On the Membership and Correctness Problem for State Serializability and Value Opacity

Dissertation

vorlegt von

Jürgen König, M.Sc.

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
an der
Fakultät für Elektrotechnik, Informatik und Mathematik

der

Universität Paderborn

Paderborn, Mai 2023

Thesis advisor: Professor Dr. Heike Wehrheim Jürgen König

On the Membership and Correctness Problem for State Serializability and Value Opacity

Abstract

Transactional memory is a concept of concurrency control managing the accesses of multiple threads to shared memory. Similar to databases, threads issue transactions that can read from and write to the shared memory. Each transaction is supposed to look to the outside as being executed as an atomic block. However, a transactional memory implementation may execute transactions concurrently to optimize runtime as long as to the outside the execution appears to be correct. There are multiple definitions formalizing what exactly “appears to be correct” means. These are called correctness conditions. While numerous approaches for checking if single executions or complete implementations fulfil these conditions have been proposed, the underlying theoretical problems have received less attention. These are the membership problem (checking whether a single execution is correct) and the correctness problem (checking whether all possible executions of an implementation are correct). In this thesis we give a detailed overview over the complexity results for each of these problems and present two results of our own for each of them. The correctness conditions we focussed on were strict state serializability - a version of serializability - and value opacity, which is a correctness condition similar to serializability but specifically designed for transactional memory. We show that the membership problem for value opacity is NP-complete by reduction from state serializability, a less strict version of strict state serializability. Additionally, we give assumptions under which the membership problem for value opacity is equivalent to the membership problem for a variant of opacity called conflict opacity that is easier to solve. For the correctness problem we investigate its decidability for strict state serializability and value opacity under assumptions. We show that the correctness problems for both conditions are decidable under these assumptions.

Thesis advisor: Professor Dr. Heike Wehrheim Jürgen König

On the Membership and Correctness Problem for State Serializability and Value Opacity

Zusammenfassung

Transaktionaler Speicher (TS) ist ein Konzept zur Kontrolle von Nebenläufigkeit, das die Zugriffe von mehreren Threads auf geteilten Speicher verwaltet. Ähnlich wie bei Datenbanken, können Threads Transaktionen ausführen, die vom Speicher lesen und auf ihn schreiben können. Jede Transaktion soll nach außen so aussehen, als ob sie in einem atomaren Block ausgeführt wurde. Trotzdem kann eine TS Implementation mehrere Transaktionen nebenläufig ausführen, um die Laufzeit zu verbessern, solange nach außen hin die Ausführung korrekt erscheint. Es gibt mehrere Definitionen, die formalisieren was genau "nach außen hin korrekt erscheinen" bedeutet. Diese Definitionen nennt man Korrektheitsbedingungen. Obwohl schon zahlreiche Ansätze für das Überprüfen, ob einzelne Ausführungen oder ganze Implementationen von TSs diesen Korrektheitsbedingungen genügen, vorgestellt wurden, ist den zugrundeliegenden theoretischen Problemen weniger Aufmerksamkeit zuteilgeworden. Diese sind das Membershipproblem (überprüfen, ob eine einzelne Ausführung korrekt ist) und das Correctnessproblem (überprüfen ob alle Ausführungen einer Implementation korrekt sind). In dieser Arbeit geben wir einen detaillierten Überblick über die bereits existierenden Ergebnisse zur Komplexität dieser Probleme und präsentieren zwei eigene Ergebnisse für jedes dieser Probleme. Die Korrektheitsbedingungen, auf die wir uns in dieser Dissertation konzentriert haben, sind Strict State Serializability und Value Opacity. Wir zeigen durch eine Reduktion von State Serializability, eine weniger strenge Version von Strict State Serializability, dass das Membershipproblem für Value Opacity NP-vollständig ist. Außerdem, präsentieren wir Annahmen, unter denen das Membershipproblem für Value Opacity äquivalent zu dem für Conflict Opacity, eine simplere Variante von Value Opacity, ist. In Bezug auf das Correctnessproblem untersuchen wir seine Entscheidbarkeit für Strict State Serializability und Value Opacity unter einschränkenden Annahmen. Wir zeigen, dass die Correctnessprobleme für beide Bedingungen entscheidbar sind unter diesen Annahmen.

Contents

Listing of Figures

Acknowledgments

I would like to thank my advisor Heike Wehrheim who supported me in conducting more theoretical research and whose feedback helped me a lot in presenting my thoughts in a digestible manner. I also want to thank the other members of my committee, Dr. habil. Stephan Merz, Prof. Dr. Stefan Böttcher, Prof. Dr. Christian Scheideler and Dr. Ulf-Peter Schroeder for their time and efforts reading and reviewing this thesis. Additionally, I want to thank my (former) colleagues: Marie-Christine Jakobs, Steffen Beringer, Manuel Töws, Julia Krämer, Cedric Richter, Felix Pauck, Jan Haltermann, Arnab Sharma and Elisabeth Schlatt. I enjoyed my time with them. On a more personal note, I would like to thank my friends and my parents Christa and Franz-Josef for keeping me sane and being great supporters during my time writing this thesis.

1 Introduction

Multicore processors are as of now present in almost every computer system. They allow for a high degree of concurrency in computing and enable large increases in processing speed. For a program to fully take advantage of these processors, it must execute multiple concurrent operations as else only a single core is being utilized. The most prevalent way of handling and implementing such concurrency is via the thread model. In this model several threads execute at once and have access to a shared memory. Designing such programs is harder than designing programs having only a single thread as the actions of different threads may be interleaved with each other in a non-deterministic manner. These interleavings may cause computation results that would not be possible in a traditional single core system.

To address these issues, concepts were developed to simplify implementing such programs. The most common approaches are based on locking. Locking is the concept of making incorrect interleavings impossible by blocking threads from proceeding into certain sections of code until it is safe to do so. Examples of concepts employing locking are critical sections, monitors and barriers. They guarantee that threads are blocked until certain conditions hold. For example, a thread may only enter a critical section when no other thread is in a critical section and a barrier blocks a thread from proceeding past a certain point of code until all threads have reached that barrier in their execution.

The main issue with such concepts is that the burden is on the programmer to use these concepts in a manner that allows for enough concurrency to effectively use the given computational power, but prevents incorrect interleavings from occurring. At the same time the programmer also needs to avoid the typical issues with locks, such as deadlocks, priority inversion and convoying. All of this combined is a very challenging task even for seasoned programmers. This sentiment is also echoed in an article about the concurrency revolution by Larus et al. "[...] humans are quickly overwhelmed by concurrency, and find it much more difficult to reason about concurrent than sequential code." [95]. These issues lead to a higher frequency of bugs in concurrent programs involving synchronisation with locks.

The consequences of such bugs range from slight inconveniences to major damage and human causalities. Two examples of the latter case are the Therac 25 and the northeast blackout of 2003. The Therac 25 was a machine used for radiation therapy, which was controlled by a computer. A concurrency error in the software led to it administering about 100 times of the actually needed radiation dose to a narrower area than intended, which caused the deaths of 3 people [65, 64]. In 2003, a race condition contributed to a blackout over parts of the USA and Ontario. It affected millions of people, and it was said to contribute to the death of almost 100 people [76, 77, 55]. The manager of the company responsible for the system stated "We had in excess of three million online operational hours in which nothing had ever exercised that bug. I'm not sure that more testing would have revealed it." [77].

Given the higher rate of bugs, how well hidden they can be and their possible consequences, verification of such programs is needed. Verification is very costly and is also specific to the program at hand. So in summary concurrent programs are hard to write and also hard to verify, a very inconvenient combination of problems.

We will illustrate and discuss these issues along a running example shown in Figure 1.1. The class `Account` implements an account with a balance. It has a getter (`getBal`) and a setter (`setBal`) method for its balance and a method to double it (`doubleBalance`). Each instance of `Account` starts with a balance of 10. We assume the system consists of two threads accessing one instance of `Account`. In the example executions (Figures 1.2 and 1.3) both threads call `doubleBalance` once.

```java
public class Account{
    int balance;

    public Account(){
        balance = 10;
    }

    public void setBal(int bal){
        balance = bal;
    }

    public int getBal(){
        return balance;
    }

    public void doubleBalance(){
        int x;
        x = getBal();
        x = x * 2;
        setBal(x);
    }
}
```

Figure 1.1: Account class running example

Typically, issues with concurrent programs occur whenever in an execution two concurrent threads access the same (shared) object and one of them modifies the object. Two examples of executions illustrating the problem of such conflicting accesses are shown in Figures 1.2 and 1.3. In Figure 1.2, a sequential execution is shown, meaning one method call executes after the other without any concurrency taking place. While in actuality `doubleBalance` is being called by both threads, here we only show the calls to `getBal` and `setBal` as they are the ones causing potential conflicts. As one can see, the execution of thread one sets the balance to 20 and the subsequent execution of thread two then sets it to 40. Such sequential behaviour and its respective result is what a programmer usually assumes their code to have. But as shown in Figure 1.3, this assumption may prove wrong whenever the method calls of both threads interleave. In this example, thread one first reads the balance and thread two reads it afterwards. Now both do their internal computations on the value, which results in 20 for both. Then thread one sets the balance to 20 and thread two sets it to 20 as well afterwards. Thus, the resulting balance of the account is 20. This behaviour does not match a sequential execution and is not intended by the programmer. Considering all possible interleavings during programming code of non-trivial complexity is an impossible task. So, as discussed above, the common approach is to limit concurrency in some fashion via locking. Besides the type of approach used, the implementation of locks can be characterized by its granularity, which is on a scale in between coarse and fine.

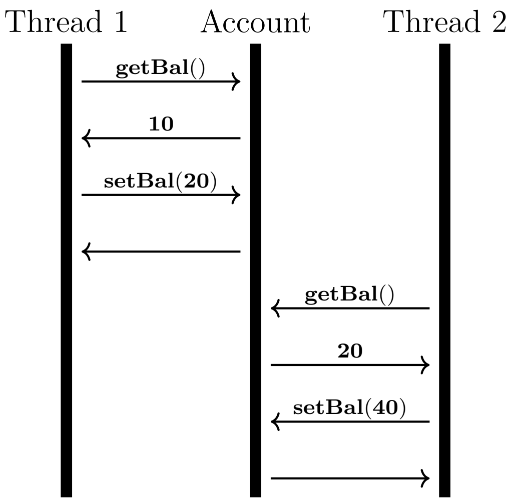


Figure 1.2: Sequential execution of `doubleBalance`

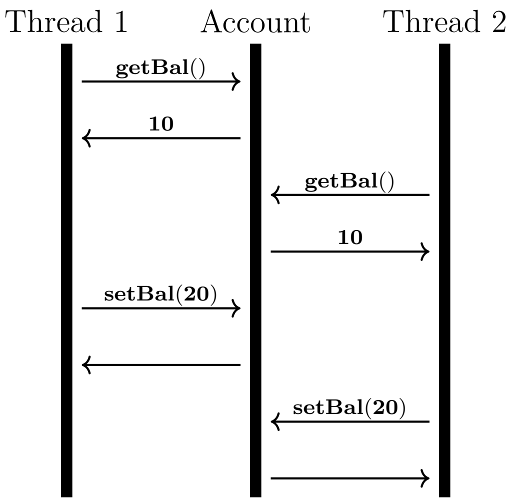


Figure 1.3: Interleaved execution of `doubleBalance`

Coarse-grained locking uses only a small number of locks, and puts them around large sections of code. The result of applying this concept to our running example is shown in Figure 1.4. Here the complete `Account` class is equipped with one static lock, so whenever any thread tries to modify one instance of `Account`, no other thread may modify any instance of `Account`. This ensures a correct execution of a program with only calls to `doubleBalance`, but is also fairly inefficient as no concurrency takes place in the `doubleBalance` method and even non-conflicting accesses to different `Account` instances cannot be concurrent. So, in summary, this style of locking reduces the overall concurrency, and thus the speed of the computations of the program. On the other hand the locking operations themselves cause only a small computational overhead, and it is easy to program with coarse-grained locking.

Fine-grained locking includes the use of potentially multiple locks, and tries to minimize the amount of sections protected by locks. It tries to put locks only around sections, which actually can cause conflicts. In Figure 1.5 one can see the application of fine-grained locking to our running example. Instead of a static lock for the whole class, the critical sections are protected by a "synchronized" statement specific to the instance of `Account`. This prevents other threads from accessing this section when calling the method for the same object. Also, the read and the write accesses are protected separately. This means during the internal computations the account can be accessed by other threads. This is advantageous whenever the internal computations take a lot of time, and thus the read would be blocked for a long time. This may cause issues when the value has changed during the computations. Thus, before actually writing to the account, a check whether the value has changed is necessary. If it has changed, the method does not write to the account and signals it has failed. Else it proceeds with writing and signals it was successful. As one can see, the complexity of programming, even in a toy example, went up significantly. Also, locking twice per method instead of once carries overhead with it. On the other hand, concurrent readers can access the account while the method is doing its internal computations.

As seen by this in example, fine-grained locking allows for potentially faster computations. On the other hand, it increases the overhead caused by locking and unlocking and the overall complexity of the program, making it more prone to errors. So overall, locks either have the problem of reduced performance or steeply increased complexity.

Also, one can see that the implementation of the locks depends on the code at hand, and would look very different given other use cases, which illustrates the verification issues mentioned above.

```
        static Lock userLock = new Lock();
        public void doubleBalance(){
            try{
                userLock.lock();
                int x;
                x = getBal();
                x = x * 2;
                setBal(x);
            }
            finally{
                userLock.unlock();
            }
        }
```

Figure 1.4: Coarse-grained locking applied to `doubleBalance`

Transactional memory *Transactional memory* (TM) is a solution to these problems. It shifts the complexity of implementing concurrency away from the programmer to a separate algorithm by giving the programmer access to a number of operations to access shared memory. This algorithm then can be verified separately as it is its own entity. It was proposed by Herlihy and Moss in 1993 as a multiprocessor architecture [50]. This approach employed novel processor instructions built into the architecture. TMs employing such an approach are also called hardware transactional memory (HTM). In 1997 Shavit and Toitou proposed a similar approach, which was software based, called software transactional memory (STM) [90]. STM implements the properties of TM using standard processor instructions. There have been several approaches (e.g. [27, 22, 47]) proposed, some of them combining HTM and STM into so-called hybrid TMs [23, 59]. In its

```
public boolean doubleBalance(){
    int x,y;
    synchronized{this}{
        x = getBal();
        y = x;
    }
    x = x * 2;
    synchronized{this}{
        if(y == getBal()){
            setBal(x);
            return true;
        }
        else return false;
    }
}
```

Figure 1.5: Fine-grained locking applied to `doubleBalance`

original form TM was completely lock-free, which completely sidestepped any of the known issues with locks (deadlocks, priority inversion and convoying). Later on, the usefulness of lock free TMs was questioned [34, 26] and approaches using locks were proposed [27, 82, 2]. An example of how a programmer could use an STM is shown in Figure 1.7. As one can see, a block of code is marked to be executed atomically as a *transaction*.

Transactions are a concept taken from databases. A transaction is a block of instructions meant to follow the ACID (atomicity, consistency, isolation, durability) principle. In short these conditions require that transactions execute completely or not all (atomicity), leave the system in a valid state (consistency), to the outside look as if executed sequentially (isolation), and if completed successfully stay so in the face of crashes (durability).

A transaction starts with a begin and ends either in a commit or abort. In between, read and write instructions take place. These instructions form a set of reads from the shared memory and tentative changes to this memory. On an abort, the tentative changes do not become visible. On the other hand, on a commit, all tentative changes are visible to other transactions. A transaction in the context of TMs may also include internal computations.

The actual details of the implementation are up to the TM. For example, a TM may not wait for the commit and already write during each write instruction (*direct update*) instead of writing at commit time (*deferred update*). Also, transactions may be concurrent to improve performance. But no matter which exact implementation strategy is chosen, it is important that a TM functions as intended, seemingly executing all transactions sequentially. This makes the verification of TMs an important topic in this research field with the main fields being testing, model checking and deductive verification. To use these methods, it is important to know what constitutes correct behaviour for a TM.

