = 10 \end{bmatrix} , \begin{bmatrix} \texttt{beneficiary} = \texttt{a} \\ \texttt{now} = 0 \\ \texttt{auctionEnd} = 2 \\ \\ \texttt{highestBid} = 20 \\ \texttt{highestBidder} = \texttt{c} \\ \texttt{pendingReturns}[\texttt{b}] = 10 \end{bmatrix} \right) \tag{6.3}$$

We generate positive examples by enumerating invocation sequences and producing the pairs of states reached by executing them in the two contracts.

The definition of negative examples relies on a relation between states which compares return values of read-only methods. As a base case, a negative example is any pair of states that are distinguished by a read-only method, i.e., invoking this method on each of the two states results in different return values. For instance, the following pair of states are distinguished by the `HighestBid` method:

$$\left(\begin{bmatrix} \texttt{beneficiary} = \texttt{a} \\ \texttt{now} = 0 \\ \texttt{auctionStart} = 0 \\ \texttt{biddingTime} = 2 \\ \texttt{highestBid} = 20 \\ \texttt{highestBidder} = \texttt{c} \\ \texttt{pendingReturns}[\texttt{b}] = 10 \end{bmatrix} , \begin{bmatrix} \texttt{beneficiary} = \texttt{a} \\ \texttt{now} = 0 \\ \texttt{\_auctionEnd} = 2 \\ \\ \texttt{highestBid} = 30 \\ \texttt{highestBidder} = \texttt{b} \\ \\ \end{bmatrix} \right) \tag{6.4}$$

The first state is obtained by calling $\texttt{constructor}(2, \texttt{a}) \cdot \texttt{bid}(\texttt{b}, \mathbf{10}) \cdot \texttt{bid}(\texttt{c}, 20)$ in the `Auction` contract while the second one is obtained by calling $\texttt{constructor}(2, \texttt{a}) \cdot \texttt{bid}(\texttt{b}, \mathbf{30}) \cdot \texttt{bid}(\texttt{c}, 20)$ in the reference auction. The difference between the two sequences, i.e., the argument to the second bid, is written in bold font (the last bid in the reference auction sequence is not accepted because it is smaller than the previous one). Such pairs of states should be excluded from any simulation relation because otherwise, the reference contract cannot mimic the invocation of that particular read-only method in the other contract. Going further, any pair of states from which executing the *same* sequence of invocations leads to states that are distinguishable by some read-only method is also a negative example (this again relies on the assumption that contracts are deterministic). For instance, the predecessors of the pair of states in (6.4), reached before making the last bid (i.e., $\texttt{bid}(\texttt{c}, 20)$), which is the same in both contracts, is such an example:

$$\left(\begin{bmatrix} \texttt{beneficiary} = \texttt{a} \\ \texttt{now} = 0 \\ \texttt{auctionStart} = 0 \\ \texttt{biddingTime} = 2 \\ \texttt{highestBid} = 10 \\ \texttt{highestBidder} = \texttt{b} \end{bmatrix}, \begin{bmatrix} \texttt{beneficiary} = \texttt{a} \\ \texttt{now} = 0 \\ \texttt{\_auctionEnd} = 2 \\ \\ \texttt{highestBid} = 30 \\ \texttt{highestBidder} = \texttt{b} \end{bmatrix} \right)$$

As we hinted above, negative examples can also be identified based on invocation sequences, in this case two distinct ones. Therefore, their generation is oblivious to state representations and based on an enumeration of pairs of invocation sequences.

Note that Sim in Equation 6.2 is indeed a separator between the examples described above.

Verifying Simulation Relations

To verify that a simulation candidate is indeed valid we rely on deductive verification. We generate a *simulation-checking* contract with one function for each of the functions common to the input contracts, invoking each version in turn. Figure 6.3 lists an excerpt of this contract for our running example. The inheritance mechanism ensures that each state of `SimulationCheck` is a disjoint union of a state of `Auction` and `RefAuction`, respectively. The simulation-checking contract lists the given candidate simulation relation, in this case Sim in Equation 6.2, as both a pre- and post-condition to each function (as well as a post-condition of the constructor), and asserts that both versions of each function yields the same results. Sim is a valid simulation relation if all the pre/post-conditions and assertions are satisfied by `SimulationCheck`.

This deductive verification step completes the proof that `Auction` is a behavioral refinement of `RefAuction` and that it inherits all its behavioral properties, e.g., a bid is accepted only if it is

```
contract SimulationCheck is Auction, ReferenceAuction {

  // @notice postcondition Sim
  constructor(uint _biddingTime, address payable _beneficiary)
    Auction(_biddingTime, _beneficiary)
    ReferenceAuction(_biddingTime, _beneficiary) public { }

  // @notice precondition Sim
  // @notice postcondition Sim
  function checkBid() public payable {
    r0 = Auction.bid();
    r1 = ReferenceAuction.bid();
    assert (r0 == r1);
  }
}
```

Figure 6.3: Validating the simulation relation Sim.

bigger than every previous bid.

6.3 Behavioral Refinement

The formalization of behavioral refinement between contracts relies on a simple yet universal model of computation, namely labeled transition systems. A *labeled transition system* (LTS) $A = (Q, \Sigma, s_0, \delta)$ over the possibly-infinite alphabet Σ is a possibly-infinite set Q of states with initial state $s_0 \in Q$, and a transition relation $\delta \subseteq Q \times \Sigma \times Q$. The ith symbol of a sequence $\tau \in \Sigma^*$ is denoted τ_i, and ϵ is the empty sequence. An *execution* of A is an alternating sequence of states and transition labels (also called *actions*) $\rho = s_0, a_0, s_1 \dots a_{k-1}, s_k$ for some $k > 0$ such that $\delta(s_i, a_i, s_{i+1})$ for each $0 \leq i < k$. We write $s_i \xrightarrow{a_i \dots a_{j-1}}_A s_j$ as shorthand for the subsequence $s_i, a_i, ..., s_{j-1}, a_{j-1}, s_j$ of ρ. (in particular $s_i \xrightarrow{\epsilon} s_i$). The projection $\tau|\Gamma$ of a sequence τ is the maximum subsequence of τ over the alphabet Γ. This notation is extended to sets of sequences as usual. A *trace* of A is the projection $\rho|\Sigma$ of an execution ρ of A. The set of traces of an LTS A is denoted by $T(A)$. An LTS is *deterministic* if for any state s and sequence $\tau \in \Sigma^*$, there is at most one state s' such that $s \xrightarrow{\tau} s'$.

A contract is interpreted as an LTS whose traces represent sequences of invocations to the contract's methods together with their inputs and observable outcomes. A typical example of an observable outcome is the return value, which can be read through invocations from other contracts. Other examples include effects like gas consumption, changes on the state of other contracts, changes on Blockchain global variables like the balances of external accounts, etc. To simplify the technical exposition, we will mostly focus on return values but this is not a limitation. This LTS interpretation is used to formalize and reason about the soundness of our methodology. It is not intended to be constructed explicitly.

Essentially, the states of the LTS are composed of an *internal* part represented as assignments to the contract's fields and the balance of the address at which the contract is deployed, and an *environment* part represented as assignments to environment variables, e.g., `now` in Figure 6.1, which influence the contract's behavior. The transitions represent invocations to the contract's methods or updates of the environment variables, e.g., increasing the value of `now`[3]. The labels record method names, arguments, and observable outcomes. For uniformity, updates of environment variables are modeled as invocations to some fictitious methods.

Formally, an *invocation label* $m(\vec{u})$ is a method name m along with a vector $\vec{u}$ of argument values. An *operation label* $\ell = m(\vec{u}) \Rightarrow v$ is an invocation label $m(\vec{u})$ along with a return value v. We assume a fixed, but unspecified, domain Vals of argument or return values. Vals includes a distinguished return value $\bot$ associated to invocations that revert. We use $\mathsf{inv}(\ell)$ to refer to the invocation label in an operation label ℓ. This notation is extended to sequences or sets of operation labels as expected. An *interface* Σ is a set of operation labels over a finite set of method names. We use $\Sigma^{\checkmark}$ to denote the subset of Σ that excludes operation labels with $\bot$ as a return value, and $\mathsf{Meths}(\Sigma)$ to denote the method names in Σ.

Definition 6.1. *A* (smart) contract *is an LTS* $C = (Q, \Sigma, s_0, \delta)$ *over an interface* Σ.

Example 6.1. *Figure 6.4 pictures a fragment of the LTS interpretation of* `RefAuction`*. This LTS contains an initial state from where a number of transitions corresponding to constructor invocations are enabled. These different transitions correspond to different sets of arguments passed to the constructor. As mentioned above, a state of this LTS consists of a valuation of all the fields of* `RefAuction`*, e.g.,* `beneficiary` *and* `_auctionEnd`*, the balance of the address at which this contract is deployed, written as* `balance[this]`*, and the environment variable* `now`*. Invocations that revert, e.g., bidding 10 when the highest bid is 20, marked using the* $\bot$ *return value, or invocations to*

[3]This LTS can be thought of as a composition between an LTS defining the evolution of the variables controlled by the contract and an LTS defining the evolution of the environment variables (whose states are valuations of these variables). The states of the two LTSs share the valuation of the environment variables (read by the first LTS and updated by the second).

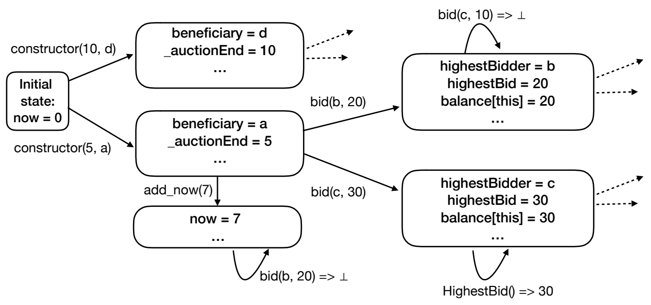


Figure 6.4: A fragment of the LTS interpretation of `RefAuction`. Boxes represent states and arrows represent transitions. The representation of states emphasizes the fields changed by the incoming transition.

read-only methods like `HighestBid` *are represented as self-loops. Transitions also represent updates of* `now`*, e.g., increasing its value by 7.*

Modeling updates of environment variables as *labeled* LTS transitions is important to ensure that the resulting LTS is deterministic. For instance, assuming that such updates are ϵ transitions in the LTS interpretation of `RefAuction` (Figure 6.4), the singleton sequence $\texttt{constructor}(5, \texttt{a})$ leads to two distinct states, where $\texttt{now} = 0$ and $\texttt{now} = 7$, respectively. Determinism is important for the soundness of our learning procedure (see §6.5).

Remark 6.1. The notion of contract in Definition 6.1 considers the return value as the only observable outcome of an invocation. This notion can be extended to include other observable effects by enriching the structure of transition labels. For instance, it is quite frequent that the methods of a contract invoke Solidity primitives like `send` for transferring Ether, or methods of other contracts, and possibly even check their return values. An invocation to such a method m can be represented by a transition labeled by $m(\vec{u}) \Rightarrow I, v$ where $\vec{u}$ and v are the arguments and return value of this invocation, and I is a sequence of operation labels corresponding to the "internal" calls made during this invocation (e.g., a call to `send` with its arguments and return value).

The standard refinement relation between two LTSs is defined as the inclusion between the set of traces produced by the two LTSs. For practical reasons, we consider an extension of this notion that allows a contract to refine another even if (1) it has a larger interface (it defines a larger set of methods) or (2) invocations revert more often (this is sound since any update of a reverted invocation is discarded).

```
contract PAX {
  mapping(address => uint256) public balances;
  address public owner;
  constructor() public {
    owner = msg.sender;
  }
  function mint(address to, uint val) public {
    require(msg.sender == owner);
    balances[to] = balances[to] + val;
  }
  function transfer(address to, uint val) public {
    require(val <= balances[msg.sender]);
    balances[msg.sender] = balances[msg.sender] - val;
    balances[to] = balances[to] + val;
  }
  function transferOwnership(address _newOwner) public {
    require(msg.sender == owner);
    owner = _newOwner;
  }
}
```

Figure 6.5: A contract managing a set of tokens.