Correctness conditions Such behaviour is formalized in correctness definitions, that resemble the informal notion of correctness we discussed. Such definitions are called *correctness conditions*.

Correctness conditions are languages defined on words called *histories*, which are logs of the instruction calls (begin, write, read, commit, abort) during the execution of a TM. A short example showing two transactions can be found in Figure 1.6. It corresponds to the executions of Figures 1.2 and 1.3 with the red actions belonging to thread 1 and the green actions belonging to thread 2. The exact meaning and definitions of histories are explained later in Chapter 2.

Correctness conditions are mainly divided in two categories: *conflict-based* and *value-based*. In the first case, definitions are based upon conflicts between different actions, and do not take into account the actual values written and read. For example, a write and a read on the same variable are considered to be in conflict. In the second case, values, are taken into account, requiring that either the end state, or all transactions view values of the shared memory corresponding to a sequential execution of previous transactions.

Up until now several correctness conditions have been proposed, earlier ones such as linearizability [51], view serializability [97], state serializability, strict state serializability and conflict serializability [72] were taken from related fields. Serializability definitions usually do not consider aborted transactions, which with their original context being databases makes sense as a transaction without effect can read any inconsistent state without issue. But for actual programs, the values read by aborted transaction can matter as inconsistent values may cause errors such as infinite loops to occur. Thus, later on, correctness conditions were designed with TMs in mind, such as (value) opacity, conflict opacity and virtual world consistency [42, 46, 53]. These conditions require aborted transaction to read values that could have been caused by a sequential execution.

$$\mathbf{B}_{t_1}\mathbf{R}_{t_1}(bal., 10)\mathbf{W}_{t_1}(bal., 20)\mathbf{C}_{t_1}\mathbf{B}_{t_2}\mathbf{R}_{t_2}(bal., 20)\mathbf{W}_{t_2}(bal., 40)\mathbf{C}_{t_2}$$
$$\mathbf{B}_{t_1}\mathbf{R}_{t_1}(bal., 10)\mathbf{B}_{t_2}\mathbf{R}_{t_2}(bal., 10)\mathbf{W}_{t_1}(bal., 20)\mathbf{W}_{t_2}(bal., 40)\mathbf{C}_{t_1}\mathbf{C}_{t_2}$$

Figure 1.6: Example histories, **B** (Begin), **W** (Write), **R** (Read), **C** (Commit), bal. is short for balance.

```
public boolean doubleBalance(){
    int x;
    atomic{
        x = getBal();
        x = x * 2;
        setBal(x);
    }
}
```

Figure 1.7: Implementation of `doubleBalance` using an STM

The complexity of the verification of TM correctness conditions

A theoretical question - with practical implications - is how complex verification is for these different correctness conditions. There are two main problems that have been formalized with regard to this:

1. the *membership problem*
2. and the *correctness problem*.

The membership problem is the problem of verifying whether a single history is correct under such a correctness condition, meaning it is a member of the language defined by the condition. It is a formalization of the problem of testing single executions for their correctness.

The correctness problem or alternatively the model checking problem is the problem of verifying whether a TM is correct under a correctness condition. A TM is called correct under a correctness condition whenever all the histories it produces are correct under this condition. It formalizes the verification of the correctness claims of a TM. The correctness problem is always equally hard or harder than the membership problem for any correctness condition. This is because the correctness of a single history is equivalent to checking a TM that only outputs one history for correctness.

Besides being a result of a theoretical question, such complexity results also have practical implications. When designing verification approaches, it is very helpful to know what is achievable in the best case. If for example the correctness problem for a certain condition is undecidable, then a model checking approach would need to take this into account, for example by requiring certain properties for the TM to reduce the complexity of the problem. Also, the proofs of complexity sometimes can be inspiration for verification approaches, e.g. showing that a problem is NP-complete can involve giving an algorithm or a reduction to another problem, which can be used for practical approaches.

Existing works are concerned with the membership problem for linearizability [40], conflict serializability and serializability [73] and the correctness problem for conflict serializability [3, 37], sequential consistency and linearizability [3] and conflict opacity [42]. Specifically for the membership problem for opacity and for the correctness problem of serializability and opacity, there are still gaps in the existing literature.

Contributions The main goal of this thesis is to close these gaps by giving the complexity/decidability of each of these problems. Additionally, we provide an overview of already existing results with regard to the complexity of the membership and correctness problem for different conditions and compare these results.

We provide the following contributions:

1. we prove that the membership problem for opacity is NP-complete,
2. we compare conflict opacity and value opacity and show that they coincide under three assumptions,
3. we prove that the correctness problem for strict state serializability is decidable under assumptions
4. and we prove that the correctness problem for opacity is decidable under assumptions.

The first two contributions are concerned with the membership problem, which as mentioned previously has some transference to the correctness problem. Our complexity result for the membership problem for opacity is done via a reduction from state serializability. This reduction shows that the problem for value is at most as complex as the problem for state serializability. We show how the properties of state serializability map to the properties of opacity, offering deeper insights into how these conditions compare. Additionally, the results give a baseline for the runtime of any algorithm testing single executions for opacity.

The comparison between conflict opacity and opacity shows differences in the two conditions. Conflict opacity is a stricter condition than value opacity and is thus easier to verify for membership and correctness. Our results imply that under reasonable assumptions checking for conflict opacity is sufficient to also determine value opacity.

The last two contributions are concerned with the correctness problem and both show that the problem in question is decidable. In both cases we consider the correctness problem with the TM being deterministic and having a finite amount of variables, threads and possible values. This is an assumption also made in related literature [3].

We show that the correctness problem for strict state serializability is decidable under the assumption that the given TM terminates without any pending transactions, and each transaction is transitively influencing the end state. The result is proven by giving an algorithm solving the problem. The intuition of the algorithm is to explore the state space of all possible executions, and determine whether all executions the TM terminates in are serializable. As this state space is infinite, the algorithm employs a state space reduction technique that groups similar histories into equivalence classes. We prove that this makes the state space finite, and thus our algorithm can explore it in finite time. Beside the result itself, this state space reduction technique can also be used in actual model checking approaches for TMs.

Lastly, we show that the correctness problem for value opacity is decidable for implementations where each read is justifiable by a write of a previous transaction, and it is possible to identify the writer for each read unambiguously. These assumptions require all implementations to finish each transaction in a finite amount of steps. We modify the solution for serializability to opacity by accounting for additional constraints, and the possibility of aborts in opacity, which state serializability does not have. Again, this technique could be used to design an actual model checking approach for a TM.

All results of this thesis are formally proven.

Structure This thesis is structured as follows. In Chapter 2, we present a more in-depth overview of TMs and their correctness conditions, and we introduce the formal notation and definitions of correctness conditions needed for the remainder of this thesis. The main part of this thesis is grouped into two chapters. In Chapter 3, we are concerned with the membership problem. First, we give an overview of the related work in this field, then we present our complexity results for value opacity and finally the comparison between conflict opacity and opacity under different assumptions.

In Chapter 4, we are concerned with the correctness problem. After giving an overview of the related work, we first present our results for strict state serializability and then our results for value opacity.

In Chapter 5, we end our thesis with a conclusion and an outlook into possible future work.

The appendixes contain all proofs that were left out of their respective sections for space purposes. The proofs left out of Chapter 3 can be found in Appendix A.

The proofs left out of Chapter 4 can be found in Appendix B for the correctness problem for strict state serializability and in Appendix C for the correctness problem for value opacity.

2
Basics of STMs and Correctness Conditions

In this chapter, we will first give further context and information about TMs and their correctness criteria, and then present the notation and definitions needed for this thesis. In Section 2.1, we will expand upon the introduction and give a short overview of the mechanisms and functions of TMs. In Section 2.2, we will give an overview of different correctness conditions and illustrate them with examples. The complexity of these conditions and how they compare to each other is discussed in the respective related work sections of Chapters 3 and 4. Finally, in Section 2.3, we introduce the notation and definitions used for the rest of the thesis.

2.1 Transactional Memory

As discussed in the introduction, TMs are a mechanism to aid a programmer in managing accesses to shared memory. We also discussed the general history of TMs in the introduction. As we are mostly concerned with the complexity of correctness conditions, which is mostly independent of the inner workings of TMs, we will only give a limited overview of the design of TMs. The interested reader can find further information in the following articles [69, 48]. The main three fields where TMs can be differentiated are:

1. *data versioning*,
2. *contention management*
3. and *synchronisation strategy.*

Data versioning Data versioning refers to how a TM handles writes of transactions. As the changes made by a transaction are supposed to be either visible completely or not at all, there are two strategies that are typically used [48, 41]:

- Direct (eager) update: Whenever a transaction writes to a variable, it directly updates it.
- Deferred (lazy) update: All the writes of a transaction are saved locally and written to the shared memory whenever the transaction commits.

As a transaction may abort in transactional memory, it is necessary to somehow ensure that its writes can be undone. For direct update TMs, this is typically realized with an undo log for each transaction. This log stores the old value of a variable in a memory location whenever the transaction the log belongs to writes to that variable. On an abort, the previous values are taken from the undo log and restored to their variables[48, 41]. For deferred update data versioning, typically a buffered update is used. The writes of a transaction writes are stored in this buffer during its execution. On commit, the writes are written to the memory, on abort they are simply discarded [48, 41].

In comparison, the direct update approach has the advantage that commits are significantly less complicated and thus faster [41, 14]. This is because all the values are already written to the memory and so reads always read the newest written value [82]. Aborts may become slower as old values need to be restored to memory from the undo log [41, 14].

The deferred update approach offers the advantage that synchronisation is significantly easier as all synchronisation related to acquiring the right to write can be done at one point during the commit. For locks, deadlocks can then be prevented as they can be acquired in some fixed order. Additionally, they are held for a shorter time as the locks are acquired only at the end of a transaction. Aborts can then be executed faster compared to direct update as the memory remains unchanged. Dice and Shavit argue that deferred update offers more scalability [26]. A disadvantage is that commits may take longer as values may need to be written to memory [41, 14], and that acquiring the value that has been most recently written to a variable requires more computation as it is stored in write buffers [82].

Synchronisation strategy There are two main categories with regard to the synchronisation strategies employed by transactional memories:

- *blocking*
- and *non-blocking*.

A blocking implementation may block a process/transaction from proceeding. For example, this may happen in the case of conflicts. These implementations usually use locks. A non-blocking implementation, on the other hand, never blocks a process/transaction from proceeding, which may mean that it gets aborted. Transactional memory in its original design was supposed to be non-blocking [50], although later on multiple blocking implementations were developed [27, 82, 2].

Saha et al. discuss the trade-offs concerning this choice in a paper published 2006 [82]. A blocking implementation reduces the number of aborts as threads may be blocked and continue later instead of being aborted. Blocking synchronisation also simplifies memory management and allows for optimization because it is often known beforehand if a transaction will commit. On the flipside, a blocking implementation needs to address priority inversion, deadlock, convoying and lock contention [39]. Also, for blocking synchronisation composing operations is difficult [39]. While non-blocking implementations do not suffer from these issues, Ennals argues that non-blocking synchronisation (specifically referring to obstruction-freedom and lock-freedom) hinders performance of transactional memory [34]. His findings are supported by Dice and Shavit [26].

Contention management Contention management is the process of detecting conflicts and resolving them. There are two overarching categories of TMs with regard to contention management:

- *optimistic*
- and *pessimistic.*

A pessimistic TM has synchronisation strategies in place that avoid conflicts and subsequent aborts from occurring. The synchronisation mechanisms for pessimistic TMs involve blocking code threads during execution. Pessimistic TMs have been shown to have superior performance compared to optimistic TMs in some contexts [2].
An optimistic TM lets transactions run freely, and when a conflict between two transactions is detected the TM manages the conflict. This is usually done by aborting one of them and restarting it later or by halting one of the conflicting transactions [2, 91]. To this end, such TMs need procedures to handle detecting a conflict and deciding which of the conflicting transactions may proceed. These procedures are discussed below. Pessimistic and optimistic are not distinct features, there are also TMs in which certain transactions are pessimistic (never abort) and others are optimistic [74, 6].

The detection of conflicts then can be categorized either as

- *lazy*
- or *eager.*

A lazy conflict detection detects conflicts as transactions try to commit. An eager conflict detection does so during operations that can cause conflicts. In general, lazy conflict detection offers the advantage of less false positives and the option to batch conflict checking. Eager conflict detection may reduce the time an eventually aborted transaction is running and how much of it must potentially be undone. Additionally, eager conflict detection makes halting transactions in favour of aborting them easier [41, 14]. There is a limited amount of work with regard to studying and evaluating different conflict detection strategies [93, 14, 94, 86]

Given such a conflict, its handling is determined by contention managers. The goal of a contention manager is to ensure progress, i.e. that transactions at some point commit successfully. Its goal is not to ensure correctness of the TM [45]. There is a significant amount of contention managers ranging from fairly simple algorithms to elaborate schemes [44, 43, 45, 52, 85, 39, 30, 75, 92, 67].

2.2 Correctness Conditions

The correctness goals of TMs were originally derived from the already mentioned ACID principle used in the context of databases. This principle stands for the following properties:

- *Atomicity*: The transactions are supposed to execute in one step or not at all. (Note that atomicity in TM research is often used to mean both atomicity and isolation [87].)
- *Consistency*: The transactions are supposed to preserve the correctness requirements of the database.
- *Isolation*: The internal computations of transactions are supposed to be invisible to other transactions.
- *Durability*: If transactions commit, their effects are supposed to survive system crashes.

In the context of TMs transactions are meant to make small critical sections of code, which are specifiable by the programmer, execute atomically [50]. Usually durability is not a design goal [87], although in recent time there also has been research into handling crashes [11, 79, 21]. Over time different correctness conditions evolved from the ACID principle. Here we will give an overview of relevant correctness conditions for TMs. Correctness conditions are defined by which executions they deem correct and which not. Such executions are modelled by histories. First, we will introduce how histories are visualized in this section and describe the three ways how reads-from relationships are handled in the correctness conditions. Finally, we will present the correctness conditions used for TMs, first conditions taken from other fields and then correctness conditions designed for TMs.

Histories A history consists of a number of transactions executed by a number of threads. As mentioned in the previous section, we are interested in TMs with read/write objects. In this case, the actions of a transaction typically consist of five operations:

1. begin: starts a transaction,
2. read: reads the value of a variable from memory,
3. write: (possibly tentatively) writes a value to a variable in memory,
4. abort: undoes all writes made up to this point by the transaction and stops it
5. and commit: tentative writes are written to memory if necessary and the transaction is finished.

These operations begin with an invoke and end with a response. A number of operations or parts of them may be abstracted away if they are implicitly clear or not needed for a specific correctness condition. This is often done by omitting begin, commit and abort operations, or by contracting the invoke and response of operations into a single event.

An example of our visualisation of histories is shown in Figure 2.1.

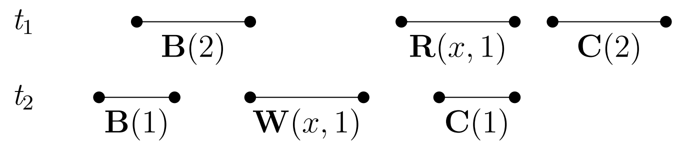


Figure 2.1: Example history visualisation

Each row represents a different thread, denoted by t with an index. Each line represents an operation where the first dot indicates the invoke event and the second dot indicates the response event. For the examples in this section, we map the following abbreviations to each operation:

1. begin of transaction $tr \rightarrow \mathbf{B}(tr)$,
2. read of value val from variable $x \rightarrow \mathbf{R}(x, val)$,
3. write of value val on variable $x \rightarrow \mathbf{W}(x, val)$,
4. commit of transaction $tr \rightarrow \mathbf{C}(tr)$
5. and abort of transaction $tr \rightarrow \mathbf{A}(tr)$.

Note that values may be left out in certain notations. Here, we assume the initial state of all variables is 0.