For instance, Figure 6.5 lists several functions of a contract called `PAX`[4] that allows an owner to mint some set of tokens for some specific address (function `mint`), transfer tokens between different addresses (function `transfer`), and change ownership (function `transferOwnership`). Also, Figure 6.6 lists an excerpt from the `ERC20` reference contract in OpenZeppelin [13]. `PAX` defines the method `transferOwnership` that does not occur in `ERC20` and the method `mint` in `PAX` can revert if it is not called by the owner while its counterpart in `ERC20` can not. We consider `PAX` to be a refinement of `ERC20` because any sequence of non-reverted invocations to methods of `PAX` that exist in both is admitted by `ERC20` as well (when looking only at arguments and return values as in the LTS interpretation). This extension of the notion of refinement also allows that a contract is a refinement of several reference contracts. For instance, `PAX` is also a refinement of the contract `Ownable` from the OpenZeppelin library, which implements an ownership mechanism.

Definition 6.2. *A contract C_1 over interface Σ_1* refines *another contract C_2 over interface Σ_2 when $T(C_1)|\Sigma_2^{\checkmark} \subseteq T(C_2)$.*

[4]A variation of a contract extracted from [14].

```
contract ERC20 {
  mapping(address => uint256) public balances;
  function mint(address to, uint val) public {
    balances[to] = balances[to] + val;
  }
  function transfer(address to, uint val) public {
    require(val <= balances[msg.sender]);
    balances[msg.sender] = balances[msg.sender] - val;
    balances[to] = balances[to] + val;
  }
}
```

Figure 6.6: An excerpt from the `ERC20` reference contract.

6.4 Behavioral Simulations

The standard methodology for proving refinement is based on simulation relations, which are the analog of inductive invariants in proofs of safety. Simulation relations enable an induction scheme to prove inclusion of traces which generally can go forward, from initial states towards end states, or backward, from end states towards initial states. While both types of reasoning, forward or backward, are sound for proving refinement, forward reasoning is easier to automate while being complete for proving refinement of *deterministic* LTSs only [126]. Since smart contracts are most often deterministic, we focus on forward reasoning in this work.

Let $C_1 = (Q_1, \Sigma_1, s_0^1, \delta_1)$ and $C_2 = (Q_2, \Sigma_2, s_0^2, \delta_2)$ be two contracts. A simulation relation R relates states of C_1 and C_2, respectively, in particular their initial states, such that any transition of C_1 from a state q (corresponding to a non-reverted invocation) can be reproduced by C_2 from a state related by R to q (i.e., C_2 has a transition with the same label from the state related by R to q). The end states of the two transitions in C_1 and C_2, respectively, must again be related by R[5]. Formally, a relation[6] $R \subseteq Q_1 \times Q_2$ is called a *(behavioral) simulation* from C_1 to C_2 iff $R(s_0^1, s_0^2)$

[5]This is a variation called *forward* simulation relation, which corresponds to the forward reasoning mentioned above, from initial states towards end states. In general, proving refinement may also require establishing the existence of a *backward* simulation, which is similar but the preservation of steps is defined in the reverse direction, i.e., for any transition of C_1 *leading* to a state q and any state q' of C_2 related by R to q, there exists a transition of C_2 with the same label leading to q' and starting from a state related by R to the source state of C_1's transition.

[6]For readability, we write binary relations as predicates, e.g., $R(s_1, s_2)$ instead of $(s_1, s_2) \in R$.

and for all $s_1, s_1' \in Q_1$, $a \in \Sigma_1$, and $s_2 \in Q_2$,

$$s_1 \xrightarrow{a}_{C_1} s_1' \wedge R(s_1, s_2) \implies \exists s_2' \in Q_2.\ s_2 \xrightarrow{a|\Sigma_2^{\checkmark}}_{C_2} s_2' \wedge R(s_1', s_2')$$

We adapted the standard definition of a simulation relation to take into account the restriction to non-reverted invocations and that Σ_1 is not necessarily included in Σ_2. Transitions with labels that exist only in C_1 should be mimicked by ϵ (skip) transitions of C_2.

Example 6.2. *The relation* $\mathsf{Sim}_1 \overset{def}{=} \#$`balances` $=$ `balances` *is a simulation relation from* `PAX` *in Figure 6.5 to* `ERC20` *in Figure 6.6 (the field* `balances` *of* `PAX` *is prefixed by #). This holds because in particular, executing a method of* `PAX` *which is not defined by* `ERC20` *does not affect* `balances`*.*

The following statement follows from standard results relating simulation relations and refinement [126].

Theorem 6.1. *If there exists a simulation relation from a contract C_1 to a contract C_2, then C_1 refines C_2. Moreover, if C_1 refines C_2 and C_2 is deterministic, then there exists a simulation relation from C_1 to C_2.*

Theorem 6.1 reduces refinement proofs to synthesizing simulation relations. The next section shows that a simulation relation can be seen as a "separator" between two sets of pairs of states (of the two contracts), analogous to an inductive invariant being a separator between "safe" and "unsafe" states. This enables a learning from examples approach for computing simulation relation candidates.

6.5 Learning Simulations From Examples

We describe a learning procedure for simulation relations which relies on a classification of pairs of states (of the two contracts) as *positive*, included in every simulation, or *negative*, excluded from every simulation. This classification is based on a notion of *observational distinguishability* between states which holds when two states can be distinguished by return values of read-only methods. We say that an invocation label $m(\vec{u})$ is *read-only* in a contract C when it is enabled in every state, i.e., for any trace $\sigma_1 \cdot \sigma_2 \in T(C)$, there exists $v \in \mathsf{Vals}$ such that $\sigma_1 \cdot (m(\vec{u}) \Rightarrow v) \cdot \sigma_2 \in T(C)$, and it does not enable other invocations, i.e., for any value $v \in \mathsf{Vals}$ and trace $\sigma_1 \cdot (m(\vec{u}) \Rightarrow v) \cdot \sigma_2 \in T(C)$, we have that $\sigma_1 \cdot \sigma_2 \in T(C)$ as well. A method m is *read-only* in a contract C when every invocation label $m(\vec{u})$ is read-only. For instance, the methods `PendingReturns` and `HighestBid` of the contract `RefAuction` in Figure 6.1 are read-only, while `bid` is not read-only.

Let C_1 and C_2 be two contracts over interfaces Σ_1 and Σ_2, respectively. A method $m \in \mathsf{Meths}(\Sigma_1) \cap \mathsf{Meths}(\Sigma_2)$ is called an *observation method* when it is read-only in both C_1 and C_2. Given a set of observation methods Obs, two states s_1 and s_2 of C_1 and C_2, respectively, are *(observationally) distinguishable* w.r.t. Obs, denoted by $s_1 \bowtie_{\mathsf{Obs}} s_2$, if

$$\exists m \in \mathsf{Obs}, \vec{u} \in \mathsf{Vals}^*, v \in \mathsf{Vals}.\ s_1 \xrightarrow{m(\vec{u})\Rightarrow v}_{C_1} s_1 \wedge \neg s_2 \xrightarrow{m(\vec{u})\Rightarrow v}_{C_2} s_2$$

We will omit the set of methods Obs from the notations when they are not important or understood from the context.

Example 6.3. *`PendingReturns` and `HighestBid` in Figure 6.1 are observation methods for the pair of contracts `Auction` and `RefAuction`. The pair of states in Equation 6.4 are distinguishable with respect to these two observation methods (`HighestBid` in particular).*

The following result shows that any pair of distinguishable states is excluded from any simulation. It follows from an instantiation of the definition of a simulation on transitions corresponding to observation method invocations.

Lemma 6.1. *Let C_1 and C_2 be two contracts and Obs a set of observation methods. For any simulation R from C_1 to C_2,*

$$s_1 \bowtie_{\mathsf{Obs}} s_2 \implies \neg R(s_1, s_2)$$

We define two relations P and N over states of C_1 and C_2 representing positive and negative examples for simulation relations, respectively:

$$P(s_1, s_2) : \exists \sigma \in (\Sigma_2^{\checkmark})^*.\ s_0^1 \xrightarrow{\sigma}_{C_1} s_1 \wedge s_0^2 \xrightarrow{\sigma}_{C_2} s_2$$
$$N(s_1, s_2) : \exists s_1', s_2',\ \exists \sigma \in (\Sigma_2^{\checkmark})^*.\ s_1 \xrightarrow{\sigma}_{C_1} s_1' \wedge s_2 \xrightarrow{\sigma}_{C_2} s_2' \wedge s_1' \bowtie s_2'$$

where s_0^1 and s_0^2 are the initial states of C_1 and C_2, respectively. This classification is sound under the assumption that C_2 is deterministic. For negative examples, assuming by contradiction that $(s_1, s_2) \in N$ is included in a simulation relation, the state s_2' reached by C_2 when mimicking some sequence of invocations σ of C_1 should "simulate" the corresponding state s_1' of C_1. However, this cannot be the case since the two states are distinguishable (by Lemma 6.1).

Theorem 6.2. *For any simulation relation R from a contract C_1 to a deterministic contract C_2, we have that:*

$$P \subseteq R \subseteq \neg N.$$

Example 6.4. *Positive and negative examples for the pair of contracts* `Auction` *and* `RefAuction` *(listed in Figure 6.1) are given in Equation 6.3 and Equation 6.4, respectively.*

The reverse of Theorem 6.2 does not hold, i.e., there exist relations R that separate P from N and that are not simulation relations. For instance, if the set of observation methods `Obs` is empty, then N is also empty. However, not every superset of P satisfies the inductiveness requirement of a simulation relation. This is similar to the fact that not every superset of the reachable set of states in a program is an inductive invariant. This source of incompleteness can be removed by adapting the approach used in the ICE framework [85] for inductive invariant synthesis.

Theorem 6.2 implies that there exists no simulation relation when the set of positive and negative examples intersect.

Corollary 6.1. *If $P \cap N \neq \emptyset$, then there exists no simulation from C_1 to C_2, provided that C_2 is deterministic.*

For deterministic contracts where the return value of an invocation in a given state is unique, positive and negative examples can be represented precisely using *invocation sequences*. This enables a procedure for enumerating such examples which consists in enumerating (pairs of) invocation sequences and which is oblivious to state representations.

A contract C is *return-value deterministic* if it is deterministic and for any method m, arguments $\vec{u}$, and admitted trace $\sigma \in T(C)$, there is a single label $m(\vec{u}) \Rightarrow v$ such that $\sigma \cdot (m(\vec{u}) \Rightarrow v) \in T(C)$. Determinism does not imply uniqueness of return values. For instance, an extension of `Auction` (Figure 6.1) with a read-only method `foo` that returns a *random* value, computed using `block.difficulty` for instance, remains deterministic. The following result shows that states of return-value deterministic contracts can be represented precisely using invocation sequences.

Lemma 6.2. *For any return-value deterministic contract C, the following holds:*

$$\forall s, s'.\ s_0 \xrightarrow{\sigma} s \wedge s_0 \xrightarrow{\sigma'} s' \wedge \mathit{inv}(\sigma) = \mathit{inv}(\sigma') \implies s = s'$$

Based on Lemma 6.2, each positive example can be represented by a single invocation sequence (the pair of states being reproducible by running this sequence of invocations in both contracts) and each negative example can be represented by two invocation sequences, each sequence representing a state in one of the two contracts. Also, checking that a pair of states is a negative example reduces to checking whether by running the same (possibly-empty) sequence of invocations on the two states, irrespectively of the return values, leads to two states which are distinguishable. This is sound under the return-value determinism assumption.

The classification of simulation examples we presented above makes it possible to leverage off-the-shelf learning algorithms that compute formulas that are satisfied by positive examples and falsified by negative ones, e.g., [85, 86, 139, 149, 155], up to a bounded enumeration of such examples. The problem of checking whether such a formula is a valid simulation relation is discussed in the next section.