We will shortly introduce specific serial histories called *witnesses*, in which transactions execute after one another. Almost all correctness conditions deem a history as correct if a witness fulfilling certain requirements exists. The exact requirements for a witness are specific for each condition. But typically witnesses contain a subset of or all events of the original history. If there exists a reads-from relation in the original history, the witness must match it completely or partly, depending on the correctness condition.

Handling of reads-from relations One of the main distinguishing factors of different conditions is how they model the reads-from relation of a history. There are three main types of reads-from relations in the context of TM correctness conditions.

In the first it is assumed a read on a variable reads from the most recent write on that variable. We will call this a *most-recent reads-from relation.* For this to be an unambiguous relation for each history, we need to assume an atomic point where writes and reads take place. We assume deferred update semantics, and thus we assume each write of a transaction takes place at the response event of its commit and each read takes place at its response. When this type of reads-from relation is used, histories typically do not contain explicit values.

In the second type of reads-from relation, it is assumed that each value is only written once for each variable. Then if transaction tr_1 writes a value to a variable and transaction tr_2 reads that value on the same variable, then tr_2 reads that variable from tr_1. This is called an *unambiguous value-based reads-from relation*. One could also just assume this relation to be given explicitly as an additional input instead of being coded into a history via values.

The third way is to not have an explicit reads-from relation. Transactions write and read arbitrary values from variables without any defined relation. For a witness it is then required that the value of each read matches the value from the most recent write on that variable. For a witness this is well-defined as it is sequential and there are no concurrent transactions. This is called an *ambiguous value-based reads-from relation*.

Note that most conditions employing a reads-from relation any of the three ways to derive the reads-from relation can be used. In the case of changing the reads-from relation of a correctness condition to an ambiguous value-based reads-from relation the requirements for a witness must be altered slightly to reflect that the writer to a read can be one out of multiple transactions.

We will describe correctness conditions with the reads-from relation the original authors used. For some conditions it does not make sense to use a different way than the originally intended one, because of their unique properties.

Derived correctness conditions In this paragraph, we will discuss correctness conditions borrowed from other fields.

Sequential consistency was introduced in 1979 by Lamport [60]. It does not include the concept of transactions. A history fulfils sequential consistency whenever the behaviour of each thread matches the behaviour of a legal witness. The witness includes all events of the history. Originally, this was defined for arbitrary objects. In our case of read/write objects, this means each read reads the most recent write on its variable. We assume an ambiguous value-based reads-from relation for our examples as it is the most natural for read/write objects. Each witness of a history must be a permutation of the original history derived by repeatedly swapping adjacent events of different threads. Examples illustrating sequential consistency can be found in Figure 2.2.

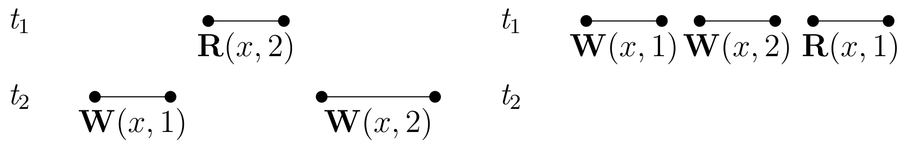


Figure 2.2: Sequential consistency examples, left correct, right incorrect

The left example is correct as both threads match the behaviour of a legal witness where thread 2 executes first and then thread 1. But if we take the prefix until the read of thread 1, this history is not sequentially consistent as there is no write of 2 on x. The right example is incorrect. There is no legal witness matching the event order of thread 1.

Linearizability was introduced by Herlihy and Wing in 1990 and is a correctness condition for concurrent data structures such as locks and arrays [51]. Linearizability does not use the notion of transactions. Intuitively, it requires that the operations of each object can be assigned a single point in time (linearization point) at which they seemingly took place. These linearization points must lie within the execution time of the operation. For each object a subsequence of the witness containing only events of that object is considered. Each of these subsequences must be legal with regard to the specification of their respective objects. Again, we assume read/write objects and an ambiguous value-based reads-from relation.

Examples illustrating linearizability can be found in Figure 2.3.

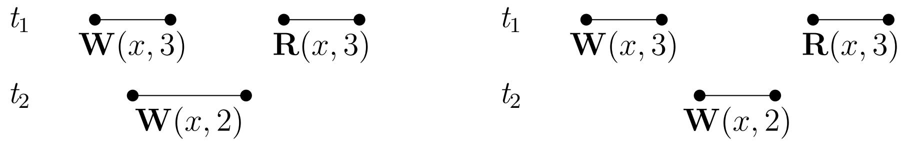


Figure 2.3: Linearizability examples, left correct, right incorrect

In the left example the linearization point of the operation of thread 2 can be

before the linearization point of the operation of thread 1, resulting in a legal witness. The right example is incorrect as the linearization point of the operation of thread 2 must be in between the linearization points of the operations of thread 1.
Serializability is a correctness condition originally meant for databases, which also has been adopted for TMs. There are several versions and notations used for serializability [73, 35, 37]. It is mostly defined for transactions containing read and write events. Versions of serializability are view serializability [97], state serializability, strict serializability [73], conflict serializability [73, 35] and causal serializability [80]. In all of these versions of serializability aborted transactions are not part of witnesses.

View serializability was published by Yannakakis in 1984. The idea is that in a concurrent history transactions should read values that they would also read in some serial history. A history is thus view serializable if there is an equivalent witness. Here equivalency means for each transaction each read reads from the same transaction in the history and the witness. In the original definition the most-recent reads-from relation is used. Additionally, the most recent writer for each variable before the end of the original history must also be the most recent writer for that variable in the witness. For other definitions it is sometimes explicitly required. Note that there are definitions of view serializability using value-based reads-from relations.

Two example histories to illustrate view serializability can be found in Figure 2.4. We omit concrete values as they are not relevant for the definition.
The top example is correct. Consider the witness with transaction order $1, 2, 3$. As in the original history, transaction 1 reads from the initial state and the most recent write on x and y at the end of the history is from transaction 3. This example also shows that view serializability is not prefix-closed. Consider the prefix containing transactions 1 and 2, both possible witnesses do not fulfil the requirements. The witness with transaction order $1, 2$ has different last writers than the original history, and in the one with transaction order $2, 1$ transaction 1 reads x and y from transaction 2. The bottom example is not correct as no matter how transactions 2 and 3 are ordered any witness has either transaction

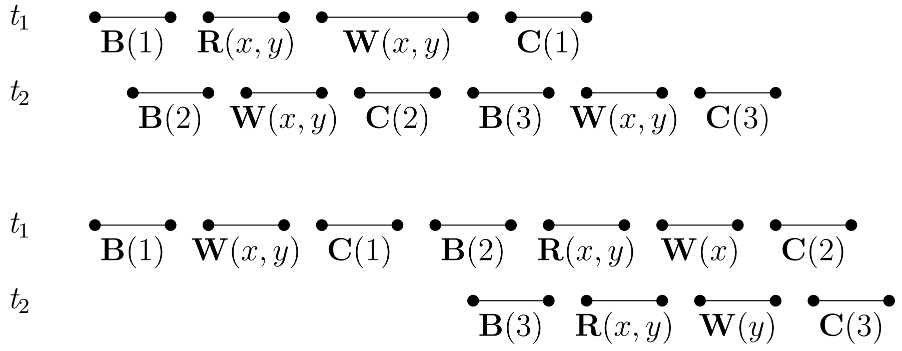

t_1
$\mathbf{B}(1)$ $\mathbf{R}(x,y)$ $\mathbf{W}(x,y)$ $\mathbf{C}(1)$
t_2
$\mathbf{B}(2)$ $\mathbf{W}(x,y)$ $\mathbf{C}(2)$ $\mathbf{B}(3)$ $\mathbf{W}(x,y)$ $\mathbf{C}(3)$
t_1
$\mathbf{B}(1)$ $\mathbf{W}(x,y)$ $\mathbf{C}(1)$ $\mathbf{B}(2)$ $\mathbf{R}(x,y)$ $\mathbf{W}(x)$ $\mathbf{C}(2)$
t_2
$\mathbf{B}(3)$ $\mathbf{R}(x,y)$ $\mathbf{W}(y)$ $\mathbf{C}(3)$

Figure 2.4: View serializability examples, top correct, bottom incorrect

2 reading y from transaction 3 or transaction 3 reading x from transaction 2. In both of these cases this differs from the original history where each read is from transaction 1.

State serializability was introduced by Papadimtriou in 1979. A history intuitively is state serializable whenever there exists a witness with an identical end state. As with the view serializability definition, it uses the most-recent reads-from relation. For the end state to be identical in a witness, the reads of any transaction influencing the end state must read from the same transaction as in the original history. Influencing the end state means that either a transaction is the most recent writer on a variable before the end of the history/witness, or it gets read by a transaction that influences the end state. Any transaction that does not influence the end state is called dead. All of its read are not considered for determining state serializability.

Two examples can be found in Figure 2.5.
Note that the original definition of histories made by Papadimitriou contracted certain history events together. We will discuss this in further detail in the notation section. However, we will use our general history format here, but we omit values. The top history is correct as neither transaction 1 nor transaction 2 influence the end state. This is because they are overwritten by transaction 3. So while in any witness either transaction 1 or transaction 2 reads x from a different writer compared to the original history, it does not matter for correctness. So, for example, the witness with transaction order $1, 2, 3$ is a serial history with an

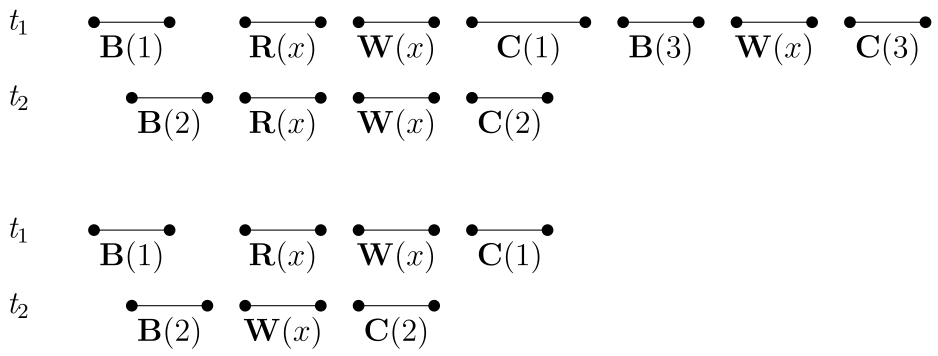


Figure 2.5: State serializability examples, top correct, bottom incorrect

identical end state to the original history. The prefix of this history containing only transactions 1 and 2 is not state serializable as both possible witnesses are not state serializable. In the witness with transaction order $1, 2$, transaction 2 reads x from transaction 1 instead of the initial state, and for the witness with transaction order $2, 1$ it is the other way around. The bottom history is incorrect. Consider both possible witnesses, which are transaction 1 and then transaction 2 or the other way around. In the first case, transaction 2 is read by the end state which it is not in the original history. In the second case, transaction 1 is the last writer before the end state on x, which is correct, but it reads x from transaction 2 which it does not in the original history.

Conflict serializability is a variation of state serializability which is more efficiently recognizable. It was also introduced by Papadimitriou in 1979 [73]. It does not explicitly use the concept of a reads-from relation. Values are not taken into account for this type of serializability. Instead, it requires that witnesses preserve the order of conflicting events. Events are conflicting whenever one of them is a write on a variable, and the other is either a write on that variable or a read of that variable. This order also implies an order of transaction. A witness preserving the transaction order must exists, meaning it must be acyclic. Conflict serializability is prefix-closed. There cannot be a prefix of a history such that the conflict graph of the history is acyclic and the conflict graph of the witness is cyclic. This is because adding events to a history does not remove edges from the conflict graph [9].

Note that conflict serializability requires that there are atomic points where read and write events take place. For these examples, since we assume deferred update semantics, all writes of a transaction will take place during the commit operation. Here, we assume the point at which reads and commit take effect to be the response event of the respective operations.

In Figure 2.6 two examples illustrating conflict serializability are shown.

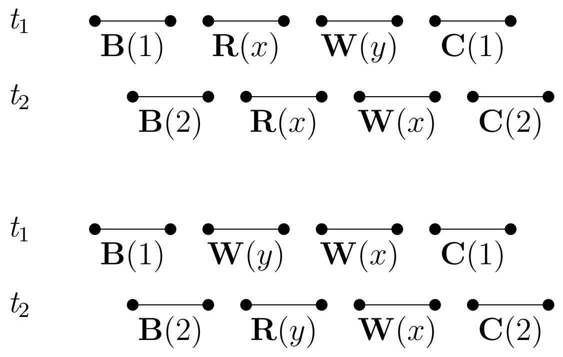


Figure 2.6: Conflict serializability examples, top correct, bottom incorrect

The top example is correct. This is proven by the witness with transaction order $1, 2$. This witness is possible as the read of transaction 1 on x happened before the commit of transaction 2. The bottom example is incorrect as in any correct witness transaction 1 must happen before transaction 2 as both write to x. However, transaction 1 must also happen after transaction 2 as transaction 1 writes to y and transaction 2 reads from y. Both cannot be the case at the same time in a serial history. Thus, a correct witness does not exist.

Strict (state) serializability, also introduced by Papadimitriou in 1979, is an expansion of state serializability [73]. A history is strictly serializable if in addition to the requirements of state serializability the witness preserves the order of non overlapping transactions (called *real-time order*). It also uses the most-recent reads-from relation. By the same argument as for state serializability, strict state serializability is not prefix-closed. The requirement to preserve the real-time order can also be added to view serializability, which is then called *strict view serializability.*

Two examples illustrating strict state serializability can be found in Figure 2.7.

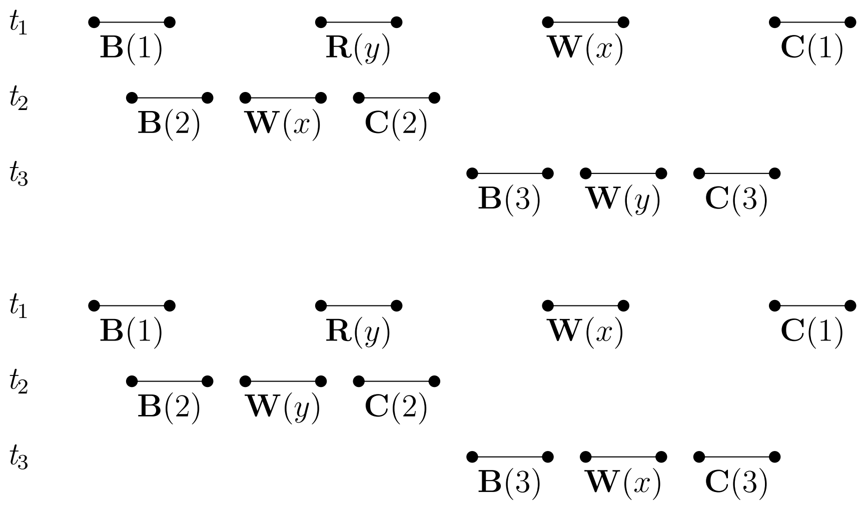


Figure 2.7: Strict serializability examples, top correct, bottom incorrect

The top example is correct as the witness with transaction order $2, 1, 3$ has an identical end state to the original history and preserves the real-time order between transaction 2 and transaction 3. The bottom example is incorrect. A witness with an identical end state must have transaction 2 happen after transaction 1, so transaction 1 still reads y from the initial state. But also transaction 3 must happen before transaction 1 so 1 is still the most recent writer on x before the end state. Additionally, transaction 2 must happen before transaction 3 as they are real-time ordered. Thus, any witness must have the transaction order $3, 1, 2, 3$, which is not possible in a serial history. So there does not exist a witness with the same end state as the original history.

Causal serializability is a value-based definition published a notable amount later (1997) than the definitions above [80]. It uses the unambiguous value-based reads-from relation. Its main feature is that it is only required for each thread to have its own witness. This means different threads do not have to agree on a witness. Also, in a witness for a thread, only transactions of that thread are required to read from the same transactions as in the original history. In addition to this, for each variable all witnesses must agree on a total order of transactions writing to this variable.

We illustrate this in Figure 2.8.