6.6 Verifying Simulations

We reduce the problem of verifying that a simulation candidate is valid to checking that it is an inductive invariant for a composition of contracts, which is formalized using a slight variation of the standard product construction for their LTS interpretations. Therefore, the *product* $C_1 \times C_2$ of two contracts is defined as follows: the states are pairs of states of C_1 and C_2, respectively, and a state (s_1, s_2) can perform a transition labeled by $a \in \Sigma_1^{\checkmark}$ to one of the following states:

- (s_1', s_2') if s_1 and s_2 can perform a transition labeled by a to s_1' and s_2', respectively
- (s_1', s_2) if $a \notin \Sigma_2^{\checkmark}$ and s_1 can perform a transition labeled by a to s_1', and
- a fail state $\lightning$ if $a \in \Sigma_2^{\checkmark}$ and only s_1 can perform an a transition.

The second case is required for simulation relations towards reference contracts that have a smaller interface while the last case makes it possible to detect invalid simulation candidates. Note also that $C_1 \times C_2$ excludes transitions corresponding to reverted invocations of C_1. An *inductive invariant* for a contract $C = (Q, \Sigma, s_0, \delta)$ is a set of states I such that (1) $s_0 \in I$ and (2) if $s \in I$ and $s \xrightarrow{a} s'$, for some symbol a, then $s' \in I$. The following theorem shows that any inductive invariant of the product (that does not contain the fail state) is also a simulation relation. The reverse holds when C_2 is deterministic.

Theorem 6.3. *Let C_1 and C_2 be two contracts. If R is an inductive invariant for $C_1 \times C_2$ such that $\lightning \notin R$, then R is simulation from C_1 to C_2. Moreover, if C_2 is deterministic and R is a simulation*

```
contract A {
  ...
  function foo(uint x) public view returns (uint) { require(x>42); ... }
  function bar() public view returns (uint) { ... }
}
contract B {
  ...
  function foo(uint x) public view returns (uint) { ... }
}
contract AxB is A, B {
  ...
  function sync_foo(uint x) public {
    r0 = A.foo(x);
    r1 = B.foo(x);
    require(r0 != ⊥);
    assert (r0 == r1);
  }
  function sync_bar() public {
    A.bar();
  }
}
```

Figure 6.7: A contract `AxB` representing the product of the LTS interpretations of two contracts `A` and `B`.

from C_1 to C_2, then R is an inductive invariant for $C_1 \times C_2$ and $\lightning \notin R$.

In the following, we discuss a concrete instantiation of the results above that relies on source code instead of LTS interpretations. The most important point is defining a contract that represents the product of the LTS interpretations of two contracts. As hinted in §6.2.3, such a contract can be defined using the inheritance mechanism of Solidity. The more subtle issues are related to enforcing transitions with the same label, since the label includes an invocation *and* a return value, and dealing with reverted invocations and methods that are defined in only one of the two contracts.

We explain these issues using the contracts `A` and `B` in Figure 6.7, where `B` is intended to simulate `A` (their fields are omitted). The method `foo` is defined in both contracts, but `A`'s version contains a `require` that may revert certain invocations, while the method `bar` is defined only in contract `A`. Note that methods defined only in `B` can be ignored while checking whether it simulates another contract.

The contract `AxB` is used to represent the product of the LTS interpretations of `A` and `B`. Since `foo` is defined in both contracts, the method `sync_foo` represents synchronous invocations of `foo` in `A` and `B` while also ensuring equality of return values, unless `foo` fails in `A`. Transitions of the product corresponding to `bar` invocations are represented using the method `sync_bar`. If `AxB` verifies the assertion, then its LTS interpretation restricted to invocations of `sync_foo` and `sync_bar` is the product of the LTS interpretations of `A` and `B`. Note that `AxB` can fail the assertion although `B` is deterministic and it simulates `A`. This is possible when `A` is *not* return-value deterministic, e.g., `foo` can return two values in both `A` and `B` when executed from the initial state.

By Theorem 6.3, if a relation R between states of `A` and `B` is an inductive invariant of `AxB` restricted to `sync_foo` and `sync_bar` (it holds before and after every invocation) and `AxB` verifies the assertion, then R is simulation from `A` to `B`.

Remark 6.2. This construction can be extended to handle certain specificities of Solidity. For instance, to deal with payable functions like `bid` in the auction contracts from Figure 6.1, it is sufficient to introduce a ghost variable in the reference contract that tracks the value of the balance (i.e., adding the amount in `msg.value`). Then, a simulation relation relates this ghost variable and the balance of the simulated contract instead of the two balances. Also, to establish the fact that a reference contract invocation makes the same "external " calls (to Solidity primitives like `send` or to other contracts) as the invocation of the contract it simulates (see Remark 6.1) we rely on auxiliary variables that record the sequence of calls with their arguments in each of the two invocations. We then *assert* the equality between these auxiliary variables and *assume* that these calls have the same return values. This models the fact that the two contracts refine one another when placed in the same context where the environment produces the same responses.

6.7 Implementation

In this section we describe an implementation of our methodology for Solidity smart contracts. Our implementation consists of four main components: an *example generator*, an example-guided *synthesizer*, a *blockchain oracle*, and a deductive *verifier*. As input, our implementation requires a pair of Solidity smart contracts with overlapping function signatures, and parameters to limit example generation, including the sets of values to use for transaction parameters, and the number of contract states to explore.

Given these inputs, the example generator provides the synthesizer with positive and negative examples, where each example corresponds to a pair of contract states. In turn, the synthesizer provides the verifier with a candidate simulation relation separating positive and negative examples. Since examples correspond to contract states on an Ethereum blockchain, the synthesizer relies on an oracle to evaluate expressions on examples. Finally, the verifier validates candidate simulation relations.

While this simple scheme sufficed for our empirical study, in principle, the selection of input parameters could be automated in a refinement loop from spurious verifier counterexamples, i.e., following *counterexample guided inductive synthesis* [157]. Furthermore, although we assume the annotated contract against which the given unannotated contract is compared is identified a priori, in principle this identification might be performed, e.g., via machine learning classifiers.

6.7.1 Example Generation

Our example generator executes transaction sequences on the Ganache [11] personal blockchain for Ethereum using the Web3 Ethereum JavaScript API [19] and Solidity compiler [21]. Given limits transaction parameters, e.g., small sets of integer and address values, the example generator systematically explores every transaction sequence in lexicographic order up to the given threshold on the number of contract states. For each state we record as observations the return values for each read-only (view) function over the given parameter limits. In case the states reached in some transaction sequence yield different observations, we return the transaction sequence and observations as a counterexample refuting simulation. Otherwise, the example generator yields positive and negative examples according to §6.5.

Two notable issues that the example generator must overcome are potential nondeterminism, e.g., due to account creation and transaction block mining, and controlling transaction parameters, e.g., the message *sender* parameter. While the former can be managed via parameters to Ganache, the latter required instantiating auxiliary contracts at various addresses to invoke target functions - effectively setting the sender to the auxiliary contract's address.

6.7.2 Synthesis

Our synthesizer component extends the Precondition Inference Engine (PIE) [139], a tool which learns a set of *features*, i.e., atomic predicates, (and a Boolean combination of these features) separating positive and negative examples by enumerating candidate features of increasing complexity. We extend PIE along two principle axes. First, we extend its grammar to include types and operations to handle Solidity language features like addresses, arrays, and maps. Second, instead of concrete examples on which to evaluate candidate features, we make examples *symbolic*, and delegate evaluation of features on examples to a blockchain oracle.

As an optimization we provide the synthesizer with a set of *seed features* generated from the given pair of contracts. Intuitively, the seed features correspond to equalities between terms over the respective contracts' fields that are likely to hold. For instance, when contracts each have a read-only (view) function f which evaluates terms t_1 and t_2, respectively, we generate the equality $t_1 = t_2$. While this is not generally feasible for view functions with complex control flow, it is useful in practice, since many view functions have simple bodies, e.g., a single return statement.

6.7.3 Verification

Our verifier consists of the reduction from simulation checking to deductive verification, described in §6.6, along with the solc-verify verifier [97], which in turn reduces Solidity contract verification to Boogie verification (and ultimately SMT solving). We have contributed only GitHub issues and feature requests to solc-verify.

Besides the nuances described in §6.6 relating to effects on global state, e.g., balances, verifying the simulation-checking contract with solc-verify involved one key nuance regarding modular verification. In particular, while invocations to the annotated reference contract can be verified *modularly*, i.e., using only its pre- and post-conditions, invocations to the unannotated contract are verified *inline*, i.e., explicitly reasoning about the statements in its implementation. Besides helping to virtualize potentially-conflicting global effects between the two invocations, such modularity generally improves tractability.

6.8 Case Study of Solidity Smart Contracts

In this section we outline our case study of Solidity smart contracts, including collection methodology, a partial taxonomy, and an analysis of syntactic similarities. Our starting points for sourcing canonical contracts included the Solidity documentation [21], the Etherscan block explorer and analytics platform [10], the State of the DApps curated directory for decentralized applications [18], and the OpenZepplin contracts library [13].

Contract	Source	Variations
Auction	Ethereum	3
CrowdSale (token offering)	OpenZepplin	3
ERC 20 (token)	OpenZepplin	5
ERC 165 (interface detection)	OpenZepplin	4
ERC 721 (non-fungible token)	OpenZepplin	3
Escrow (payment)	OpenZepplin	3
Gambling	dice2.win	1
LifeCycle (life cycle)	OpenZepplin	5
Lottery	Etherscan	1
MultiSigWallet	ConsenSys	1
Ownable (ownership)	OpenZepplin	5
Roles (access control)	OpenZepplin	5
Voting	Etherscan	2
	Total	41

Figure 6.8: Collection of smart contracts.

A first observation is that a vast number of contracts on the, e.g., Ethereum blockchain are variations on a relatively-small number of canonical contacts like those listed in the first column in Figure 6.8. We found that more than half of the 47 398 contracts extracted from the Ethereum blockchain and studied in [78], which cover each of the eighteen Ethereum application categories from State of the DApps, contain keywords associated with these canonical contracts. This finding seems consistent with common practice, since standardization mechanisms such as Ethereum Request for Comment (ERC) are widely used.

In order to use these canonical contracts as targets for our verification methodology, we manually annotated them with full functional specifications, and verified the annotations with solc-verify [97].

To source contract variations, we collected contracts from Etherscan, as well as popular Blockchain platforms including Moloch Ventures [12], 0xcert [6], Sirin Labs [15], Bit Nation [8], and Crypto Kitties [7]. Overall we collected a set of 43 unannotated contracts, 41 of these contracts are categorized in Figure 6.8 based on which canonical contract they implement. The remaining two contracts are referred to as *multi-contracts* as they simultaneously implement multiple canonical contracts. The collected contracts can be found at [16].

Finally, to assess the need for the automated synthesis procedure described in §6.5, we considered weaker syntactic approaches with varying degrees of sophistication. For example, simply considering the conjunction of equalities between fields with the same names could work for simple single-field contracts, like ownership. In case contracts renamed fields, some field-name similarity heuristic would be required. For contracts with multiple fields, more sophisticated field-matching heuristics would be required, and so on. Then there are contracts whose simulation involves arithmetic expressions, further complicating heuristics. While the current generation of smart contracts we've studied are relatively simple, future generations could render such heuristics fairly useless. As noted in §6.11, our approach is relatively complete, and, as demonstrated in §6.9, capable of synthesizing simulations for many non-trivial contracts.

6.9 Experimental Evaluation

In this section we outline an empirical study of our automated verification approach applied to the Solidity smart contracts described in §6.8 using the implementation described in §6.7. We are able to run our tool on all contracts from Figure 6.8 except MultiSigWallet and Gambling, which require generating non-primitive transaction parameters, including addresses of deployed token contracts and components of cryptographic signatures.