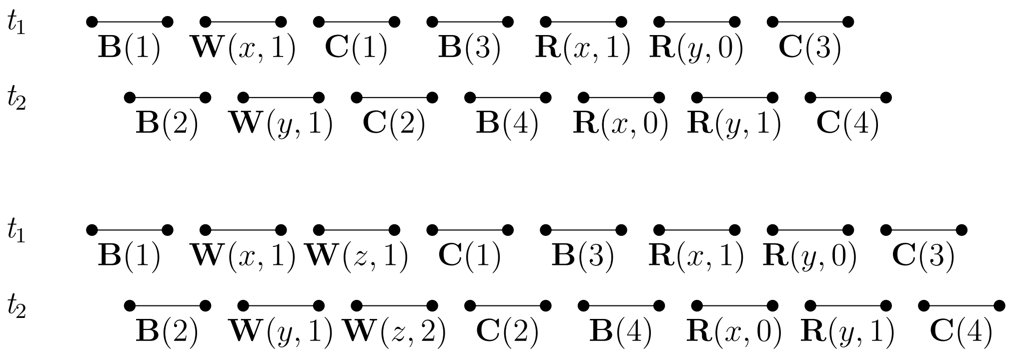

t_1
$\mathbf{B}(1)$ $\mathbf{W}(x,1)$ $\mathbf{C}(1)$ $\mathbf{B}(3)$ $\mathbf{R}(x,1)$ $\mathbf{R}(y,0)$ $\mathbf{C}(3)$
t_2
$\mathbf{B}(2)$ $\mathbf{W}(y,1)$ $\mathbf{C}(2)$ $\mathbf{B}(4)$ $\mathbf{R}(x,0)$ $\mathbf{R}(y,1)$ $\mathbf{C}(4)$
t_1
$\mathbf{B}(1)$ $\mathbf{W}(x,1)$ $\mathbf{W}(z,1)$ $\mathbf{C}(1)$ $\mathbf{B}(3)$ $\mathbf{R}(x,1)$ $\mathbf{R}(y,0)$ $\mathbf{C}(3)$
t_2
$\mathbf{B}(2)$ $\mathbf{W}(y,1)$ $\mathbf{W}(z,2)$ $\mathbf{C}(2)$ $\mathbf{B}(4)$ $\mathbf{R}(x,0)$ $\mathbf{R}(y,1)$ $\mathbf{C}(4)$

Figure 2.8: Causal serializability examples, top correct, bottom incorrect

The top history is correct as for thread 1 in the witness with transaction order $1, 3, 2, 4$ each read made in thread 1 is legal. The same holds for thread 2 for the witness with transaction order $2, 4, 1, 3$. The bottom history, on the other hand, is incorrect as now transactions 1 and 2 write on z. This means they must have the same order in the witness for each thread, which makes finding witnesses for both threads impossible.

Snapshot isolation is also a database correctness condition [8], which has been used for TMs [81]. As the name suggests, each transaction is supposed to take a snapshot of the memory, usually at the start of its execution. During its execution it only reads from this snapshot. So the only two reads-from relations compatible with snapshot isolation are the unambiguous value-based reads-from relation and the ambiguous value-based reads-from relation as one of the ideas of snapshot isolation is that a transaction is not necessarily reading the most recent values.

If a transaction t has written to variables, to which another already committed transaction concurrent to t also wrote to, then if t tries to commit it must abort. This is called the first-committer wins principle. As witnesses are not useful to model snapshot isolation, we will discuss the examples for snapshot isolation, shown in Figure 2.9, without referring to them.
The top example is correct as both transactions see a snapshot of the memory where $x = 0$ and $y = 0$. They are concurrent, but do not write on the same variables; thus no transaction needs to abort. The bottom example is incorrect

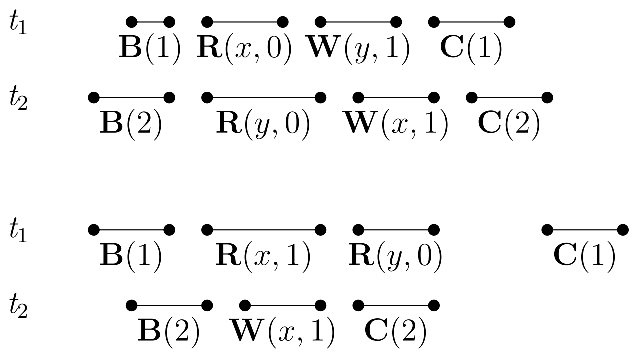

t_1
$\mathbf{B}(1)$ $\mathbf{R}(x,0)$ $\mathbf{W}(y,1)$ $\mathbf{C}(1)$
t_2
$\mathbf{B}(2)$ $\mathbf{R}(y,0)$ $\mathbf{W}(x,1)$ $\mathbf{C}(2)$
t_1
$\mathbf{B}(1)$ $\mathbf{R}(x,1)$ $\mathbf{R}(y,0)$ $\mathbf{C}(1)$
t_2
$\mathbf{B}(2)$ $\mathbf{W}(x,1)$ $\mathbf{C}(2)$

Figure 2.9: Snapshot isolation examples, top correct, bottom incorrect

as transaction 1 sees the value written by transaction 2, which is concurrent to it, and thus cannot be from a snapshot at the start of transaction 1.

Correctness conditions specifically designed for TMs The next step in the formalization of correctness conditions for TMs was considering that they are not used for databases but for programs. Guerraoui and Kapalka observed that even reads of aborted transactions matter in this context [46].

The issue is that if an aborted transaction has seen unexpected inconsistent values, then exceptions, infinite loops or similar might occur. They proposed a view based property *(value) opacity* that was meant to improve upon previous correctness conditions with regard to this. Opacity has 3 defining characteristics stated by the authors. First, all committed and under circumstances even commit pending transactions appear to happen atomically. Second, all aborted transactions are not read by other transactions. And third, each transaction no matter if aborted, live or committed sees a consistent state of the shared memory. For the witness that means it contains all transactions of the original history and each live and not commit pending transaction is completed to an aborted transaction. Commit pending transactions may either be completed to committed transactions or to aborted transactions. This is supposed to mirror that it is not known at which point during the commit operation a transaction becomes visible to other transactions. The witness must also preserve the real-time order of the original history. Value opacity explicitly uses an ambiguous value-based reads-from relation, meaning in this witness the reads of each transaction must be legal and not be from aborted transactions.

Two examples can be found in Figure 2.10.

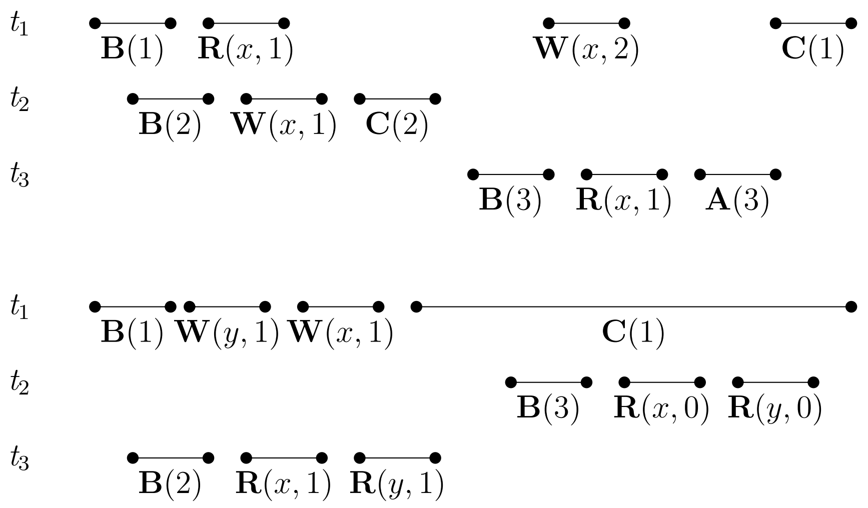


Figure 2.10: (Value) opacity examples, top correct, bottom incorrect

In the first example the witness with transaction order $2, 3, 1$ proves the opacity of the example. Note that this also shows that the original version of opacity is not prefix-closed. Because of this, works using opacity usually explicitly define it to be prefix-closed. In the second example transaction 3 is real-time ordered after transaction 2, so they have to be ordered this way in a witness as well. The only witness where the read of transaction 2 can be legal is one with transaction order $1, 2, 3$. However, in that case the reads of transaction 3 are not legal. Thus, no witness does exist for this history and it is not opaque.

Conflict opacity, presented by Guerraoui et al. in 2010, is a variation of value opacity [42]. Its relationship to value opacity is similar to the relationship of conflict serializability to view serializability. In the original work the authors referred to it as opacity, but it is not identical to value opacity. We discuss this in more detail in Section 3.3. As with conflict serializability, it does not take values into account and conflicting actions are defined identically. But witnesses now also must include aborted transactions. However, aborted transactions are treated as having no writes for the calculation of conflicts. Conflict opacity also takes the real-time order of histories into account.

Two examples are shown in Figure 2.11. The corresponding conflict graphs can be found in Figure 2.12.

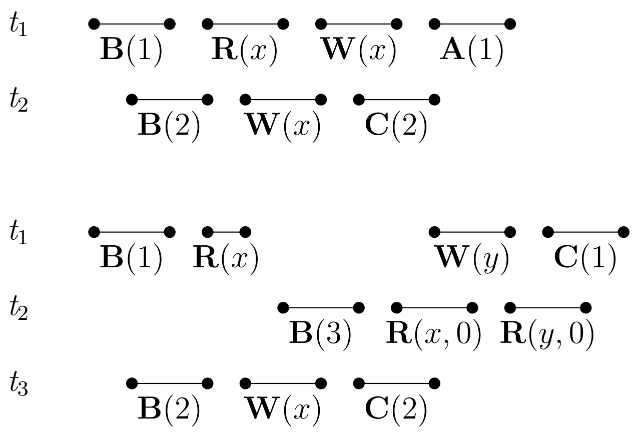
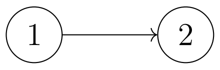


Figure 2.11: Conflict opacity examples, top correct, bottom incorrect

Note that again we assume each operation to take place at its response event and each commit to execute the writes of a transaction. The first example is correct as in a witness transaction 2 must be ordered after transaction 1 since the former writes to x after the latter reads from x. Because transaction 1 is aborted, its write on x causes no conflicts. Thus, the witness with the transaction order $1, 2$ proves the conflict opacity of the original history. The second example is not conflict opaque. In a correct witness transaction 2 is ordered after transaction 1 since it writes to x after transaction 1 reads x. Transaction 3 is ordered after transaction 2 since it reads x after 2 writes to x. By transitivity, it is then also ordered after transaction 1. But transaction 3 is also ordered before transaction 1 since it reads y before transaction 1 writes to y. Thus, any correct witness must have transaction order $3, 1, 2, 3$. This is a cyclic ordering for which there exists no witness, so the history is not conflict opaque.

Deferred-update opacity (DU-opacity), introduced by Attiya et al., is a variant of opacity that requires each transaction to read values from transactions that are either committed or commit pending at the point of the read [5]. This represents



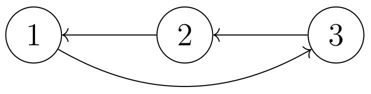
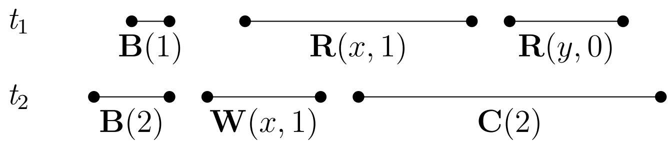


Figure 2.12: Conflict graphs for histories(top on the left, bottom on the right) of Fig. 2.11

the behaviour of deferred-update TMs where it is not correct for values of non committing or not committed transactions to be visible to other transactions. This variant is prefix-closed. Witnesses have the same constraints as for opacity and an ambiguous value-based reads-from relation is used. Additionally, each read must be legal when only considering transactions that are commit pending or committed at the point of the response of the read in the original history.

Two examples can be found in Figure 2.13.



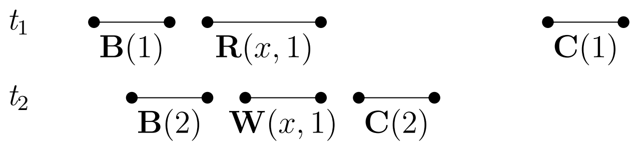


Figure 2.13: DU-Opacity examples, top correct, bottom incorrect

The top example is correct as can be seen by considering the witness with transaction order $1, 2$. Both reads are legal in this witness. Transaction 1 is considered for them as it started committing before the end of both reads. The bottom example is not correct. Although the witness with transaction order $2, 1$ seems to be correct, when checking the read of transaction 1 for legality transaction 2 is not considered as it has not started committing in the original history when the read ended. Thus, the read is not legal and no valid witness exists for this history.

In 2012 Imbs and Raynal presented a relaxation of value opacity called *virtual world consistency* (VWC) [53]. It is defined on histories that only contain aborted or committed transactions. While opacity assumes that the complete execution of all transactions must be equivalent to a sequential execution, VWC only requires this for committed transactions. This is because aborted transactions do not need to see a system state that agrees with the sequential execution in the overall witness as they have no effect on the shared memory. They only have to see some consistent values that could have been produced by the transactions before them to avoid errors. So instead of only having one witness, there is one witness for all committed transactions and one witness for each aborted transaction. The witness for the committed transaction has the same constraints as with opacity with the exception that an unambiguous value-based reads-from relation is used. The witness must be legal, which is easy to determine as it only contains committed transactions. The witness for each aborted transaction only contains its causal past, i.e. committed transactions of the same thread ordered before it, (committed) transactions it reads from and any committed transactions ordered to transactions in the causal past by the previous two ways. Note that these witnesses do not need to agree on a transaction order. These witnesses then must be legal. Notably, VWC does not require witnesses to preserve the real-time order of transactions.

Two examples can be found in Figure 2.14.

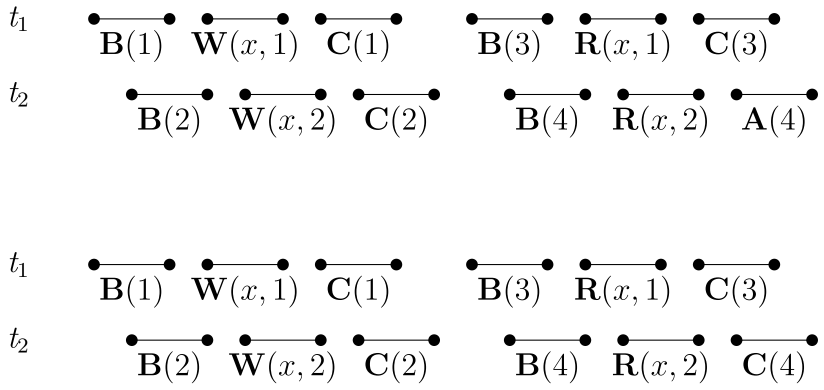

t_1
$\mathbf{B}(1)$ $\mathbf{W}(x,1)$ $\mathbf{C}(1)$ $\mathbf{B}(3)$ $\mathbf{R}(x,1)$ $\mathbf{C}(3)$
t_2
$\mathbf{B}(2)$ $\mathbf{W}(x,2)$ $\mathbf{C}(2)$ $\mathbf{B}(4)$ $\mathbf{R}(x,2)$ $\mathbf{A}(4)$
t_1
$\mathbf{B}(1)$ $\mathbf{W}(x,1)$ $\mathbf{C}(1)$ $\mathbf{B}(3)$ $\mathbf{R}(x,1)$ $\mathbf{C}(3)$
t_2
$\mathbf{B}(2)$ $\mathbf{W}(x,2)$ $\mathbf{C}(2)$ $\mathbf{B}(4)$ $\mathbf{R}(x,2)$ $\mathbf{C}(4)$

Figure 2.14: VWC examples, top correct, bottom incorrect

The top example is correct. The witness for all committed transactions has the transaction order $1, 2, 4$. For the aborted transaction 4 its witness has transaction order $2, 3, 4$ as there are no other transactions in its causal past. The bottom example is incorrect as now transaction 4 is committed as well; thus there trivially is no legal witness for all committed transactions.

In 2013 Doherty et al. presented *Transactional Memory Specification* (TMS) 1 and 2. Notably it is defined as an I/O automaton which is atypical for correctness conditions. The idea behind this was to enable easier proofs of correctness for TMs, for example via forward simulation. This also makes it prefix-closed by design. It is also defined for general specifications and operations, not only read/write objects. However, as before, we will describe it in this context. Also, both do not require unique values for each write to a variable. The design of TMS 1 was overlapping with the design of VWC and had similar goals, especially allowing for individual witnesses to justify the reads of aborted transactions. Overall, it requires that there is at least one serialization consisting of all committed transactions at any point of an execution. Additionally, for each live or aborted transaction there must be one serialization including the transaction itself, a number of transactions that were at some point commit invoked, and all transactions that are committed and real-time ordered before any transaction in the former two. This serialization may contain aborted transactions that commit invoked at some point. The intuition behind allowing this is that if an aborted transaction was allowed to invoke a commit in the first place then its values when written to the shared memory do not cause inconsistencies (except if the program itself was bugged). As long as the reading transaction does not commit, the overall execution stays correct.

Figure 2.15 shows two examples.

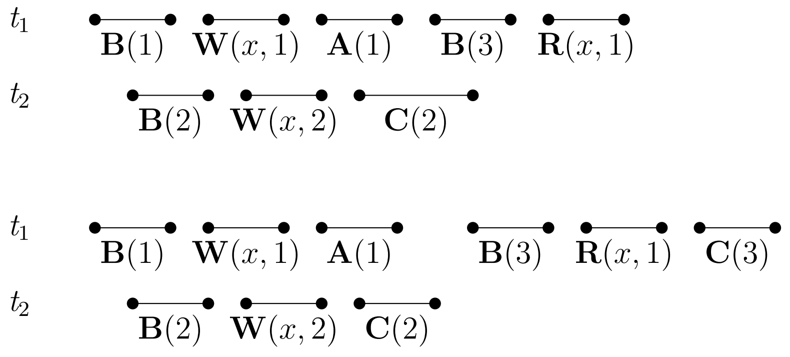


Figure 2.15: TMS 1 examples, top correct, bottom incorrect

The top example is correct, there is trivially a witness containing all committed transactions. For the aborted transaction 1 there exists a legal witness only containing itself. For the live transaction 3 consider the witness with transaction order $1, 3$. Transaction 2 does not need to be contained as it is not real-time ordered before transaction 1 or transaction 3. This witness is legal as transaction 3 reads transaction 1. The bottom example is incorrect as there is no witness containing all committed transactions. This is because in this example transaction 3 is committed. Thus, the overall witness must contain it, and it does not contain transaction 1; thus overall it is not legal.

TMS 2 is a stronger version of TMS 1 designed to be close to typical deferred update implementations of TMs. The I/O automaton is designed to correspond to the operations usually used by TMs. Notably, any transaction being read by another transaction must be committed or commit later on, and aborted transactions may not be read. We will not use witnesses to describe TMS 2 as it is far easier to understand by explaining it as a specification of behaviour. It is assumed implicitly that at each point there is a fixed memory state. A writing transaction updates variables somewhere in its commit operation, after which the transaction always commits. A transaction must read from a memory state that existed during its execution. This means that aborted transactions cannot justify a read of a live or aborted transactions. If a transaction is a writer, all of its reads must be the current memory state when it commits.

Two examples can be found in Figure 2.16.

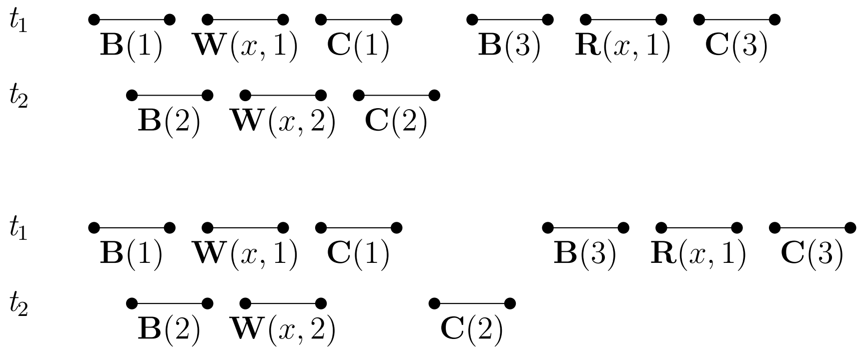


Figure 2.16: TMS 2 examples, top correct, bottom incorrect

The top example is correct as transaction 2 may have updated the memory before transaction 1 as both commits overlap. Thus, transaction 3 may read from transaction 1. In the second example the commit of transaction 2 does not overlap the commit of transaction 1, and happens later in the history. Thus, transaction 3 may not read from transaction 1 as during its execution the value of x was 2 without ambiguity.

2.3 Basic Notation and Terms

In this section, we will introduce the formal notation and the definitions needed for the remainder of the thesis. We will first present a block of general definitions, then all opacity related definitions and finally all serializability related definitions.

2.3.1 General Definitions

These definitions are the building blocks for any other definition and include events and transactions.

Events We will first define *events*, which represent operations inside a transactional system. Operations consist of an invoke event and a return event.
Operations are executed by *threads* and can operate on *variables* by reading and writing *values*. Sequences of events of the same thread may be grouped into *transactions* (see Section 2.3.1). An overview of the IDs of these sets, variable naming conventions of their elements and instances of them used in examples is shown in Table 2.1.

All sets except the set of transactions are assumed to be finite in this thesis. Assuming the transaction set to be infinite is necessary to make modeling implementations feasible. If the transaction set is finite, for any instance of the correctness problem the number of transactions in any execution produced by the given automaton is finite. This is not a desirable property as such an automaton is supposed to model a TM and an upper bound to the number of transactions is not a typical property of implementations. But this assumption also makes it necessary to not formally include transaction identifiers in events. Otherwise,

Name	Set ID	Variable IDs	Instances
Threads	T	$t, t_1, \ldots$	$t, t_1,$
Transactions	Tr	$tr, tr_1, \ldots$	$11, 12, \ldots, 21, 22, \ldots$
Variables	Var	$var, var_1, \ldots, x, y, z \ldots$	$x, y, z \ldots$
Values	Val	$val, val_1, \ldots$	$1, 2, 3 \ldots$

Table 2.1: Set IDs and variable IDs for basic sets

TMs producing histories with arbitrarily many transactions cannot be modeled by finite automata, as the events of different transactions must be produced by different transitions. This is an issue as such an automaton being finite is a requirement for the definitions of correctness problems later in this section. As we will see later, for any given history (see Section 2.3.2) there is only one way excluding isomorphisms of grouping of events into transactions. We still include transaction identifiers as a superscript if helpful.
There are two types of operations, transaction related operations and object related operations. Transaction related operations are beginning, committing and aborting a transaction. As we are concerned with read/write objects, object related operations are reading from a variable and writing to a variable. For all operations the transactional memory returns one of the following responses:

- a success,
- an abort
- or a success and a value (if the operation is a read).

An abort signals that the operation failed and in addition the overall transaction has been aborted, while a success indicates that the corresponding operation has succeeded. See Table 2.2 for an overview of the invoke response event notation. As discussed, events do not need to explicitly have their transaction as an index; thus, in Table 2.3 the transaction identifiers are marked red. If it is not appropriate or helpful, we will omit the transaction identifier, e.g. $\mathbf{Inv}_t(\mathbf{B})$ instead of $\mathbf{Inv}_t^{tr}(\mathbf{B})$. For some definitions operations are contracted into a single event without an invoke and a response [46, 42], if it is not relevant for the correctness condition.

Invocations	Possible matching responses
$\mathbf{Inv}_t^{tr}(\mathbf{B})$	$\mathbf{Resp}_t^{tr}(\mathbf{B}), \mathbf{A}_t^{tr}$
$\mathbf{Inv}_t^{tr}(\mathbf{R}(var))$	$\mathbf{Resp}_t^{tr}(\mathbf{R}(var, val)), \mathbf{A}_t^{tr}$
$\mathbf{Inv}_t^{tr}(\mathbf{W}(var, val))$	$\mathbf{Resp}_t^{tr}(\mathbf{W}(var)), \mathbf{A}_t^{tr}$
$\mathbf{Inv}_t^{tr}(\mathbf{C})$	$\mathbf{Resp}_t^{tr}(\mathbf{C}), \mathbf{A}_t^{tr}$
$\mathbf{Inv}_t^{tr}(\mathbf{A})$	$\mathbf{A}_t^{tr}$

Table 2.2: Events of TM algorithms

Original Sequence	Abbreviation
$\mathbf{Inv}_t^{tr}(\mathbf{B})\mathbf{Resp}_t^{tr}(\mathbf{B})$	$\mathbf{B}_t^{tr}$
$\mathbf{Inv}_t^{tr}(\mathbf{R}(var))\mathbf{Resp}_t^{tr}(\mathbf{R}(var, val))$	$\mathbf{R}_t^{tr}(var, val)/\mathbf{R}_t^{tr}(var)$
$\mathbf{Inv}_t^{tr}(\mathbf{W}(var, val))\mathbf{Resp}_t^{tr}(\mathbf{W}(var))$	$\mathbf{W}_t^{tr}(var, val)/\mathbf{W}_t^{tr}(var)$
$\mathbf{Inv}_t^{tr}(\mathbf{C})\mathbf{Resp}_t^{tr}(\mathbf{C})$	$\mathbf{C}_t^{tr}$
$\mathbf{Inv}_t^{tr}(\mathbf{B})\mathbf{A}_t^{tr}$	ϵ
(Any Invoke except $\mathbf{Inv}_t^{tr}(\mathbf{B})$)$\mathbf{A}_t^{tr}$	$\mathbf{A}_t^{tr}$

Table 2.3: Event abbreviations

Each operation with a positive response is abbreviated into a single event of that operation. An aborted begin operation is abbreviated with the empty word ϵ as it is not relevant for the correctness conditions discussed in this thesis. Each operation with an abort response is abbreviated into an abort. For conflict-based conditions values are usually omitted as they are not of relevance. See Table 2.3 for an overview of these abbreviations. We denote the set of all events by Ev members of this set are denoted $ev, ev', ev_1, ev_2 \dots$. If an event ev is executed by thread t, we write $t(ev) = t$.

Syntax of transactions A transaction is a sequence of events which obeys a specific format. A transaction can only belong to one thread. In a transaction each invoke must be followed by an appropriate response as the next event. We thus use the shortened event notation for the following definitions. A *finished* transaction starts with a begin, executes an arbitrary number of writes and reads and then either tries successfully to commit or aborts/gets aborted. Any prefix

$$\begin{aligned}
tr_1 &= \mathbf{B}_{t_1}^{tr_1}\mathbf{W}_{t_1}^{tr_1}(x, 1)\\
tr_2 &= \mathbf{B}_{t_1}^{tr_2}\mathbf{W}_{t_1}^{tr_2}(y, 2)\mathbf{A}_{t_1}^{tr_2}\\
tr_3 &= \mathbf{B}_{t_1}^{tr_3}\mathbf{R}_{t_1}^{tr_3}(x, 1)\mathbf{W}_{t_1}^{tr_3}(z, 5)\mathbf{W}_{t_1}^{tr_3}(y, 2)\mathbf{R}_{t_1}^{tr_3}(w, 3)\mathbf{C}_{t_1}^{tr_3}
\end{aligned}$$

Figure 2.17: Transaction examples

of a finished transaction is an *unfinished* transaction.

Definition 1 (Transaction)**.** *A sequence of events is a transaction of thread t iff it is a prefix of a word generated by the following regular expression:*

$$\mathbf{B}_t(\mathbf{W}_t(var, val) \mid \mathbf{R}_t(var, val))^*(\mathbf{A}_t \mid \mathbf{C}_t),$$

where var and val represent the choice between any element of Var and Val, respectively.

Figure 2.17 shows three different example transactions. Transactions tr_2 and tr_3 are finished and transaction tr_1 is unfinished.

A transaction is called *committed* when it ends with a commit, respectively, *aborted* when ending with an abort. Transaction tr_2 is aborted, transaction tr_3 is committed.

As we are concerned with deferred update semantics, we assume that each transaction only writes once to each variable as at the end only one value can be written.

Assumption 1. *No transaction ever writes to the same variable twice.*

2.3.2 Opacity Related Definitions

We introduce the opacity related definitions by starting with *general-histories* (g-histories), the history type used for opacity definitions and similar definitions, and then the definitions of conflict opacity and value opacity. Then we will present the definitions related to value opacity including the membership and correctness problem for it, and then do the same for conflict opacity.

$$\begin{aligned}
h_1 &= \mathbf{Inv}_{t_1}(\mathbf{B})\mathbf{Resp}_{t_1}(\mathbf{B})\mathbf{Inv}_{t_1}(\mathbf{W}(x,2))\mathbf{Inv}_{t_2}(\mathbf{B})\mathbf{Resp}_{t_1}(\mathbf{W}(x))\mathbf{Resp}_{t_2}(\mathbf{B})\mathbf{Inv}_{t_1}(\mathbf{R}(x))\mathbf{Resp}_{t_1}(\mathbf{R}(x,2))\\
h_2 &= \mathbf{Inv}_{t_1}(\mathbf{C})\mathbf{Inv}_{t_1}(\mathbf{B})\mathbf{Inv}_{t_1}(\mathbf{W}(x,2))\mathbf{Inv}_{t_1}(\mathbf{B})\\
h_3 &= \mathbf{B}_{t_1}\mathbf{R}_{t_1}(x,2)\mathbf{B}_{t_2}\mathbf{W}_{t_2}(y,1)\mathbf{C}_{t_2}\mathbf{C}_{t_1}\\
h_4 &= \mathbf{B}^{11}_{t_1}\mathbf{R}^{11}_{t_1}(x,2)\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(y,1)\mathbf{C}^{21}_{t_2}\mathbf{C}^{11}_{t_1}\\
h_5 &= \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(y,1)\mathbf{R}^{11}_{t_1}(x,2)\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(x,3)\mathbf{C}^{21}_{t_2}\mathbf{C}^{11}_{t_1}\mathbf{B}^{12}_{t_1}\mathbf{R}^{12}_{t_1}(y,1)\mathbf{C}^{12}_{t_1}
\end{aligned}$$

Figure 2.18: Example for g-histories

General Histories A general-history (g-history) is a sequence of operations of different threads, in which the operations are grouped into transactions. In the context of opacity, the invokes and responses for begins, writes, reads and aborts are treated as happening atomically one after another. For commits, on the other hand, responses are not treated as being atomic. This means the $\mathbf{Inv}(\mathbf{C})$ invoke and its possible responses $\mathbf{A}$ and $\mathbf{Resp}(\mathbf{C})$ are used. For begin, write, read and abort invokes and responses, the abbreviations shown in Table 2.3 are used. For simplicity, in the context of opacity, we call the set of these events also Ev.

Definition 2 (G-History)**.** *A g-history is a sequence of events $ev_0 \dots ev_n$, where for all i, $0 \leq i \leq n$, it holds that $ev_i \in Ev$.*

Figure 2.18 shows multiple g-histories, for most of them the abbreviated version of events is used for brevity.

For such a sequence of events indexed by threads, up to isomorphism there is not more than one way to assign transactions identifiers to events, whenever the g-history is *well-formed*.

In a well-formed g-history the projection of the g-history onto each thread forms a sequence of transactions, where only the last transaction may be unfinished. This avoids nonsense g-histories where for example two begins of the same thread follow each other, or a transaction commits twice. The set of all well-formed g-histories is denoted $\mathcal{H}$. In the example h_1 is well-formed while h_2 is not. In the examples in Figure 2.18, each color represents a transaction in all well-formed g-histories. Given an event ev, its transaction in a g-history h is denoted $tr_h(ev)$.