The overview of Figure 6.12 summarizes our results, listing the generated simulation relations (omitting atomic terms) and verification outcomes. Each row, labeled $c \times n$ corresponds to n unannotated contracts compared against one canonical annotated contract c, e.g., auction $\times$ 3 corresponds to 3 distinct unannotated auction contracts compared with one canonical auction contract. (The rows labeled multi-i $\times$ 3 are exceptions; in these cases we consider one unannotated contract compared against 3 distinct canonical contracts, corresponding to cases of multiple inheritance/interfaces.) In all but 3 cases we are able to generate plausible candidate simulation relations, and in all but 3 cases we are able to verify these relations – see §6.9.1.

In the "simulation relations" column, we list the learned simulation relations in prefix notation, omitting atomic terms, i.e., contract fields and constants. In the verified column, we list the number of canonical-and-unannotated-contract pairs for which a candidate simulation relation was:

- computed and verified, e.g., $\mathsf{T} \times 3$ in the auction row indicates success for 3 contract pairs;
- computed but not verified, e.g., $\mathsf{F} \times 1$ in the crowdsale row indicates a candidate simulation relation our implementation did not verify; and

contracts	simulation relations	verified
auction $\times$ 3	$(\wedge(\wedge(\wedge(\wedge(\wedge(=)(=))(=))(=))(=))(=\ (+)))\times 3$	T×3
crowdsale $\times$ 3	$(\wedge(\wedge(\wedge(\wedge(=)(=))(=))(=))(=))\times 3$	F×1, T×2
erc165 $\times$ 4	$(=)\times 4$	T×4
erc20 $\times$ 5	$(\wedge(=)(=))\times 3$	T×3, $\bot\times 2$
erc721 $\times$ 3	$(\wedge(\wedge(\wedge(=)(=))(=))(=))\times 3$	T×3
escrow $\times$ 3	$(\wedge(=)(=))\times 3$	T×3
finalizable $\times$ 2	$(\wedge(=)(=))\times 2$	T×2
lottery $\times$ 1	$(\wedge(=)(=))\times 1$	F×1
multi-1 $\times$ 3	$(\wedge(=)(=))\times 1$, $(=)\times 1$	T×2, $\bot\times 1$
multi-2 $\times$ 3	$(\wedge(=)(=))\times 2$, $(=)\times 1$	T×3
ownable $\times$ 4	$(=)\times 4$	T×4
pausable $\times$ 3	$(\wedge(=)(=))\times 3$	T×3
signer-role $\times$ 2	$(=)\times 2$	T×2
voting $\times$ 2	$(\wedge(=)(=))\times 2$	T×2
whitelisted $\times$ 3	$(\wedge(=)(=))\times 3$	F×1, T×2

Figure 6.9: Summary of results. **Overview**: generated simulation relations (atomic terms omitted) and verification outcomes.

- not computed, e.g., $\bot \times 2$ in the erc20 row indicates 2 pairs for which our implementation did not compute plausible candidate relations.

Our approach synthesizes simulation relations which are notably simpler than the inductive invariants which would be required to verify the functional properties of unannotated contracts by other means. For example, the inductive invariants for typical auction contracts would require disjunctions over auction phases, e.g., active vs. completed, while simulation relations between typical auction contracts need only conjunctions of equalities (see Figure 6.12). Previous works on relational verification make the same observation [80, 43].

For each phase we summarize runtimes, in seconds. Distributions with mean μ, standard deviation σ, and population count n are represented as $\mu \pm \sigma : k$, where σ is omitted when 0, and k is omitted when equal to the subject count n of the row labeled $c \times n$. Among the three phases, synthesis generally takes much longer, e.g., minutes, than example generation, e.g., seconds, and verification, e.g., one second.

contracts	transactions	traces	states	positive	negative	time
auction $\times$ 3	174	47	78	47	370	33$\pm$1.6
crowdsale $\times$ 3	99.3$\pm$11.5	25	34	12.7$\pm$8.1	330	20$\pm$1
erc165 $\times$ 4	26	7	10	7	36	5.7$\pm$0.1
erc20 $\times$ 5	287.6$\pm$391.3	21$\pm$3.5: **3**	38$\pm$6.9: **3**	12.6$\pm$11.8	60$\pm$54.8	52.5$\pm$59.3
erc721 $\times$ 3	154	39	74	31.7$\pm$12.7	100	121.6$\pm$1.1
escrow $\times$ 3	76	19	34	19	100	10.2$\pm$0.1
finalizable $\times$ 2	28	7	10	6	28	4.4$\pm$0.1
lottery $\times$ 1	176	33	62	33	370	20.7
multi-1 $\times$ 3	212$\pm$322.2	7: **2**	10: **2**	3$\pm$2.6	14$\pm$14	53$\pm$66.2
multi-2 $\times$ 3	50$\pm$41.6	13$\pm$10.4	22$\pm$20.8	11$\pm$12.2	47.3$\pm$46.1	24.8$\pm$19.6
ownable $\times$ 4	26	7	10	5	28	5$\pm$0.1
pausable $\times$ 3	26	7	10	4	14	4.5$\pm$0.1
signer-role $\times$ 2	26	7	10	5	30	6.6$\pm$0.1
voting $\times$ 2	34	11	10	11	84	13.1$\pm$0.9
whitelisted $\times$ 3	66$\pm$8	17$\pm$2	30$\pm$4	15$\pm$1	118$\pm$39	10.8$\pm$0.9

Figure 6.10: **Example Generation**: counting blockchain transactions executed, transaction sequences (traces), states, and generated examples.

6.9.1 Cases Where Simulation Was Not Proved

Our implementation only failed to compute candidate simulation relations in 3 cases. However, each failure is due to the discovery of genuine counterexamples to simulation (and refinement). Counterexamples arise in 2 out of 5 ERC-20 variations and in the multi-1 contract which simultaneously implements three canonical contracts: ERC-20, Ownable, and Pausable.

The first counterexample arises due to the *transferFrom* function of ERC-20. The canonical contract subtracts the transferred amount from the sender balance before adding it to the receiver balance, reverting when the subtraction underflows, while the variation contract does the reverse. Thus after executing the following transactions:

a_1: approve(a_2, 2); a_2: transferFrom(a_1, a_1, 2)

the function allowance(a_1, a_2) returns 2 in the first case, since a_2's allowance has not decreased, but 0 in the second. Since the *transferFrom* function is present in the ERC-20 token standard [20] this counterexample corresponds to a vulnerability of the unannotated contract.

The remaining two cases arise from ERC-20's *decreaseAllowance* function. While the canonical contract reverts the transaction if the requested decrease is greater than the current allowance, the

contracts	fields	seeds	terms	queries	time
auction × 3	15	5	7	10914	2561.8±11.3
crowdsale × 3	12	4.7±0.6	5	659.3±556.6	345.3±135.7
erc165 × 4	4	1	1	43	20.3±0.1
erc20 × 5	7.6±1.3	1.8±0.4	2: **3**	121±3.5: **3**	69.2±4.1: **3**
erc721 × 3	18	4	4	131.7±12.7	104.4±10
escrow × 3	6	1	2	299±1.7	72.5±0.3
finalizable × 2	4	1	2	92	19.4±0.2
lottery × 1	6	1	2	840	353.7
multi-1 × 3	15±1	1	1.5±0.7: **2**	46.5±19.1: **2**	26.7±2.1: **2**
multi-2 × 3	13±1	1.3±0.6	1.7±0.6	74±46.8	44.5±35.5
ownable × 4	2	1	1	33	15.7±0.2
pausable × 3	4	1	2	54	13.6
signer-role × 2	2	0	1	70	21.3±0.1
voting × 2	6±2.8	0	2	4037.5±4635.1	324±303
whitelisted × 3	4.3±0.6	0.7±0.6	2	2114±3105.9	172.6±152.5

Figure 6.11: **Synthesis**: counting contract fields and seed features passed as input, non-atomic terms in generated simulations, and blockchain-oracle queries.

variation contract simply sets the allowance to zero without reverting the transaction. Thus after executing the following transactions:

a_1: increaseAllowance(a_2, 1)
a_1: decreaseAllowance(a_2, 2)

the function allowance(a_1, a_2) returns 1 in the first case, but 0 in the second. Note that the *decreaseAllowance* function is not present in the ERC-20 token standard but only in the OpenZeppelin implementation that we use as the canonical ERC-20 contract.

Our implementation is limited since it does not automatically generate loop and contract invariants for verifying candidate simulation relations. Generally speaking, loop invariants on otherwise-unannotated contracts are necessary for methods with loops; contract invariants can be required in cases where the unannotated-contract state invariants are not implied by the combination of canonical-contract state invariants (which are given) and candidate simulation relations (which are computed by our synthesizer). While our experiments never required loop invariants, contract invariants were required in one case, to characterize fields of the unannotated contracts which have no direct correspondence to canonical-contract fields. In particular, one of whitelisted's unannotated contracts maintains a length field equal to the number of elements in an array; the corresponding

contracts	lines of code	verified fns.	unverified	time
auction × 3	481.3±11.6	11	0	1.9±0.1
crowdsale × 3	527.7±31.6	19.7±0.6	0.3±0.6	2.1±0.2: **2**
erc165 × 4	115.3±15.9	3	0	0.8
erc20 × 5	431±124.7: **3**	11.7±1.5: **3**	0: **3**	1.2: **3**
erc721 × 3	684.7±7.1	10	0	1.3
escrow × 3	227.7±11	7	0	1.1±0.1
finalizable × 2	146.5±20.5	5	0	0.8
lottery × 1	224	-	-	-
multi-1 × 3	408±5.7: **2**	5.5±0.7: **2**	0: **2**	1.2±0.2: **2**
multi-2 × 3	582±46.9	6.7±2.1	0	1.1±0.1
ownable × 4	185.3±25.2	4.5±0.6	0	0.8
pausable × 3	206.3±3.2	5	0	0.8
signer-role × 2	159.5±2.1	6	0	0.9
voting × 2	167±9.9	5	0	0.9
whitelisted × 3	177±5.3	5.7±0.6	0.3±0.6	1±0.1: **2**

Figure 6.12: **Verification**: counting lines of Solidity code, verified functions, and unverified functions.

canonical contract has no such length field. Such relationships hold equally in all positive and negative examples since examples only include reachable contract states. In contrast, invariant-generation for individual contracts would distinguish a contract's reachable and unreachable states. We consider generating contract and loop invariants orthogonal to simulation relations, and standard techniques exist [141].

6.9.2 Example Generation Phase

For the example generation phase we count blockchain transactions executed, transaction sequences (traces), states encountered, and positive and negative examples. Our implementation usually learns simulation relations from a relatively small set of examples: 100 examples usually suffice, up to 450 in the worst cases. Another observation is that the total number of positive and negative examples is several times the number of explored states. This happens because negative examples arise not only from observationally-inequivalent states encountered among the executed transaction sequences, but also inductively from prefixes of longer negative examples – see Lemma 6.2. The 3 cases where no examples were generated correspond to genuine counterexamples to simulation.

6.9.3 Synthesis Phase

For the synthesis phase we count the fields and seed features given to the synthesizer, non-atomic terms in the generated simulation relation, and the number of queries to the blockchain oracle for evaluating new features against examples. Note that the seed features were automatically generated as explained in §6.7.2. The primary factors to overall runtime, which is roughly proportional to the number of oracle queries, are the number and sizes of generated terms.

Despite similarities between varying canonical contract refinements, a naive syntactic strategy of listing equalities, i.e., that used to generate seed features, would not suffice (cf. §6.8). In most cases, the synthesizer is forced to generate terms that are not seed features while enumerating a relatively small number of candidates (column "queries").

6.9.4 Verification Phase

For the verification phase we count the lines of Solidity code, verified functions, and unverified functions. While verification succeeds in most cases, current limitations in solc-verify yield a few failures. The first two cases were caused by skipping and reporting parsing errors for functions which have Solidity features that are not supported by the tool. The last case requires establishing a contract invariant (see §6.9.1), yet we do not currently apply invariant-generation to individual contracts. Note that for the auction contract, the function which allows previous highest bidders to reclaim their bids invokes the Solidity `send` function to transfer ether. Thus, to prove that this function preserves the candidate simulation relation, we apply the technique described in Remark 6.2 where we use shadow variables to record the status of the invocations of `send`.