We also need to ensure that the naming of transactions over multiple g-histories is consistent to make sure g-histories are comparable. If the projection upon one thread is identical for both g-histories, then two transactions are named identically if they are at the same posirpjrption in the projection. This significantly simplifies comparing histories in the context of this thesis. Thus, in this thesis, transactions are identified by two arguments: their thread and the current count of transactions of that thread.

Our examples never contain more than 9 transactions per thread or 9 threads, so we use a two digit simplified notation. The first digit denotes the thread and the second one the transaction. In the example h_4 shows the assignment of these simplified thread identifiers to h_3. We can now define the g-history independent function $thr : tr \rightarrow t$ which maps each transaction onto its thread. For example, $thr(11)$ returns t_1.

For a transaction tr and g-history h, we denote the case that tr is unfinished as $unfin_h(tr)$. The set of all unfinished transactions in g-history h is denoted $unfin(h)$. If a transaction tr in a g-history h ends on a commit, we call it *committed* and denote it $co_h(tr)$. If a transaction tr in a g-history h ends on an abort, we call it *aborted* and denote it $ab_h(tr)$. If it ends on a commit invoke, we call it commit pending and denote it $comP_h(tr)$. If a transaction is finished, we denote it $fin_{ph}(tr)$.

Two important notions for transactions in the context of a g-history are the *write set* and the *read set*. The write set of transaction tr in a g-history h is denoted $WS_h(tr) \subseteq Var \times Val$. It is the set of all variable and value pairs such that the transaction contains a corresponding write event for each of them. The read set $RS_h(tr)$ is defined similarly. If the values are not relevant, both are simply a set of variables and are denoted $WS_h^{vo}(tr)$ and $RS_h^{vo}(tr)$ (vo stands for variable-only), respectively. The write set for transaction 21 in h_4 is $WS_{h_4}(21) = \{(y, 1)\}$, and its write set without values is $WS_{h_4}^{vo}(21) = \{var_2\}$. The read set for transaction 11 in h_4 is $RS_{h_4}(11) = \{(var, val)\}$, and its write set without values is $RS_{h_4}^{vo}(11) = \{var\}$.

The set of all transactions of a g-history h is $tr(h)$, and we write $tr \in h$ if a transaction tr occurs in h. For h_4 it holds that $tr(h_4) = \{11, 21\}$ and also $11 \in h_4$. Given a event ev in g-history h, $tr_h(ev)$ denotes the transaction of event ev in h. In the example, it holds that $tr_{h_4}(\mathbf{B}_t^{11}) = 11$. If an event occurs in a g-history h, we write $ev \in h$. For example, $\mathbf{B}_t^{11} \in h_4$ holds. The set of events occuring in a

$$\begin{aligned}
h_1 &= \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(var, val_2)\mathbf{C}^{11}_{t_1}\mathbf{B}^{12}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(var, val)\mathbf{R}^{12}_{t_1}(var, val)\mathbf{Inv}^{21}_{t_2}(\mathbf{C})\\
h_2 &= \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(var, val_2)\mathbf{C}^{11}_{t_1}\mathbf{B}^{12}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(var, val)\mathbf{R}^{12}_{t_1}(var, val)\\
h_3 &= \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(var, val_2)\mathbf{C}^{11}_{t_1}\mathbf{B}^{12}_{t_1}\mathbf{W}^{12}_{t_2}(var, val_1)\mathbf{C}^{12}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{R}^{21}_{t_2}(var, val_2)\mathbf{C}^{21}_{t_2}\\
h_{1,c} &= \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(var, val_2)\mathbf{C}^{11}_{t_1}\mathbf{B}^{12}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(var, val)\mathbf{R}^{12}_{t_1}(var, val)\mathbf{Inv}^{21}_{t_2}(\mathbf{C})\mathbf{Resp}^{21}_{t_2}(\mathbf{C})\mathbf{A}^{12}_{t_1}\\
h_{2,c} &= \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(var, val_2)\mathbf{C}^{11}_{t_1}\mathbf{B}^{12}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(var, val)\mathbf{R}^{12}_{t_1}(var, val)\mathbf{A}^{12}_{t_1}\mathbf{Inv}^{21}_{t_2}(\mathbf{C})\mathbf{A}^{21}_{t_2}\\
h_{1,s} &= \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(var, val_2)\mathbf{C}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(var, val)\mathbf{C}^{21}_{t_2}\mathbf{B}^{12}_{t_1}\mathbf{R}^{12}_{t_1}(var, val)\mathbf{A}^{12}_{t_1}
\end{aligned}$$

Figure 2.19: Example histories for value opacity

g-history h is denoted $Ev(h)$. If the event ev is ordered before another event ev' in g-history h, we write $ev <_h ev'$. In the example, $\mathbf{B}^{11}_{t_1} <_{h_4} \mathbf{B}^{21}_{t_2}$ is true.

Given two g-histories h and h', the concatenation of both is denoted $h \cdot h'$. To simplifiy notation when given multiple g-histories as a set $\mathcal{H}' = \{h_0, \ldots, h_n\}$, we use $\bullet$ analogue to a sum symbol for concatenation, for example $\underset{h \in \mathcal{H}'}{\bullet} h$ or $\underset{0 \leq k \leq n}{\bullet} h_k$.

Also, for two g-histories we write $h \preceq h'$ if h is a prefix of h', and $h \sqsubseteq h'$ if h is a subsequence of h'. In the example $h_4 = \mathbf{B}^{11}_{t_1}\mathbf{R}^{11}_{t_1}(x, 2) \cdot \mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(y, 1)\mathbf{C}^{21}_{t_2}\mathbf{C}^{11}_{t_1}$, $\mathbf{B}^{11}_{t_1}\mathbf{R}^{11}_{t_1}(var, val) \preceq h_4$, and $\mathbf{B}^{11}_{t_1}\mathbf{W}^{21}_{t_2}(y, 1) \sqsubseteq h_4$ hold.

Two transactions tr_1, tr_2 are *real-time ordered* in a g-history h, denoted $tr_1 \prec_h tr_2$, when the commit or abort event of tr_1 occurs before the begin event of tr_2. The real-time order of h, $h.RT \subseteq Tr \times Tr$, contains all pairs (tr_1, tr_2) where $tr_1 \prec_h tr_2$. These pairs are also called *real-time elements* or in short rt-elements. For example, in h_5 it holds that $11 \prec_{h_5} 12$. A g-history is called serial if all events of a transaction directly follow each other.

Value opacity A general description of value opacity was given in Section 2.2. Here we will only give the definitions needed to formalise it and the definition of its membership and correctness problem.

To define opacity we first need to define the notion of legality for a serial history, equivalence and the completion of a g-history. Then we will define opacity and explain it along the examples shown in Figure 2.19. A serial g-history where each transaction except the last one is committed and the last transaction can either be committed or aborted is *legal* whenever each object adheres to its specification. For read/write objects, adhering to their specification means that for each variable the most recently written value is read by a read on that variable. For the case that no most recent write exists, an arbitrary initialization value is read. This value must be consistent for all reads with no most recent writer and not be written to any variable by any transaction. A transaction is legal in a serial g-history whenever the subhistory consisting of itself and all preceding committed transactions is legal. Lastly, let $h|_t$ be the projection of h onto all events of thread t. Two g-histories are *equivalent* whenever they contain the same events and each thread executes the same transactions in the same order.

Definition 3 (Equivalence)**.** *Two g-histories h_1 and h_2 are equivalent whenever*

$$tr(h_1) = tr(h_2) \text{ and } \forall t \in T : h_1|_t = h_2|_t.$$

Given two g-histories h_1, h_2 that are equivalent, we say that h_1 preserves the real-time order of h_2 whenever $h_2.RT \subseteq h_1.RT$.

A completion of a g-history h is a g-history where each unfinished transaction of h is aborted by inserting an abort event, unless it is commit pending. In this case it is either aborted or committed by inserting the corresponding response event. All the new events in the completion are inserted at some point after the last event of their respective transactions. The set of completions for a given g-history h is denoted $compl(h)$. In the example, two possible completions of h_1 are $h_{1,c}$ and $h_{2,c}$

Definition 4 (*OP* [46])**.** *A g-history h is opaque whenever there exists a serial g-history h_s such that*

1. *h_s is equivalent to a g-history in $compl(h)$,*
2. *each transaction in h_s is legal in the subhistory of h_s containing all committed transactions before the transaction*
3. *and h_s preserves the real-time order of h.*

For simplicity, we also refer to the second condition as h_s being legal.

A serial g-history h_s meeting the requirements of the above definition for a g-history h is called an *OP*-witness of h. In Figure 2.19 history h_1 is opaque as $h_{1,s}$ is an OP-witness. Note that transaction 12 reads a value from 21. This is permitted as the latter has issued a commit invoke event. For g-history h_2 no such witness exists as transaction 21 is not commit pending, and thus will be aborted in any completed g-history. This means the read of transaction 12 cannot be justified in any witness. Also, for h_3 no such witness exists as transaction 21 reads a value from transaction 11. This is not permitted as this value is overwritten by transaction 12 which is real-time ordered after transaction 11 and before transaction 21. Let $OW(h)$ be the set of all OP-witnesses for a g-history h. As mentioned previously, the definition is not prefix-closed. The original authors argued that since STM histories are generated progressively each subhistory must be opaque for the STM to be opaque [46]. While this is a reasonable argument for the correctness problem, for the membership problem where we are given arbitrary g-histories this is not a reasonable assumption. For the membership problem we can prove our results also for the original not prefix-closed version of value opacity.

Going forward, we also assume that, in the context of value opacity, the same value is not read twice or more from the same variable by a single transaction, if a history contains values. This comes w.l.o.g. as we argue next. Consider a g-history h with a transaction having two reads from the same variable reading the same value (for short duplicate reads). Let h_s be a witness for it, it is easily provable that h_s with one of the duplicate reads removed is a witness for h with this read also removed. This is because the witness is equivalent to a completion of the new history (one read was removed for both), the real-time order is still preserved (removing reads does not change the real-time order) and any legal transaction is still legal (witness unchanged except for the removal of a read). On the other hand, assume a g-history h and a witness h_s for it. If we add a duplicate read to a transaction in h, then h_s with this read inserted in the same transaction at the same position is a witness for the new g-history. It is trivially equivalent to a completion of the new g-history and preserves its real-time order. Each transaction is still legal as if the original read is legal, then its duplication is as well.

We will define the membership and correctness problem for value opacity next. The membership problem is the question whether a single history is opaque.

Problem 1 (Membership problem for OP)**.** *Given a g-history h, is h opaque?*

The correctness problem is the question whether a given implementation DFA I does only produce opaque histories.

Problem 2 (Correctness problem for OP)**.** *Given an implementation I, are all g-histories produced by I opaque?*

Conflict opacity Conflict Opacity (CO) was first defined by Guerraoui et al. [42]. It employs the notion of *conflicts* between transactions. Such conflicts determine an order of transactions called *conflict order*. Whenever this order is acyclic, it defines one or multiple serial histories. These are deemed equivalent in behaviour to the original history. So a history is *conflict opaque* whenever its conflict order is acyclic.

We will first present which conflicts can occur in a g-history. Let var be an arbitrary variable. Two transactions tr_1, tr_2 are in conflict whenever either

- tr_1 and tr_2 both write on var and commit, (W/W conflict)
- tr_1 writes on var and commits and tr_2 reads from var, (W/R conflict)
- tr_1 reads from var and tr_2 writes on var and commits (R/W conflict)
- or tr_1 is real-time ordered before tr_2. (RT conflict)

The order of the conflicting events also determines the resulting conflict order. As Guerraoui et al., we assume deferred update semantics, meaning writes occur at the commit event and reads occur at the read event. Figure 2.20 shows examples for conflicts and the conflict order they induce. Note that, as in the work of Guerraoui et al., we use valueless single events instead of invoke response notation. The simplified notation is shown in Table 2.3.

We differ in notation from Guerraoui et al. in that we use explicit begin events for overall consistency with other g-histories. In the original notation a transaction has no begin statement and implicitly begins with its first event. Both notation styles are equivalent as they can be converted into histories of the other style without changing their conflict order. Given a g-history in explicit begin notation,

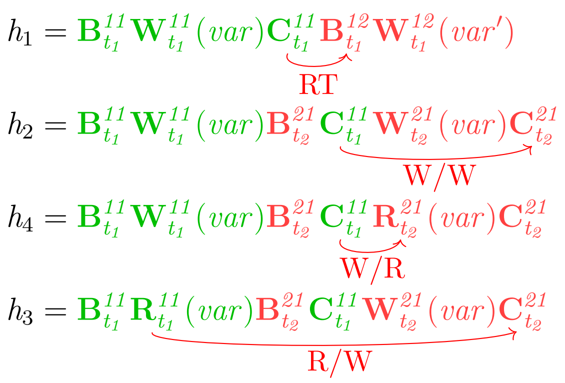

$h_1 = \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(var)\mathbf{C}^{11}_{t_1}\mathbf{B}^{12}_{t_1}\mathbf{W}^{12}_{t_1}(var')$
RT
$h_2 = \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(var)\mathbf{B}^{21}_{t_2}\mathbf{C}^{11}_{t_1}\mathbf{W}^{21}_{t_2}(var)\mathbf{C}^{21}_{t_2}$
W/W
$h_4 = \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(var)\mathbf{B}^{21}_{t_2}\mathbf{C}^{11}_{t_1}\mathbf{R}^{21}_{t_2}(var)\mathbf{C}^{21}_{t_2}$
W/R
$h_3 = \mathbf{B}^{11}_{t_1}\mathbf{R}^{11}_{t_1}(var)\mathbf{B}^{21}_{t_2}\mathbf{C}^{11}_{t_1}\mathbf{W}^{21}_{t_2}(var)\mathbf{C}^{21}_{t_2}$
R/W

Figure 2.20: Conflicts in conflict opacity

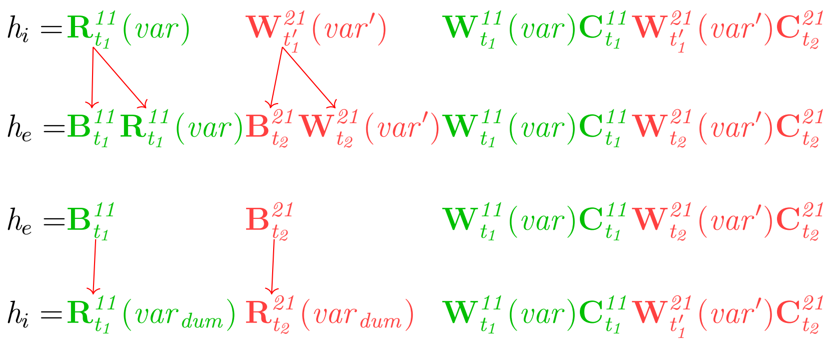

$h_i = \mathbf{R}^{11}_{t_1}(var) \quad \mathbf{W}^{21}_{t_1'}(var') \quad \mathbf{W}^{11}_{t_1}(var)\mathbf{C}^{11}_{t_1}\mathbf{W}^{21}_{t_1'}(var')\mathbf{C}^{21}_{t_2}$
$h_e = \mathbf{B}^{11}_{t_1}\mathbf{R}^{11}_{t_1}(var)\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(var')\mathbf{W}^{11}_{t_1}(var)\mathbf{C}^{11}_{t_1}\mathbf{W}^{21}_{t_2}(var')\mathbf{C}^{21}_{t_2}$
$h_e = \mathbf{B}^{11}_{t_1} \quad \mathbf{B}^{21}_{t_2} \quad \mathbf{W}^{11}_{t_1}(var)\mathbf{C}^{11}_{t_1}\mathbf{W}^{21}_{t_2}(var')\mathbf{C}^{21}_{t_2}$
$h_i = \mathbf{R}^{11}_{t_1}(var_{dum}) \quad \mathbf{R}^{21}_{t_2}(var_{dum}) \quad \mathbf{W}^{11}_{t_1}(var)\mathbf{C}^{11}_{t_1}\mathbf{W}^{21}_{t_1'}(var')\mathbf{C}^{21}_{t_2}$

Figure 2.21: Example g-history conversion from implicit to explicit begin events

the conversion removes the begin statement for each transaction and replaces it with a read of that transaction on a dummy variable. This dummy variable must be never written to in the original history, so it causes no additional R/W or W/R conflicts. By nature of being a read, it causes no W/W conflicts. It also causes no new RT conflicts as each transaction keeps their starting index in the history. Given a g-history in implicit begin notation, we add a begin event directly before the otherwise first event of each transaction. Figure 2.21 shows examples of these conversions. We are now ready to formally define the conflict order of a history and then conflict opacity itself.