6.10 Related Work

Analysis of Smart Contracts. A number of systems have been proposed for detecting vulnerabilities in smart contracts. These systems are based on static analysis, e.g. [94, 106, 159, 164], symbolic execution engines, e.g. [100, 109, 125, 135, 160], or dynamic analysis, e.g., [96]. The systems based on static analysis are designed to expose certain coding patterns that are prone to critical bugs and cannot establish full functional correctness. In contrast, our work makes it possible to establish behavioral simulations towards verified contracts which implies full functional correctness. The systems based on symbolic execution or dynamic analysis are incomplete and can only establish correctness for *bounded* executions.

Functional Verification of Smart Contracts. Several previous works have developed methodologies for proving full functional correctness of smart contracts using theorem provers like Coq, F*, and Isabelle/HOL, e.g., [36, 49, 95, 104, 151], SMT solvers, e.g, [97, 165], or predicate abstraction [141]. These works rely on user-provided functional specifications while our work, by establishing behavioral simulations, makes it possible to verify contracts for which such specifications do not exist (as long as the simulations relate them with verified contracts).

Computing Refinement Relations Between Finite-State Systems. The complexity of computing simulation relations between *finite-state* systems has been addressed quite extensively in the literature, e.g. [64, 68, 102, 87, 88, 148]. Some of these works extend to infinite-state systems as long as they have *finite* similarity quotient which intuitively, means that they are simulated by a finite-state system. This is not the case for smart contracts which store infinite-domain inputs in their state, e.g., the auction bids of Figure 6.1.

Synthesizing "Small-Step" Simulation Relations. An established approach for proving the validity of compiler optimizations consists in synthesizing simulation relations from source to optimized programs, e.g., [42, 91, 132, 133, 134, 163]. These simulation relations concern traces of a small-step operational semantics of the two programs while our approach computes *behavioral* simulations which relate programs in terms of operation sequences, ignoring local memory and control-flow. Moreover, the simulation relations are synthesized at compile time during the construction of the optimized program. A reduction of simulation relation synthesis to solving a set of Horn clauses has been investigated in [81, 82]. This reduction has been evaluated only for validating compiler optimizations and applying it to smart contracts would require modeling Solidity semantics with Horn clauses, which is non-trivial.

Learning-Based Synthesis of Preconditions or Inductive Invariants. Learning from examples has been used to synthesize preconditions or inductive invariants that imply a *user-provided* specification, e.g, [85, 86, 139, 149, 155]. Our work addresses the verification problem when such specifications are lacking. The learning procedures defined in these works are however re-usable in our context. Our implementation leverages the one defined by [139].

6.11 Conclusion

Towards verifying unannotated smart contracts against precise functional specifications, we have proposed a notion of behavioral refinement, along with an automated simulation-based proof methodology. As noted in §6.5-6.6, our method is complete modulo three (unavoidable) sources of incompleteness: deductive verification, simulation for proving trace refinement, and learning from a bounded set of examples.

For verifying candidate simulation relations, our current implementation assumes manually-provided loop invariants, and, in some cases (see §6.9.1), contract invariants. This manual effort could likely be automated for many contracts of interest using standard invariant-generation techniques, e.g., [141]. Regardless, we consider the cost of any such manual effort to be offset by a significant benefit: the inheritance of arbitrary specifications established by the corresponding canonical contract(s). This includes hyperproperties like noninterference [92, 95, 158], because we use simulation relations instead of arbitrary trace refinement relations. The incompleteness due to simulation relations is thus also counterbalanced by the preservation of a larger class of specifications.

Chapter 7

Conclusion

In this chapter, we give a summary of the dissertation and describe possible extensions and future directions.

7.1 Conclusion

In this dissertation, we proposed algorithmic techniques for improving the reliability of distributed software. In particular, we proposed automated techniques that aid in checking the correctness of software programs that run on top of distributed systems. We addressed the problem of checking whether an application program running on top of a database is robust against the weakening of the database's consistency guarantee. We also addressed the problem of verifying that a smart contract running on top of a blockchain satisfies its functional properties. We advanced the state-of-the-art in several directions: (1) we developed reductions for the robustness problem in the context of several common consistency models to the well studied reachability problem; (2) we gave the first results on the decidability and complexity of verifying robustness in the context of transactional databases; (3) we introduced a new pragmatic technique for proving programs robustness that exploits Lipton's reduction theory; (4) we proposed a new behavioral simulation-based technique to verify unannotated smart contracts; and (5) we developed a new technique for the synthesis of behavioral simulation relations using counterexample driven synthesis. In the rest of this section, we summarize the primary technical contributions presented in this dissertation:

- We addressed the problems of checking robustness of application programs against substituting `SER` with `CC` or `SI`, `PC` with `CC`, and `SI` with `PC`. We considered three distinct semantics of causal consistency that were studied in the literature of distributed databases. We showed that the behaviors of a program over these three semantics coincide when the program does not contain write-write data races. In our reduction of robustness against substituting `SER` with `CC` or `SI` to reachability problem, we developed new characterizations of a class of traces that violate robustness called minimal violation. We also showed that by splitting a transaction to a read-only transaction followed by a write-only transaction, the robustness against substituting `PC` with `CC` can be reduced to robustness against substituting `SER` with `CC`. We developed a new instrumentation procedure that allows to reduce robustness against substituting `SI` with `PC` to reachability under serializability. Furthermore, we developed a pragmatic technique for proving robustness using the notion of movers. Finally, we applied the techniques to a set of applications collected from open source GitHub projects and the literature.

- We applied behavioral simulation between smart contract to allow the verification of unannotated smart contracts using annotated canonical smart contracts as specifications. We proposed a new technique based on counterexample guided synthesis to find behavioral simulation relations. In particular, our technique is based on a learning algorithm that discovers simulation relation from a set of examples and a verification algorithm based on relational verification that checks whether a discovered formula is a simulation relation by checking whether it is an invariant of the product contract. We developed a tool implementation of the techniques. We collected a benchmark of smart contracts obtained from open source GitHub projects and Etherscan, which we used to evaluate the implementation.

7.2 Future Work

In this section we discuss possible directions for future research that can build on the contributions of this dissertation:

- An interesting direction for future work is looking at the robustness problem in the context of *hybrid* consistency models where some of the transactions in the program can be declared serializable. These models include synchronization primitives similar to lock acquire/release which allow to enforce a serialization order between some transactions. Such mechanisms can be used as a “repair” mechanism in order to make programs robust. We believe that our approach can be extended to these models.

- Another interesting direction for future work is to consider robustness of application programs that use operations offered by abstract data types (ADTs), e.g., counters, lists, and sets, to access data stored on databases instead of the standard read/write operations. Conflict-free replicated data types (CRDTs) [153] are recently introduced ADTs that can be implemented by distributed systems while achieving availability, convergence, and partition tolerance. CRDTs ensure efficient resolution of the effects of concurrent updates to replicated data. Thus, an interesting question is to look at the robustness of programs that use CRDTs against substituting each linearizable CRDT with the corresponding weakly consistent CRDT implementation [154].

- For application programs that are not robust against the weakening of consistency relative to serializability, it is still possible that these programs do satisfy their invariants/specifications. However, we are not aware of generic verification tools that can facilitate the verification of programs that run on top of databases that implement different variations of weak consistencies. Thus, an interesting direction for future work is to develop new proof tools that facilitate the verification of these programs.

- The current implementation for synthesizing simulation relations can be improved from multiple perspectives: (1) automate the identification of canonical smart contracts against which to consider refinements, e.g., using machine-learning classifiers; (2) relax compatibility requirements on function signatures between smart contracts, e.g., to allow simulation among contracts that have similar functions with varying parameter types; (3) eliminate the need to provide example-generation parameters when synthesizing a simulation relation, e.g., using verifier counterexamples to drive example generation; (4) eliminate the need for manually-provided contract and loop invariants using standard invariant-generation techniques, e.g., [141]; and (5) use synchronized-loop product together with the synchronized function product for verifying simulation relation.

- Popular blockchains such as Ethereum terminate a smart contract execution if the amount of the computational resources, called gas, it has consumed exceeds a certain limit fixed by the author of the transaction. Consequently, smart contracts with inefficient code are gas-inefficient and error-prone. An interesting future direction is to develop a technique that given a smart contract, it synthesizes a gas-efficient contract that is a refinement of the original contract.

Bibliography

[1] Bitcoin, https://bitcoin.org

[2] Ethereum, https://ethereum.org

[3] Twitter, https://github.com/edmundophie/cassandra-twitter

[4] Blockchain is empowering the future of insurance. (2016), https://techcrunch.com/2016/10/29/blockchain-is-empowering-the-future-of-insurance/

[5] Northern trust uses blockchain for private equity record- keeping. (2017), http://www.reuters.com/article/nthern-trust-ibm-blockchain-idUSL1N1G61TX

[6] 0xcert (2019), https://github.com/0xcert

[7] Awesome cryptokitties (2019), https://github.com/cryptocopycats

[8] Bitnation (2019), https://github.com/Bit-Nation

[9] Carrefour says blockchain tracking boosting sales of some products. (2019), https://www.reuters.com/article/us-carrefour-blockchain-idUSKCN1T42A5

[10] Etherscan (2019), https://etherscan.io, retrieved November 19th, 2019

[11] Ganache (2019), https://www.trufflesuite.com/docs/ganache/overview

[12] Moloch ventures (2019), https://github.com/MolochVentures

[13] Openzeppelin (2019), https://github.com/OpenZeppelin

[14] Paxos standard erc20 stablecoin pax (2019), https://github.com/paxosglobal/pax-contracts/blob/3d50aa32c4691c46d2bf3f8150fff270849e8dbe/contracts/PAXImplementation.sol, retrieved November 19th, 2019

[15] Sirin-labs (2019), https://github.com/sirin-labs

[16] Smart contracts benchmark (2019), https://github.com/beillahi/smart-contract-simulation-data

[17] Solidity (2019), https://solidity.readthedocs.io/en/v0.4.24/solidity-by-example.html#blind-auction, retrieved November 19th, 2019

[18] State of the dapps (2019), https://www.stateofthedapps.com

[19] web3.js - ethereum javascript api (2019), https://web3js.readthedocs.io/en/v1.2.4/

[20] Erc-20 token standard (2020), https://eips.ethereum.org/EIPS/eip-20

[21] Solidity, the contract-oriented programming language (2020), https://solidity.readthedocs.io/en/v0.5.0/

[22] Why defi utopia would be finance without financiers: quicktake. (2020), https://www.bloomberg.com/news/articles/2020-08-26/why-defi-utopia-would-be-finance-without-financiers-quicktake

[23] Abdulla, P.A., Arora, J., Atig, M.F., Krishna, S.N.: Verification of programs under the release-acquire semantics. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 1117–1132. ACM (2019), https://doi.org/10.1145/3314221.3314649

[24] Abdulla, P.A., Atig, M.F., Bouajjani, A., Ngo, T.P.: Context-bounded analysis for POWER. In: Legay, A., Margaria, T. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10206, pp. 56–74 (2017), https://doi.org/10.1007/978-3-662-54580-5_4

[25] Adya, A.: Weak consistency: A generalized theory and optimistic implementations for distributed transactions. Ph.D. thesis (1999)

[26] Ahamad, M., Neiger, G., Burns, J.E., Kohli, P., Hutto, P.W.: Causal memory: Definitions, implementation, and programming. Distributed Comput. **9**(1), 37–49 (1995), `https://doi.org/10.1007/BF01784241`

[27] Alglave, J., Cousot, P.: Ogre and pythia: an invariance proof method for weak consistency models. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 3–18. ACM (2017), `http://dl.acm.org/citation.cfm?id=3009883`

[28] Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. In: Sharygina, N., Veith, H. (eds.) Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8044, pp. 141–157. Springer (2013), `https://doi.org/10.1007/978-3-642-39799-8_9`

[29] Alglave, J., Maranget, L.: Stability in weak memory models. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 50–66. Springer (2011), `https://doi.org/10.1007/978-3-642-22110-1_6`

[30] Alglave, J., Maranget, L., Tautschnig, M.: Herding cats: Modelling, simulation, testing, and data mining for weak memory. ACM Trans. Program. Lang. Syst. **36**(2), 7:1–7:74 (2014), `https://doi.org/10.1145/2627752`

[31] Almeida, S., Leitão, J., Rodrigues, L.E.T.: Chainreaction: a causal+ consistent datastore based on chain replication. In: Hanzálek, Z., Härtig, H., Castro, M., Kaashoek, M.F. (eds.) Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013. pp. 85–98. ACM (2013), `https://doi.org/10.1145/2465351.2465361`

[32] Alomari, M., Cahill, M.J., Fekete, A.D., Röhm, U.: The cost of serializability on platforms that use snapshot isolation. In: Alonso, G., Blakeley, J.A., Chen, A.L.P. (eds.) Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, Mexico. pp. 576–585. IEEE Computer Society (2008), `https://doi.org/10.1109/ICDE.2008.4497466`

[33] Alomari, M., Fekete, A.D., Röhm, U.: A robust technique to ensure serializable executions with snapshot isolation DBMS. In: Ioannidis, Y.E., Lee, D.L., Ng, R.T. (eds.) Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China. pp. 341–352. IEEE Computer Society (2009), `https://doi.org/10.1109/ICDE.2009.22`

[34] Alturki, M.A., Chen, J., Luchangco, V., Moore, B.M., Palmskog, K., Peña, L., Rosu, G.: Towards a verified model of the algorand consensus protocol in coq. In: Sekerinski, E., Moreira, N., Oliveira, J.N., Ratiu, D., Guidotti, R., Farrell, M., Luckcuck, M., Marmsoler, D., Campos, J., Astarte, T., Gonnord, L., Cerone, A., Couto, L., Dongol, B., Kutrib, M., Monteiro, P., Delmas, D. (eds.) Formal Methods. FM 2019 International Workshops - Porto, Portugal, October 7-11, 2019, Revised Selected Papers, Part I. Lecture Notes in Computer Science, vol. 12232, pp. 362–367. Springer (2019), `https://doi.org/10.1007/978-3-030-54994-7_27`

[35] Alur, R., McMillan, K.L., Peled, D.A.: Model-checking of correctness conditions for concurrent objects. Inf. Comput. **160**(1-2), 167–188 (2000), `https://doi.org/10.1006/inco.1999.2847`

[36] Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying ethereum smart contract bytecode in isabelle/hol. In: Andronick, J., Felty, A.P. (eds.) Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018. pp. 66–77. ACM (2018), `https://doi.org/10.1145/3167084`

[37] Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: On the verification problem for weak memory models. In: Hermenegildo, M.V., Palsberg, J. (eds.) Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010. pp. 7–18. ACM (2010), `https://doi.org/10.1145/1706299.1706303`

[38] Atig, M.F., Bouajjani, A., Burckhardt, S., Musuvathi, M.: What's decidable about weak memory models? In: Seidl, H. (ed.) Programming Languages and Systems - 21st European Symposium on Programming, ESOP 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. Lecture Notes in Computer Science, vol. 7211, pp. 26–46. Springer (2012), `https://doi.org/10.1007/978-3-642-28869-2_2`

[39] Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in TSO analysis. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 99–115. Springer (2011), `https://doi.org/10.1007/978-3-642-22110-1_9`

[40] Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts (sok). In: Maffei, M., Ryan, M. (eds.) Principles of Security and Trust - 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10204, pp. 164–186. Springer (2017), `https://doi.org/10.1007/978-3-662-54455-6_8`

[41] Barnett, M., Chang, B.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures. Lecture Notes in Computer Science, vol. 4111, pp. 364–387. Springer (2005), `https://doi.org/10.1007/11804192_17`

[42] Barrett, C.W., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.D.: TVOC: A translation validator for optimizing compilers. In: Etessami, K., Rajamani, S.K. (eds.) Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3576, pp. 291–295. Springer (2005), `https://doi.org/10.1007/11513988_29`

[43] Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M.J., Schulte, W. (eds.) FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6664, pp. 200–214. Springer (2011), `https://doi.org/10.1007/978-3-642-21437-0_17`

[44] Beillahi, S.M., Bouajjani, A., Enea, C.: Checking robustness against snapshot isolation. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference,

CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11562, pp. 286–304. Springer (2019), https://doi.org/10.1007/978-3-030-25543-5_17

[45] Beillahi, S.M., Bouajjani, A., Enea, C.: Robustness against transactional causal consistency. In: Fokkink, W.J., van Glabbeek, R. (eds.) 30th International Conference on Concurrency Theory, CONCUR 2019, August 27-30, 2019, Amsterdam, the Netherlands. LIPIcs, vol. 140, pp. 30:1–30:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2019), https://doi.org/10.4230/LIPIcs.CONCUR.2019.30

[46] Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil, E.J., O'Neil, P.E.: A critique of ANSI SQL isolation levels. In: Carey, M.J., Schneider, D.A. (eds.) Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995. pp. 1–10. ACM Press (1995), https://doi.org/10.1145/223784.223785

[47] Bernardi, G., Gotsman, A.: Robustness against consistency models with atomic visibility. In: Desharnais, J., Jagadeesan, R. (eds.) 27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada. LIPIcs, vol. 59, pp. 7:1–7:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016), https://doi.org/10.4230/LIPIcs.CONCUR.2016.7

[48] Betarte, G., Cristiá, M., Luna, C.D., Silveira, A., Zanarini, D.: Towards a formally verified implementation of the mimblewimble cryptocurrency protocol. In: Zhou, J., Conti, M., Ahmed, C.M., Au, M.H., Batina, L., Li, Z., Lin, J., Losiouk, E., Luo, B., Majumdar, S., Meng, W., Ochoa, M., Picek, S., Portokalidis, G., Wang, C., Zhang, K. (eds.) Applied Cryptography and Network Security Workshops - ACNS 2020 Satellite Workshops, AIBlock, AIHWS, AIoTS, Cloud S&P, SCI, SecMT, and SiMLA, Rome, Italy, October 19-22, 2020, Proceedings. Lecture Notes in Computer Science, vol. 12418, pp. 3–23. Springer (2020), https://doi.org/10.1007/978-3-030-61638-0_1

[49] Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Béguelin, S.Z.: Formal verification of smart contracts: Short paper. In: Murray, T.C., Stefan, D. (eds.) Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS@CCS 2016,

Vienna, Austria, October 24, 2016. pp. 91–96. ACM (2016), https://doi.org/10.1145/2993600.2993611

[50] Biswas, R., Enea, C.: On the complexity of checking transactional consistency. Proc. ACM Program. Lang. **3**(OOPSLA), 165:1–165:28 (2019), https://doi.org/10.1145/3360591

[51] Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7792, pp. 533–553. Springer (2013), https://doi.org/10.1007/978-3-642-37036-6_29

[52] Bouajjani, A., Enea, C., Guerraoui, R., Hamza, J.: On verifying causal consistency. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 626–638. ACM (2017), http://dl.acm.org/citation.cfm?id=3009888

[53] Bouajjani, A., Enea, C., Hamza, J.: Verifying eventual consistency of optimistic replication systems. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 285–296. ACM (2014), https://doi.org/10.1145/2535838.2535877

[54] Bouajjani, A., Enea, C., Mutluergil, S.O., Tasiran, S.: Reasoning about TSO programs using reduction and abstraction. In: Chockler, H., Weissenbacher, G. (eds.) Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10982, pp. 336–353. Springer (2018), https://doi.org/10.1007/978-3-319-96142-2_21

[55] Bouajjani, A., Meyer, R., Möhlmann, E.: Deciding robustness against total store ordering. In: Aceto, L., Henzinger, M., Sgall, J. (eds.) Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II. Lecture Notes in Computer Science, vol. 6756, pp. 428–440. Springer (2011), https://doi.org/10.1007/978-3-642-22012-8_34

[56] Braithwaite, S., Buchman, E., Konnov, I., Milosevic, Z., Stoilkovska, I., Widder, J., Zamfir, A.: Tendermint blockchain synchronization: Formal specification and model checking. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles - 9th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2020, Rhodes, Greece, October 20-30, 2020, Proceedings, Part I. Lecture Notes in Computer Science, vol. 12476, pp. 471–488. Springer (2020), https://doi.org/10.1007/978-3-030-61362-4_27

[57] Brutschy, L., Dimitrov, D.I., Müller, P., Vechev, M.T.: Serializability for eventual consistency: criterion, analysis, and applications. In: Castagna, G., Gordon, A.D. (eds.) Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017. pp. 458–472. ACM (2017), http://dl.acm.org/citation.cfm?id=3009895

[58] Brutschy, L., Dimitrov, D.I., Müller, P., Vechev, M.T.: Static serializability analysis for causal consistency. In: Foster, J.S., Grossman, D. (eds.) Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. pp. 90–104. ACM (2018), https://doi.org/10.1145/3192366.3192415

[59] Burckhardt, S.: Principles of eventual consistency. Found. Trends Program. Lang. **1**(1-2), 1–150 (2014), https://doi.org/10.1561/2500000011

[60] Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014. pp. 271–284. ACM (2014), https://doi.org/10.1145/2535838.2535848

[61] Burckhardt, S., Leijen, D., Protzenko, J., Fähndrich, M.: Global sequence protocol: A robust abstraction for replicated shared state. In: Boyland, J.T. (ed.) 29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic. LIPIcs, vol. 37, pp. 568–590. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015), https://doi.org/10.4230/LIPIcs.ECOOP.2015.568

[62] Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: Gupta, A., Malik, S. (eds.) Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings. Lecture Notes in Computer Science, vol. 5123, pp. 107–120. Springer (2008), https://doi.org/10.1007/978-3-540-70545-1_12

[63] Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency for relaxed memory models. In: Abdulla, P.A., Leino, K.R.M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6605, pp. 11–25. Springer (2011), https://doi.org/10.1007/978-3-642-19835-9_3

[64] Bustan, D., Grumberg, O.: Simulation-based minimazation. ACM Trans. Comput. Log. **4**(2), 181–206 (2003), https://doi.org/10.1145/635499.635502

[65] Cahill, M.J., Röhm, U., Fekete, A.D.: Serializable isolation for snapshot databases. ACM Trans. Database Syst. **34**(4), 20:1–20:42 (2009), https://doi.org/10.1145/1620585.1620587

[66] Calin, G., Derevenetc, E., Majumdar, R., Meyer, R.: A theory of partitioned global address spaces. In: Seth, A., Vishnoi, N.K. (eds.) IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2013, December 12-14, 2013, Guwahati, India. LIPIcs, vol. 24, pp. 127–139. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2013), https://doi.org/10.4230/LIPIcs.FSTTCS.2013.127

[67] Cassandra-lock: https://github.com/dekses/cassandra-lock

[68] Cécé, G.: Foundation for a series of efficient simulation algorithms. In: 32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017. pp. 1–12. IEEE Computer Society (2017), https://doi.org/10.1109/LICS.2017.8005069

[69] Cerone, A., Bernardi, G., Gotsman, A.: A framework for transactional consistency models with atomic visibility. In: Aceto, L., de Frutos-Escrig, D. (eds.) 26th International Conference on Concurrency Theory, CONCUR 2015, Madrid, Spain, September 1.4, 2015. LIPIcs, vol. 42,

pp. 58–71. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2015), https://doi.org/10.4230/LIPIcs.CONCUR.2015.58

[70] Cerone, A., Gotsman, A.: Analysing snapshot isolation. J. ACM **65**(2), 11:1–11:41 (2018), https://doi.org/10.1145/3152396