Definition 5 (Conflict order)**.** *The conflict order of a g-history h, written as $<^{CO}_{h}$,*

$$\begin{aligned}
h_1 &= \mathbf{B}^{11}_{t_1}\mathbf{R}^{11}_{t_1}(var')\mathbf{W}^{11}_{t_1}(var)\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(var)\mathbf{C}^{21}_{t_2}\mathbf{C}^{11}_{t_1}\mathbf{B}^{12}_{t_1}\mathbf{R}^{12}_{t_1}(var)\mathbf{B}^{22}_{t_2}\mathbf{W}^{22}_{t_2}(var')\mathbf{C}^{12}_{t_1}\\
h_{1,s} &= \mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(var)\mathbf{C}^{21}_{t_2}\mathbf{B}^{11}_{t_1}\mathbf{R}^{11}_{t_1}(var')\mathbf{W}^{11}_{t_1}(var)\mathbf{C}^{11}_{t_1}\mathbf{B}^{22}_{t_2}\mathbf{W}^{22}_{t_2}(var')\mathbf{B}^{12}_{t_1}\mathbf{R}^{12}_{t_1}(var)\mathbf{C}^{12}_{t_1}\\
h_2 &= \mathbf{B}^{11}_{t_1}\mathbf{R}^{11}_{t_1}(var')\mathbf{W}^{11}_{t_1}(var)\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(var')\mathbf{W}^{21}_{t_2}(var'')\mathbf{C}^{21}_{t_2}\mathbf{B}^{22}_{t_2}\mathbf{R}^{22}_{t_2}(var'')\mathbf{W}^{22}_{t_2}(var)\mathbf{C}^{22}_{t_2}\mathbf{C}^{11}_{t_1}
\end{aligned}$$

Figure 2.22: Example histories for conflict opacity

is the union of the following orders:

1. *R/W order:*
   $\{(tr_1, tr_2) \mid tr_1, tr_2 \in h, \exists var \in RS^{vo}_h(tr_1) \cap WS^{vo}_h(tr_2), \mathbf{R}^{tr_1}_{thr(tr_1)}(var) <_h \mathbf{C}^{tr_2}_{thr(tr_2)}\}$,
2. *W/R order:*
   $\{(tr_1, tr_2) \mid tr_1, tr_2 \in h, \exists var \in RS^{vo}_h(tr_2) \cap WS^{vo}_h(tr_1), \mathbf{C}^{tr_1}_{thr(tr_1)} <_h \mathbf{R}^{tr_2}_{thr(tr_2)}(var)\}$,
3. *W/W order:*
   $\{(tr_1, tr_2) \mid tr_1, tr_2 \in h, \exists var \in WS^{vo}_h(tr_1) \cap WS^{vo}_h(tr_2), \mathbf{C}^{tr_1}_{thr(tr_1)} <_h \mathbf{C}^{tr_2}_{thr(tr_2)}\}$
4. *and RT order: $h.RT$.*

A conflict order can be visualized as a directed graph, called *conflict graph.* Note that we do not include edges that are already implied by transitivity. The conflict graph of a g-history h is denoted $CG(h)$. Each node in the graph corresponds to a transaction and each conflict (tr_1, tr_2) with $tr_1, tr_2 \in Tr$ is represented as an edge from tr_1 to tr_2. Conflict opacity is then defined as follows.

Definition 6 (CO)**.** *A g-history h is opaque under CO (alternatively conflict opaque), whenever $<^{CO}_h$ is acyclic.*

We will explain both definitions along the example. The conflict order of g-history h_1 in Figure 2.22 is:

$$\begin{aligned}
<^{CO}_{h_1} = \quad & \emptyset \quad \cup \{(11, 12), (21, 12)\} && \text{(R/W), (W/R)}\\
& \cup \{(21, 11)\} \cup \{(11, 12), (11, 22), (21, 12), (21, 22)\} && \text{(W/W), (RT)}
\end{aligned}$$

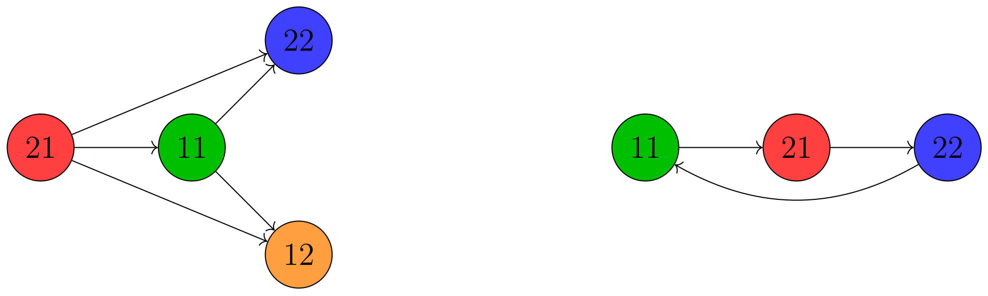


Figure 2.23: Conflict graphs for h_1 (left) and h_2 (right) of Fig. 2.22

Which overall is the conflict set:

$$\{(11, 12), (11, 22), (21, 11), (21, 12), (21, 22)\}.$$

Alternatively, its conflict graph $CG(h_1)$ is shown in Figure 2.23.

This conflict order is acyclic and g-history is conflict opaque. A serial history with the same order is $h_{1,s}$. On the other hand, the conflict order of g-history h_2 is:

$$\begin{aligned} <^{CO}_{h_2} = \{(11, 21)\} \cup \{(21, 22)\} \qquad & \text{(R/W), (W/R)} \\ \cup \{(22, 11)\} \cup \{(21, 22)\}. \qquad & \text{(W/W), (RT)} \end{aligned}$$

Which overall is the set:

$$\{(11, 21), (21, 22), (22, 11)\}.$$

It is easy to see that this order is cyclical. Its conflict graph $CG(h_2)$ is shown in Figure 2.23.

We can now define the membership and correctness problem for conflict opacity:

Problem 3 (Membership Problem for conflict opacity)**.** *Given a g-history h, is h conflict opaque?*

Problem 4 (Correctness problem for conflict opacity)**.** *Given an implementation I, are all g-histories produced by I conflict opaque?*

$$\begin{aligned}
ph_e &= \mathbf{R}^{11}_{t_1}[y]\mathbf{R}^{21}_{t_2}[x]\mathbf{W}^{21}_{t_2}[x]\mathbf{W}^{11}_{t_1}[x,y]\mathbf{R}^{12}_{t_1}[x]\mathbf{W}^{12}_{t_1}[z]\\
\overline{ph_e} &= \mathbf{T}_{\mathbf{w}}\mathbf{R}^{11}_{t_1}[y]\mathbf{R}^{21}_{t_2}[x]\mathbf{W}^{21}_{t_2}[x]\mathbf{W}^{11}_{t_1}[x,y]\mathbf{R}^{12}_{t_1}[x]\mathbf{W}^{12}_{t_1}[z]\mathbf{T}_{\mathbf{r}}\\
ph_s &= \mathbf{R}^{21}_{t_2}[x]\mathbf{W}^{21}_{t_2}[x]\mathbf{R}^{11}_{t_1}[y]\mathbf{W}^{11}_{t_1}[x,y]\mathbf{R}^{12}_{t_1}[x]\mathbf{W}^{12}_{t_1}[z]
\end{aligned}$$

Figure 2.24: Example p-histories

2.3.3 Serializability Related Definitions

Here we start by presenting p-histories (p for Papadimitriou), a subset of g-histories used by Papadimitriou for his serializability definitions. The subset contains all g-histories where transactions atomically execute their begin and read events in one atomic block and at some point later their write and commit events in one atomic block. No transactions abort in this subset. We will then define the membership and correctness problems for version of state serializability (SR^+) and strict state serializability (SSR^+), adopted to support histories with threads and transactions.

P-Histories As before, a history is a sequence of events, of which in this case there are only two types: reads and writes. We call such a history p-history and the type of event p-event. Each read and write operates on a set of variables. We again use the general sets T, Var and Tr.

Definition 7 (Set of p-events)**.** *The set of p-events is defined as*

$$PEv = \{\mathbf{W}_t[V], \mathbf{R}_t[V] \mid t \in T, V \in 2^{Var}\}.$$

This set is partitioned into read events PEv_{Rd} and write events PEv_{Wr}. In a history these events can be grouped into transactions. A transaction consists either of a single read event, if it is unfinished, or of a read event followed by a write event of the same thread, if it is finished. Then a history is a sequence of these events, which are indexed by their threads and have their set of accessed variables as parameters.

Note that all definitions for events, transactions and similar that are not explicitly mentioned here are defined analogous to the ones in the previous sections.

Definition 8 (P-History)**.** *A p-history is a sequence of p-events $pev_0 \dots pev_n$, where for all i, $0 \leq i \leq n$, $pev_i \in PEv$.*

Let $\mathcal{PH}$ be the set of all p-histories as defined above. As with the previous definition of a g-history, transaction identifiers are not mentioned as in a well-formed history there is up to isomorphism exactly one way to map transaction identifiers to events. A p-history is well-formed iff its projection on each thread is a sequence of transactions where the last one is either unfinished or finished and all others are finished. In Figure 2.24 history h_e (top) shows such a well-formed p-history. We assume each p-history to be well-formed in the remainder of this thesis.

State serializability For serializability we furthermore need to define Papadimitriou's version of equivalence of p-histories, which we call p-equivalence. To define this notion and finally state serializability, we furthermore need to define

- the reads-from relation of a p-history,
- the augmentation of a p-history
- and liveness of transactions.

As mentioned before, state serializability - as defined by Papadimitriou - uses a most-recent reads-from relation. So transaction tr_1 *reads* $v \in Var$ *from* transaction tr_2 in ph whenever there exists a write event $pev = \mathbf{W}_t^{tr_2}[V]$ and a read event $pev' = \mathbf{R}_{t'}^{tr_1}[V']$ $(t, t' \in T, V, V' \subseteq Var)$ in h and $var \in V \cap V'$ such that $pev <_{ph} pev'$ and no other p-event writing to var exists in between pev and pev'.

We will denote the reads-from relation of an arbitrary p-history ph as $ph.RF$, and it holds that $ph.RF \subseteq Tr \times Tr \times Var$. For $tr, tr' \in tr(ph)$ and $var \in Var, (tr, tr', var) \in ph.RF$ means that tr' reads var from tr in ph. If a transaction tr occurs in such an element rf i.e. it is either the first or second member of the triple, we say $tr \in rf$. In Figure 2.24 we have $(11, 12, x) \in ph_e.RF$ and $(21, 12, x) \notin ph_e.RF$. For simplicity, we call members of this relation rf-elements from now on.

To ensure that all transactions have a write they read from and all last writes are read at the end, p-histories are augmented with additional transactions. The *augmented p-history* $\overline{ph}$ for a p-history ph is the p-history where two transactions are added: tr_w at the start and tr_r at the end of the p-history. The transaction tr_w reads no variable and writes to all, while tr_r from all variables and writes to none of them. See the augmentation $\overline{ph_e}$ of history ph_e in Figure 2.24 (additional transactions in blue) for an example. We call tr_w, tr_r *augmented transactions*. All other transactions are called *non-augmented transactions*. The non-augmented transactions for a p-history ph are denoted by $tr^-(ph)$. The transactions tr_w and tr_r will only be shown in examples when needed, we will abbreviate them with $\mathbf{T_w}$ and $\mathbf{T_r}$. Despite our usual colouring marking single transactions both of them will be coloured in cyan in examples throughout this thesis. Please note that all concepts regarding concatenation, prefixes and similar do not take augmented transactions into account. For example, $\mathbf{T_w}\mathbf{R}_{t_1}^{tr_1}[x]\mathbf{T_r}$ is a prefix of $\mathbf{T_w}\mathbf{R}_{t_1}^{tr_1}[x]\mathbf{W}_{t_1}^{tr_1}[x]\mathbf{T_r}$ concerning p-histories. One can think of this as tr_w and tr_r being added to p-histories only when considering reads-from relations and liveness.

A transaction is live in a non-augmented p-history ph if it is live in its augmented version $\overline{ph}$. It is live in the augmented version iff tr_r or another live transaction reads a variable from it. In p-history ph_e transaction 21 is not live since the only variable x it writes to is never read in $\overline{ph_e}$. This term is different from the term of transaction liveness in software transactional memory.

Now we will define p-equivalence*.

Definition 9. *Two p-histories $ph, ph' \in \mathcal{PH}$ are* p-equivalent *($ph \equiv ph'$) iff*

- *they have the same set of transactions, and*
- *for any live $tr \in \overline{ph}$ and any $tr' \in \overline{ph}$, $(tr', tr, var) \in \overline{ph}.RF \Leftrightarrow (tr', tr, var) \in \overline{ph'}.RF$.*

*Note that in the original paper of Papadimitriou there is a discrepancy between formal definition of equivalence and their stated intention. Given the intention of state serializability and the definition of state serializability by Bernstein et al. [10], we chose to stick to the latter.

In the example we have $ph_e \equiv ph_s$. Similar to Bernstein et al. [10] we will assume for the remainder of the thesis that each ph is augmented.

As noted by Papadimitriou [73], it is actually required that both p-histories have the same set of *live* transactions, but w.l.o.g. this is equivalent to assuming their transactions overall are identical. A history is *serial* whenever each read event either belongs to an unfinished transaction or is directly followed by the write of its transaction. We let $\mathcal{PH}_S$ be the set of serial p-histories. In Figure 2.24 history ph_s is serial.

Definition 10 (SR^+)**.** *A p-history ph is called* serializable under SR^+ *or (state) serializable iff there exists a serial p-history ph_s such that*

- *$ph \equiv ph_s$ (p-history p-equivalence)*
- *and for all p-events $pev, pev' \in ph$ such that $t(pev) = t(pev')$: $pev <_{ph} pev' \Leftrightarrow pev <_{ph_s} pev'$ (thread order preservation).*

A serial p-history ph_s meeting the requirements of the above definition for a p-history ph is called an SR-witness of ph. We now define the membership and correctness problem for state serializability.

Problem 5 (Membership problem for state serializability)**.** *Given a p-history ph, is ph serializable under SR^+?*

Problem 6 (Correctness problem for state serializability)**.** *Given an implementation I, are all p-histories produced by I serializable under SR^+?*

If the answer to the problem question is yes, then we also say I is serializable under SR^+ or simply I is (state) serializable.

Strict state serializability State serializability can be strengthened as to also require preservation of the real-time order of transactions.

Definition 11 (SSR^+)**.** *A p-history ph is called* serializable under SSR^+ *(or* strictly (state) serializable*) iff there exists a serial p-history ph_s such that*

- *$ph \equiv ph_s$ (history equivalence)*
- *and $ph.RT \subseteq ph_s.RT$ (real-time order preservation).*

Note that real-time order is defined analogous to the definition for g-histories and it implies thread order preservation. The term of an *SSR*-witness is defined analogous to the *SR*-witness definition above. We can now define the membership and the correctness problems for both, again with the implementation being modeled as an DFA.

Problem 7 (Membership problem for strict serializability)**.** *Given a p-history ph, is ph serializable under SSR^+?*

Problem 8 (Correctness problem for strict serializability)**.** *Given an implementation I, are all p-histories produced by I serializable under SSR^+?*

3 Membership Problem

This chapter focusses on the membership problem and is divided into three parts. First, we present the related work concerned with the complexity of the membership problems of TM correctness conditions and how these different correctness conditions compare to each other in terms of what histories are correct under them. Then we present a contribution to both of these fields by (a) showing that the membership problem for value opacity is NP-complete and (b) comparing value opacity and conflict opacity under multiple assumptions, which include a set of assumptions under which both correctness conditions are equivalent. The latter contribution is based on previous work of us [56].