[71] Cerone, A., Gotsman, A., Yang, H.: Algebraic laws for weak consistency. In: Meyer, R., Nestmann, U. (eds.) 28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany. LIPIcs, vol. 85, pp. 26:1–26:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017), https://doi.org/10.4230/LIPIcs.CONCUR.2017.26

[72] Dan, A.M., Meshman, Y., Vechev, M.T., Yahav, E.: Effective abstractions for verification under relaxed memory models. Comput. Lang. Syst. Struct. **47**, 62–76 (2017), https://doi.org/10.1016/j.cl.2016.02.003

[73] Derevenetc, E.: Robustness against Relaxed Memory Models. Ph.D. thesis, University of Kaiserslautern (2015), http://kluedo.ub.uni-kl.de/frontdoor/index/index/docId/4074

[74] Derevenetc, E., Meyer, R.: Robustness against power is pspace-complete. In: Esparza, J., Fraigniaud, P., Husfeldt, T., Koutsoupias, E. (eds.) Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II. Lecture Notes in Computer Science, vol. 8573, pp. 158–170. Springer (2014), https://doi.org/10.1007/978-3-662-43951-7_14

[75] Derevenetc, E., Meyer, R., Schweizer, S.: Locality and singularity for store-atomic memory models. In: Abbadi, A.E., Garbinato, B. (eds.) Networked Systems - 5th International Conference, NETYS 2017, Marrakech, Morocco, May 17-19, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10299, pp. 133–148 (2017), https://doi.org/10.1007/978-3-319-59647-1_11

[76] Difallah, D.E., Pavlo, A., Curino, C., Cudré-Mauroux, P.: Oltp-bench: An extensible testbed for benchmarking relational databases. Proc. VLDB Endow. **7**(4), 277–288 (2013), http://www.vldb.org/pvldb/vol7/p277-difallah.pdf

[77] Du, J., Elnikety, S., Roy, A., Zwaenepoel, W.: Orbe: scalable causal consistency using dependency matrices and physical clocks. In: Lohman, G.M. (ed.) ACM Symposium on Cloud Computing, SOCC '13, Santa Clara, CA, USA, October 1-3, 2013. pp. 11:1–11:14. ACM (2013), `https://doi.org/10.1145/2523616.2523628`

[78] Durieux, T., Ferreira, J.F., Abreu, R., Cruz, P.: Empirical review of automated analysis tools on 47, 587 ethereum smart contracts. In: Rothermel, G., Bae, D. (eds.) ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020. pp. 530–541. ACM (2020), `https://doi.org/10.1145/3377811.3380364`

[79] Experiments: `https://github.com/relative-robustness/artifact`

[80] Farzan, A., Vandikas, A.: Automated hypersafety verification. In: Dillig, I., Tasiran, S. (eds.) Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I. Lecture Notes in Computer Science, vol. 11561, pp. 200–218. Springer (2019), `https://doi.org/10.1007/978-3-030-25540-4_11`

[81] Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Automated discovery of simulation between programs. In: Davis, M., Fehnker, A., McIver, A., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings. Lecture Notes in Computer Science, vol. 9450, pp. 606–621. Springer (2015), `https://doi.org/10.1007/978-3-662-48899-7_42`

[82] Fedyukovich, G., Gurfinkel, A., Sharygina, N.: Property directed equivalence via abstract simulation. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9780, pp. 433–453. Springer (2016), `https://doi.org/10.1007/978-3-319-41540-6_24`

[83] Fekete, A.D., Liarokapis, D., O'Neil, E.J., O'Neil, P.E., Shasha, D.E.: Making snapshot isolation serializable. ACM Trans. Database Syst. **30**(2), 492–528 (2005), `https://doi.org/10.1145/1071610.1071615`

[84] Fischer, M.J., Lynch, N.A., Paterson, M.: Impossibility of distributed consensus with one faulty process. J. ACM **32**(2), 374–382 (1985), `https://doi.org/10.1145/3149.214121`

[85] Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: Biere, A., Bloem, R. (eds.) 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. Lecture Notes in Computer Science, vol. 8559, pp. 69–87. Springer (2014), `https://doi.org/10.1007/978-3-319-08867-9_5`

[86] Garg, P., Neider, D., Madhusudan, P., Roth, D.: Learning invariants using decision trees and implication counterexamples. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 499–512. ACM (2016), `https://doi.org/10.1145/2837614.2837664`

[87] Gentilini, R., Piazza, C., Policriti, A.: From bisimulation to simulation: Coarsest partition problems. J. Autom. Reason. **31**(1), 73–103 (2003), `https://doi.org/10.1023/A:1027328830731`

[88] Gentilini, R., Piazza, C., Policriti, A.: Rank and simulation: the well-founded case. J. Log. Comput. **25**(6), 1331–1349 (2015), `https://doi.org/10.1093/logcom/ext066`

[89] Gibbons, P.B., Korach, E.: Testing shared memories. SIAM J. Comput. **26**(4), 1208–1244 (1997), `https://doi.org/10.1137/S0097539794279614`

[90] Gilbert, S., Lynch, N.A.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News **33**(2), 51–59 (2002), `https://doi.org/10.1145/564585.564601`

[91] Gjomemo, R., Namjoshi, K.S., Phung, P.H., Venkatakrishnan, V.N., Zuck, L.D.: From verification to optimizations. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings. Lecture Notes in Computer Science, vol. 8931, pp. 300–317. Springer (2015), `https://doi.org/10.1007/978-3-662-46081-8_17`

[92] Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982. pp. 11–20. IEEE Computer Society (1982), `https://doi.org/10.1109/SP.1982.10014`

[93] Gotsman, A., Yang, H., Ferreira, C., Najafzadeh, M., Shapiro, M.: 'cause i'm strong enough: reasoning about consistency choices in distributed systems. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 371–384. ACM (2016), https://doi.org/10.1145/2837614.2837625

[94] Grech, N., Kong, M., Jurisevic, A., Brent, L., Scholz, B., Smaragdakis, Y.: Madmax: surviving out-of-gas conditions in ethereum smart contracts. Proc. ACM Program. Lang. **2**(OOPSLA), 116:1–116:27 (2018), https://doi.org/10.1145/3276486

[95] Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) Principles of Security and Trust - 7th International Conference, POST 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10804, pp. 243–269. Springer (2018), https://doi.org/10.1007/978-3-319-89722-6_10

[96] Grossman, S., Abraham, I., Golan-Gueta, G., Michalevsky, Y., Rinetzky, N., Sagiv, M., Zohar, Y.: Online detection of effectively callback free objects with applications to smart contracts. Proc. ACM Program. Lang. **2**(POPL), 48:1–48:28 (2018), https://doi.org/10.1145/3158136

[97] Hajdu, Á., Jovanovic, D.: solc-verify: A modular verifier for solidity smart contracts. In: Chakraborty, S., Navas, J.A. (eds.) Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected Papers. Lecture Notes in Computer Science, vol. 12031, pp. 161–179. Springer (2019), https://doi.org/10.1007/978-3-030-41600-3_11

[98] Hamza, J.: On the complexity of linearizability. In: Bouajjani, A., Fauconnier, H. (eds.) Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers. Lecture Notes in Computer Science, vol. 9466, pp. 308–321. Springer (2015), https://doi.org/10.1007/978-3-319-26850-7_21

[99] Hawblitzel, C., Petrank, E., Qadeer, S., Tasiran, S.: Automated and modular refinement reasoning for concurrent programs. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided

Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9207, pp. 449–465. Springer (2015), https://doi.org/10.1007/978-3-319-21668-3_26

[100] He, J., Balunovic, M., Ambroladze, N., Tsankov, P., Vechev, M.T.: Learning to fuzz from symbolic execution with application to smart contracts. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. pp. 531–548. ACM (2019), https://doi.org/10.1145/3319535.3363230

[101] Heilman, E., Kendler, A., Zohar, A., Goldberg, S.: Eclipse attacks on bitcoin's peer-to-peer network. In: Jung, J., Holz, T. (eds.) 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015. pp. 129–144. USENIX Association (2015), https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/heilman

[102] Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing simulations on finite and infinite graphs. In: 36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995. pp. 453–462. IEEE Computer Society (1995), https://doi.org/10.1109/SFCS.1995.492576

[103] Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. **12**(3), 463–492 (1990), https://doi.org/10.1145/78969.78972

[104] Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y.A., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M. (eds.) Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10323, pp. 520–535. Springer (2017), https://doi.org/10.1007/978-3-319-70278-0_33

[105] Holt, B., Bornholt, J., Zhang, I., Ports, D.R.K., Oskin, M., Ceze, L.: Disciplined inconsistency with consistency types. In: Aguilera, M.K., Cooper, B., Diao, Y. (eds.) Proceedings of the

Seventh ACM Symposium on Cloud Computing, Santa Clara, CA, USA, October 5-7, 2016. pp. 279–293. ACM (2016), https://doi.org/10.1145/2987550.2987559

[106] Kalra, S., Goel, S., Dhawan, M., Sharma, S.: ZEUS: analyzing safety of smart contracts. In: 25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018. The Internet Society (2018), http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2018/02/ndss2018_09-1_Kalra_paper.pdf

[107] Kolluri, A., Nikolic, I., Sergey, I., Hobor, A., Saxena, P.: Exploiting the laws of order in smart contracts. In: Zhang, D., Møller, A. (eds.) Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019. pp. 363–373. ACM (2019), https://doi.org/10.1145/3293882.3330560

[108] Kozen, D.: Lower bounds for natural proof systems. In: 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. pp. 254–266. IEEE Computer Society (1977), https://doi.org/10.1109/SFCS.1977.16

[109] Krupp, J., Rossow, C.: teether: Gnawing at ethereum to automatically exploit smart contracts. In: Enck, W., Felt, A.P. (eds.) 27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018. pp. 1317–1333. USENIX Association (2018), https://www.usenix.org/conference/usenixsecurity18/presentation/krupp

[110] Kurath, A.: Analyzing Serializability of Cassandra Applications. Master's thesis, ETH Zurich, Switzerland (2017)

[111] Lahav, O., Boker, U.: Decidable verification under a causally consistent shared memory. In: Donaldson, A.F., Torlak, E. (eds.) Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020. pp. 211–226. ACM (2020), https://doi.org/10.1145/3385412.3385966

[112] Lahav, O., Giannarakis, N., Vafeiadis, V.: Taming release-acquire consistency. In: Bodík, R., Majumdar, R. (eds.) Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016. pp. 649–662. ACM (2016), https://doi.org/10.1145/2837614.2837643

[113] Lahav, O., Margalit, R.: Robustness against release/acquire semantics. In: McKinley, K.S., Fisher, K. (eds.) Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019. pp. 126–141. ACM (2019), `https://doi.org/10.1145/3314221.3314604`

[114] Lahav, O., Vafeiadis, V.: Owicki-gries reasoning for weak memory models. In: Halldórsson, M.M., Iwama, K., Kobayashi, N., Speckmann, B. (eds.) Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II. Lecture Notes in Computer Science, vol. 9135, pp. 311–323. Springer (2015), `https://doi.org/10.1007/978-3-662-47666-6_25`

[115] Lahav, O., Vafeiadis, V.: Explaining relaxed memory models with program transformations. In: Fitzgerald, J.S., Heitmeyer, C.L., Gnesi, S., Philippou, A. (eds.) FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9995, pp. 479–495 (2016), `https://doi.org/10.1007/978-3-319-48989-6_29`

[116] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978), `https://doi.org/10.1145/359545.359563`

[117] Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers **28**(9), 690–691 (1979), `https://doi.org/10.1109/TC.1979.1675439`

[118] de León, H.P., Furbach, F., Heljanko, K., Meyer, R.: Portability analysis for weak memory models. PORTHOS: one tool for all models. In: Ranzato, F. (ed.) Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10422, pp. 299–320. Springer (2017), `https://doi.org/10.1007/978-3-319-66706-5_15`

[119] Li, E., Serbanuta, T., Diaconescu, D., Zamfir, V., Rosu, G.: Formalizing correct-by-construction casper in coq. In: IEEE International Conference on Blockchain and Cryptocurrency, ICBC 2020, Toronto, ON, Canada, May 2-6, 2020. pp. 1–3. IEEE (2020), `https://doi.org/10.1109/ICBC48266.2020.9169468`

[120] Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM **18**(12), 717–721 (1975), `https://doi.org/10.1145/361227.361234`

[121] Lipton, R.J., Sandberg, J.S.: PRAM: A scalable shared memory. Tech. Rep. TR-180-88, Princeton University, Department of Computer Science (August 1988)

[122] Liskov, B., Wing, J.M.: A behavioral notion of subtyping. ACM Trans. Program. Lang. Syst. **16**(6), 1811–1841 (1994), `https://doi.org/10.1145/197320.197383`

[123] Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In: Wobber, T., Druschel, P. (eds.) Proceedings of the 23rd ACM Symposium on Operating Systems Principles 2011, SOSP 2011, Cascais, Portugal, October 23-26, 2011. pp. 401–416. ACM (2011), `https://doi.org/10.1145/2043556.2043593`

[124] Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G.: Stronger semantics for low-latency geo-replicated storage. In: Feamster, N., Mogul, J.C. (eds.) Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013. pp. 313–328. USENIX Association (2013), `https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd`

[125] Luu, L., Chu, D., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Weippl, E.R., Katzenbeisser, S., Kruegel, C., Myers, A.C., Halevi, S. (eds.) Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016. pp. 254–269. ACM (2016), `https://doi.org/10.1145/2976749.2978309`

[126] Lynch, N.A., Vaandrager, F.W.: Forward and backward simulations: I. untimed systems. Inf. Comput. **121**(2), 214–233 (1995), `https://doi.org/10.1006/inco.1995.1134`

[127] Milner, R.: Communication and concurrency. PHI Series in computer science, Prentice Hall (1989)

[128] Mitchell, T.M.: Machine learning, International Edition. McGraw-Hill Series in Computer Science, McGraw-Hill (1997), `https://www.worldcat.org/oclc/61321007`

[129] Nagar, K., Jagannathan, S.: Automated detection of serializability violations under weak consistency. In: Schewe, S., Zhang, L. (eds.) 29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China. LIPIcs, vol. 118, pp. 41:1–41:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2018), `https://doi.org/10.4230/LIPIcs.CONCUR.2018.41`

[130] Najafzadeh, M., Gotsman, A., Yang, H., Ferreira, C., Shapiro, M.: The CISE tool: proving weakly-consistent applications correct. In: Alvaro, P., Bessani, A. (eds.) Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data, PaPoC@EuroSys 2016, London, United Kingdom, April 18, 2016. pp. 2:1–2:3. ACM (2016), `https://doi.org/10.1145/2911151.2911160`

[131] Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system. Tech. rep. (2008)

[132] Namjoshi, K.S., Tagliabue, G., Zuck, L.D.: A witnessing compiler: A proof of concept. In: Legay, A., Bensalem, S. (eds.) Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8174, pp. 340–345. Springer (2013), `https://doi.org/10.1007/978-3-642-40787-1_22`

[133] Namjoshi, K.S., Zuck, L.D.: Witnessing program transformations. In: Logozzo, F., Fähndrich, M. (eds.) Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings. Lecture Notes in Computer Science, vol. 7935, pp. 304–323. Springer (2013), `https://doi.org/10.1007/978-3-642-38856-9_17`

[134] Necula, G.C.: Translation validation for an optimizing compiler. In: Lam, M.S. (ed.) Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Vancouver, Britith Columbia, Canada, June 18-21, 2000. pp. 83–94. ACM (2000), `https://doi.org/10.1145/349299.349314`

[135] Nikolic, I., Kolluri, A., Sergey, I., Saxena, P., Hobor, A.: Finding the greedy, prodigal, and suicidal contracts at scale. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. pp. 653–663. ACM (2018), `https://doi.org/10.1145/3274694.3274743`

[136] Owens, S.: Reasoning about the implementation of concurrency abstractions on x86-tso. In: D'Hondt, T. (ed.) ECOOP 2010 - Object-Oriented Programming, 24th European Confer-

ence, Maribor, Slovenia, June 21-25, 2010. Proceedings. Lecture Notes in Computer Science, vol. 6183, pp. 478–503. Springer (2010), `https://doi.org/10.1007/978-3-642-14107-2_23`

[137] Ozkan, B.K., Majumdar, R., Niksic, F., Befrouei, M.T., Weissenbacher, G.: Randomized testing of distributed systems with probabilistic guarantees. Proc. ACM Program. Lang. **2**(OOPSLA), 160:1–160:28 (2018), `https://doi.org/10.1145/3276530`

[138] Ozkan, B.K., Majumdar, R., Oraee, S.: Trace aware random testing for distributed systems. Proc. ACM Program. Lang. **3**(OOPSLA), 180:1–180:29 (2019), `https://doi.org/10.1145/3360606`

[139] Padhi, S., Sharma, R., Millstein, T.D.: Data-driven precondition inference with learned features. In: Krintz, C., Berger, E. (eds.) Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 42–56. ACM (2016), `https://doi.org/10.1145/2908080.2908099`

[140] Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979), `https://doi.org/10.1145/322154.322158`

[141] Permenev, A., Dimitrov, D., Tsankov, P., Drachsler-Cohen, D., Vechev, M.T.: Verx: Safety verification of smart contracts. In: 2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020. pp. 1661–1677. IEEE (2020), `https://doi.org/10.1109/SP40000.2020.00024`

[142] Perrin, M., Mostéfaoui, A., Jard, C.: Causal consistency: beyond memory. In: Asenjo, R., Harris, T. (eds.) Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2016, Barcelona, Spain, March 12-16, 2016. pp. 26:1–26:12. ACM (2016), `https://doi.org/10.1145/2851141.2851170`

[143] Pîrlea, G., Sergey, I.: Mechanising blockchain consensus. In: Andronick, J., Felty, A.P. (eds.) Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018. pp. 78–90. ACM (2018), `https://doi.org/10.1145/3167086`

[144] Preguiça, N.M., Zawirski, M., Bieniusa, A., Duarte, S., Balegas, V., Baquero, C., Shapiro, M.: Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine.

In: 33rd IEEE International Symposium on Reliable Distributed Systems Workshops, SRDS Workshops 2014, Nara, Japan, October 6-9, 2014. pp. 30–33. IEEE Computer Society (2014), `https://doi.org/10.1109/SRDSW.2014.33`

[145] Raad, A., Lahav, O., Vafeiadis, V.: On the semantics of snapshot isolation. In: Enea, C., Piskac, R. (eds.) Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11388, pp. 1–23. Springer (2019), `https://doi.org/10.1007/978-3-030-11245-5_1`

[146] Rackoff, C.: The covering and boundedness problems for vector addition systems. Theor. Comput. Sci. **6**, 223–231 (1978), `https://doi.org/10.1016/0304-3975(78)90036-1`

[147] Rahmani, K., Nagar, K., Delaware, B., Jagannathan, S.: CLOTHO: directed test generation for weakly consistent database systems. Proc. ACM Program. Lang. **3**(OOPSLA), 117:1–117:28 (2019), `https://doi.org/10.1145/3360543`

[148] Ranzato, F., Tapparo, F.: An efficient simulation algorithm based on abstract interpretation. Inf. Comput. **208**(1), 1–22 (2010), `https://doi.org/10.1016/j.ic.2009.06.002`

[149] Sankaranarayanan, S., Chaudhuri, S., Ivancic, F., Gupta, A.: Dynamic inference of likely data preconditions over predicates by tree learning. In: Ryder, B.G., Zeller, A. (eds.) Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008. pp. 295–306. ACM (2008), `https://doi.org/10.1145/1390630.1390666`

[150] Sergey, I., Hobor, A.: A concurrent perspective on smart contracts. In: Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y.A., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M. (eds.) Financial Cryptography and Data Security - FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers. Lecture Notes in Computer Science, vol. 10323, pp. 478–493. Springer (2017), `https://doi.org/10.1007/978-3-319-70278-0_30`

[151] Sergey, I., Kumar, A., Hobor, A.: Temporal properties of smart contracts. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice - 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November

5-9, 2018, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 11247, pp. 323–338. Springer (2018), https://doi.org/10.1007/978-3-030-03427-6_25

[152] Shapiro, M., Ardekani, M.S., Petri, G.: Consistency in 3d. In: Desharnais, J., Jagadeesan, R. (eds.) 27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada. LIPIcs, vol. 59, pp. 3:1–3:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016), https://doi.org/10.4230/LIPIcs.CONCUR.2016.3

[153] Shapiro, M., Preguiça, N.M., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In: Défago, X., Petit, F., Villain, V. (eds.) Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6976, pp. 386–400. Springer (2011), https://doi.org/10.1007/978-3-642-24550-3_29

[154] Shapiro, M., Preguiça, N.M., Baquero, C., Zawirski, M.: Convergent and commutative replicated data types. Bull. EATCS **104**, 67–88 (2011), http://eatcs.org/beatcs/index.php/beatcs/article/view/120

[155] Sharma, R., Nori, A.V., Aiken, A.: Interpolants as classifiers. In: Madhusudan, P., Seshia, S.A. (eds.) Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings. Lecture Notes in Computer Science, vol. 7358, pp. 71–87. Springer (2012), https://doi.org/10.1007/978-3-642-31424-7_11

[156] Shasha, D.E., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. **10**(2), 282–312 (1988), https://doi.org/10.1145/42190.42277

[157] Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Shen, J.P., Martonosi, M. (eds.) Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006. pp. 404–415. ACM (2006), https://doi.org/10.1145/1168857.1168907

[158] Steffen, S., Bichsel, B., Gersbach, M., Melchior, N., Tsankov, P., Vechev, M.T.: zkay: Specifying and enforcing data privacy in smart contracts. In: Cavallaro, L., Kinder, J., Wang, X.,

Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. pp. 1759–1776. ACM (2019), https://doi.org/10.1145/3319535.3363222

[159] Tikhomirov, S., Voskresenskaya, E., Ivanitskiy, I., Takhaviev, R., Marchenko, E., Alexandrov, Y.: Smartcheck: Static analysis of ethereum smart contracts. In: 1st IEEE/ACM International Workshop on Emerging Trends in Software Engineering for Blockchain, WETSEB@ICSE 2018, Gothenburg, Sweden, May 27 - June 3, 2018. pp. 9–16. ACM (2018), http://ieeexplore.ieee.org/document/8445052

[160] Torres, C.F., Schütte, J., State, R.: Osiris: Hunting for integer bugs in ethereum smart contracts. In: Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018. pp. 664–676. ACM (2018), https://doi.org/10.1145/3274694.3274737

[161] TPC: Tech. rep., Transaction Processing Performance Council (February 2010), http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf

[162] Trade: https://github.com/Haiyan2/Trade

[163] Tristan, J., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for LLVM. In: Hall, M.W., Padua, D.A. (eds.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011. pp. 295–305. ACM (2011), https://doi.org/10.1145/1993498.1993533

[164] Tsankov, P., Dan, A.M., Drachsler-Cohen, D., Gervais, A., Bünzli, F., Vechev, M.T.: Securify: Practical security analysis of smart contracts. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 67–82. ACM (2018), https://doi.org/10.1145/3243734.3243780

[165] Wang, Y., Lahiri, S.K., Chen, S., Pan, R., Dillig, I., Born, C., Naseer, I., Ferles, K.: Formal verification of workflow policies for smart contracts in azure blockchain. In: Chakraborty, S., Navas, J.A. (eds.) Verified Software. Theories, Tools, and Experiments - 11th International Conference, VSTTE 2019, New York City, NY, USA, July 13-14, 2019, Revised Selected

Papers. Lecture Notes in Computer Science, vol. 12031, pp. 87–106. Springer (2019), `https://doi.org/10.1007/978-3-030-41600-3_7`

[166] Wood, G.: Ethereum: a secure decentralised generalised transaction ledger. (2016)