3.1 Related Work

We will start by presenting the related work regarding the complexity of the membership problem for the correctness conditions, and then discuss the body of work comparing them. If it is appropriate, we will give examples illustrating the differences between conditions.

Condition	Complexity of membership problem
Sequential consistency	NP-complete [40]
Linearizability	NP-complete [96, 40]
View serializability	NP-complete [97]
State serializability	NP-complete [73]
Strict state serializability	NP-complete [88]
Conflict serializability	P [73]
Causal serializability	Unknown
Snapshot isolation	NP-complete [13]
Value Opacity	NP-complete (Section 3.2)
Conflict opacity	P (Discussed in this section.)
DU opacity	Unknown
TMS 1	Unknown
TMS 2	Unknown
VWC	Unknown

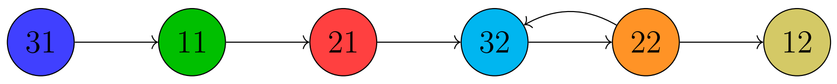
Table 3.1: Complexity of the membership problems for different correctness conditions. Results presented in this thesis in green color

3.1.1 Complexity of the Membership Problem

An overview of the published contributions to this field can be found in Table 3.1. We will present the results in the order the corresponding correctness conditions were introduced in Section 2.2. As in that section, we will divide conditions into derived conditions and conditions designed specifically for TMs.

3.1.2 Derived Correctness Conditions

The membership problems for sequential consistency [40] and linearizability [96, 40] are NP-complete. To check a history for the original definition of view serializability has been proven to be NP-complete [97]. This is because the problem can be polynomially reduced to the membership problem for state serializability, which also was proven to be NP-complete by Papadimitriou [73]. The proof by Papadimitriou involves a reduction of the non-circular SAT problem to the membership problem of state serializability. For conflict serializability the membership problem is decidable in polynomial time [73]. In practice, algorithms solving



$$\mathbf{R}_1^{11}[y]\mathbf{R}_2^{21}[]\mathbf{W}_2^{21}[y]\mathbf{R}_3^{31}[x]\mathbf{W}_3^{31}[x]\mathbf{W}_1^{11}[x]\mathbf{R}_1^{12}[]\mathbf{R}_2^{22}[x]\mathbf{R}_3^{32}[]\mathbf{W}_3^{32}[x,y]\mathbf{W}_2^{22}[x,z]\mathbf{W}_1^{12}[x]$$

Figure 3.1: Example of a conflict graph and corresponding history

this problem for specific histories employ a conflict graph (also called precedence graph). This graph models the conflicts of a history. An example of such a graph can be seen in Figure 3.1.
This graph can then be checked for cycles, for example via depth-first-search. Strict state serializability was shown to be NP-complete [88]. It was shown by Kelter that if every value in a history is being read by a live transaction, meaning it contributes to the end state, then the problem is solvable in polynomial time [54]. For causal serializability no result is known. The membership problem for the related condition causal consistency has been recently shown to be in P [13]. In the same work it was shown that checking histories for snapshot isolation is NP-complete.

Correctness conditions specifically designed for TMs To our best knowledge, there are no other works concerned with the complexity of the membership problem for these correctness conditions. Opacity is NP-complete as we show in Section 3.2. The membership problem for conflict opacity is trivially solvable in polynomial time. This is done via constructing a conflict graph (by comparing each event with one another), and then checking (for example with DFS) if this graph is acyclic. As of now, there are no complexity results known for VWC and TMS 1 and 2.

3.1.3 Relation between Different Correctness Conditions

In this section, we will present the related work with regard to how correctness conditions relate to each other, and give illustrating examples. We will order these results analogue to the order in which we presented the correctness conditions in

	SR	SSR	CSR
SR	$=$	$\supseteq$	$\subseteq$
SSR	■	$=$	$\neq$
CSR	■	■	$=$

Figure 3.2: Overview of comparison results of Papadimitriou [73].
SR= state serializability, SSR = strict state serializability, CSR = conflict serializability.

Section 2.2. For each result the condition that was introduced earlier determines its place in the order with the second condition being the tiebreaker. While we will mostly describe the related work where it is applicable, there are two sources cited for multiple conditions, which we will present beforehand.

The paper, presenting state serializability, also contained an extensive comparison of various database correctness conditions [73]. In the paper p-histories and a most-recent reads-from relation were used for all definitions. An overview of the results that are relevant to us can be found in Figure 3.2. Each cell in the figure shows the relation between the set of histories fulfilling the conditions of the correctness condition of its row to the correctness conditions of its column.

In their comprehensive overview of correctness conditions, Dziuma et al. defined correctness conditions in a uniform manner and compared them. Note that their view serializability and strict view serializability definitions are different from ours and do not use the most-recent reads-from relation but the ambiguous value-based reads-from relation. Additionally, two versions of each correctness condition were used, one for deferred update and one for eager update. We will discuss the results for the deferred update correctness conditions which we presented in Section 2.2. See Figure 3.3 for an overview of these results.

3.1.4 Derived Correctness Conditions

In the paper that presented linearizability, it is argued that sequential consistency is weaker than linearizability as it has the same requirements without the requirement to preserve the order between non overlapping operations [51]. Also, if we assume a history where each transaction consists of a single write or read, then sequential consistency is equal to view serializability with an unambiguous

	VSR	VSSR	CASR	SI	VOP	VWC
VSR	$=$	$\supseteq$	$\subseteq$	$\neq$	$\supseteq$	$\neq$
VSSR	■	$=$	$\subseteq$	$\subseteq$	$\supseteq$	$\neq$
CASR	■	■	$=$	$\neq$	$\supseteq$	$\supseteq$
SI	■	■	■	$=$	$\supseteq$	$\neq$
VOP	■	■	■	■	$=$	$\subseteq$
VWC	■	■	■	■	■	$=$

Figure 3.3: Overview of comparison results of Dziuma et al. [31].
VSR = view serializability , VSSR = strict view serializability, CASR = causal serializability, SI = snapshot isolation, VOP = value opacity , VWC = virtual world consistency

value-based read-from relation [80].

Under the same assumptions, linearizability is equal to strict view serializability [51] because the real-time order between operations of linearizability then maps to the real-time order between transactions that strict view serializability uses.

In the paper defining view serializability, it is stated that state serializability is a weaker condition than view serializability [97]. This is because in view serializability there are no dead transactions whose reads are not considered for witnesses. This can be seen in Figure 3.4. The top history is state serializable but not view serializable as the reads of dead transactions 1 and 2 do not matter for the witness. Yannakakis also claims that if each read-only transaction were to be made live by writing to a unique variable, then state serializability is equal to view serializability [97]. As shown by the top history in Figure 3.4, this does not hold. But if every *dead* (writing transactions can also be dead, see the top example) transaction were made to be live, then one can prove that the corresponding history is state serializable iff it is view serializable. The bottom example in Figure 3.4 shows this transformation. Now the reads of all transactions matter for a witness as they are all live. Thus, this history is not state serializable and not view serializable.

It is well known that conflict serializability is a stronger correctness condition than view serializability [97, 73, 9]. This is because if two writes from a transaction are not read at all in a history, their order in a witness is irrelevant for its view equivalency to the history in question, but for conflict serializability their order is

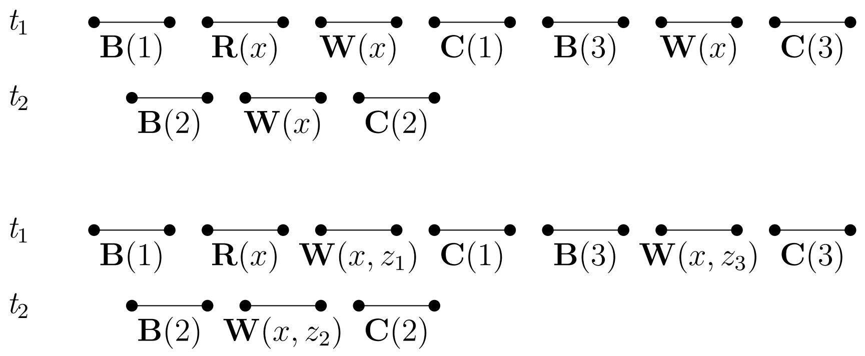

t_1
$\mathbf{B}(1)$ $\mathbf{R}(x)$ $\mathbf{W}(x)$ $\mathbf{C}(1)$ $\mathbf{B}(3)$ $\mathbf{W}(x)$ $\mathbf{C}(3)$
t_2
$\mathbf{B}(2)$ $\mathbf{W}(x)$ $\mathbf{C}(2)$
t_1
$\mathbf{B}(1)$ $\mathbf{R}(x)$ $\mathbf{W}(x, z_1)$ $\mathbf{C}(1)$ $\mathbf{B}(3)$ $\mathbf{W}(x, z_3)$ $\mathbf{C}(3)$
t_2
$\mathbf{B}(2)$ $\mathbf{W}(x, z_2)$ $\mathbf{C}(2)$

Figure 3.4: From top to bottom: state serializable but not view serializable, not state serializable and not view serializable.

part of the conflict order anyway.

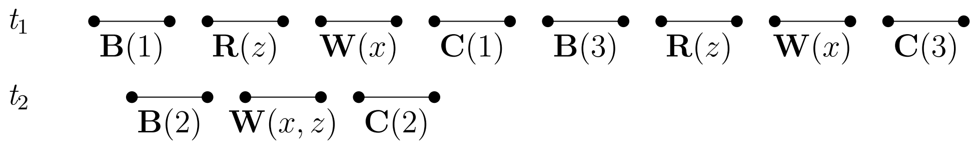

t_1
$\mathbf{B}(1)$ $\mathbf{R}(z)$ $\mathbf{W}(x)$ $\mathbf{C}(1)$ $\mathbf{B}(3)$ $\mathbf{R}(z)$ $\mathbf{W}(x)$ $\mathbf{C}(3)$
t_2
$\mathbf{B}(2)$ $\mathbf{W}(x, z)$ $\mathbf{C}(2)$

Figure 3.5: A history that is view/state serializable but not conflict serializable.

For an illustration see the history in Figure 3.5. The conflict order contains the cycle $2, 1, 2$, but the witness with transaction order $1, 2, 3$ proves the view serializability of the history.

Dziuma et al. stated that view serializability is a stronger condition than causal serializability. This was also stated in the paper presenting the original definition of causal serializability [31, 80]. It is easy to see that if there is a witness where each transaction is legal, then for each thread there exists a witness where the reads of its transactions are legal. For an example of a history that is causal serializable but not view serializable, consider Figure 3.6.
This history is trivially not view serializable as in any transaction order of a witness one of the reading transactions would only be legal if it reads the value 1 for both variables. It is causally serializable as in the witness with transaction order $1, 3, 2, 4$ each read of thread 1 is legal and in the witness with transaction order $2, 4, 1, 3$ each read of thread 2 is legal.

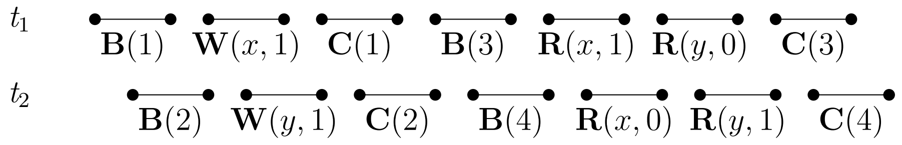


Figure 3.6: A history that is causal serializable but not view serializable.

It is well known that snapshot isolation does not imply (view) serializability, this fact was already discussed in the paper defining snapshot isolation [8]. Additionally, efforts were made to define conditions under which snapshot isolation does imply serializability [38, 84, 32]. View serializability is incomparable to snapshot isolation under ambiguous value-based reads-from relations [31]. The same holds under the most-recent reads-from relation [70].

The most-recent reads-from relation version of view serializability is not comparable to the original definition of opacity [46]. But given an ambiguous value-based reads-from relation, Dziuma et al. showed that view serializability is weaker than value opacity [31]. The proof involved showing that strict view serializability is weaker than value opacity, which we will discuss later. Basically, if all transactions are committed, view serializability is opacity without a real-time order.

Also, view serializability with an ambiguous value-based reads-from relation is weaker than virtual world consistency [31]. This is based on the fact that the additional requirements virtual world consistency has for aborted transactions are not relevant for view serializability. It does not consider aborted transactions in its definition, and the other requirements are identical to VWC.

Papadimitriou compared state serializability to conflict serializability and strict state serializability, as noted before using a most-recent reads-from relation and p-histories [73]. He showed that state serializability is a weaker condition than conflict serializability. The argument is analogue to the one presented for view serializability and conflict serializability. For an example of a history that is state serializable but not conflict serializable, see Figure 3.5. He also showed that if the write set of each transaction is a subset of its read set, then state serializability and conflict serializability are identical, later we call this the *read-before-update assumption*. State serializability is a weaker condition than strict state serializability as it has the same requirements plus the extra requirement that each witness must preserve the real-time order of its history [73]. Any strictly state serializable history is also state serializable as all its requirements for the history are fulfilled by it being strictly state serializable.

A variant of state serializability with histories closer to g-histories was shown to be incomparable to snapshot isolation, and the same result holds for conflict serializability and snapshot isolation [70].

Snapshot isolation in the survey of Dziuma et al. has been proven to be weaker than value opacity as snapshot isolation is weaker than view serializability which in turn a weaker condition than value opacity [31]. It is incomparable to virtual world consistency and causal serializability [31].

Causal serializability in the survey of Dziuma et al. has been proven to be weaker than value opacity as causal serializability is weaker than view serializability which in turn is weaker than value opacity [31].

3.1.5 Correctness Conditions Specifically Designed for TMs

In a result published by us the relationship between value opacity and conflict opacity was discussed [56]. We present this result and expand on it in Section 3.3.

The authors of the paper presenting DU-opacity discussed its differences to a prefix-closed version of value opacity. This version is explicitly defined to be prefix-closed by requiring each prefix of a history to be value opaque [5]. For an ambiguous value-based reads-from relation DU-opacity is a stronger condition than this version of value opacity. The intuition that DU-opacity was meant to be a prefix-closed version of value opacity, and thus both definitions should be identical, does not hold. The issue lies in the fact that even in this explicit prefix-closed value opacity a read may be justified by different transactions in different prefixes of the same history. Figure 3.7 illustrates the issue.
Each prefix of the history is value opaque because until the abort of transaction 1 the read of transaction 2 can be legal in a witness as a commit pending transaction can be completed as committed. When transaction 1 aborts, transaction 3 can justify the read of transaction 2 in a witness as it is commit pending with a write

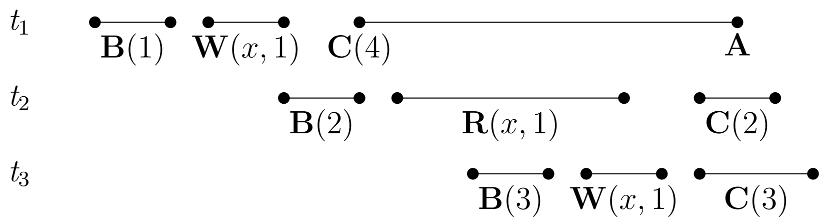


Figure 3.7: A history that is prefix-closed value opaque but not DU-opaque.

of value 1 on x. This history is not DU-opaque since transaction 3 is not commit pending or committed during the read of transaction 2.

The authors of the paper presenting VWC claim it is weaker than value opacity [53]. While intuitively true, they do not provide a formal definition of value opacity and their informal definition does not consider live transactions at all. Additionally, it is unclear which reads-from relation is used. In an example showing that there are non value opaque histories which are virtual world consistent, they implicitly employ a most-recent reads-from relation for value opacity and VWC. However, the original definition of value opacity uses an ambiguous value-based reads-from relation, and for VWC an unambiguous value-based reads-from relation seems to be used. So it is not exactly clear which definitions are compared. In the survey of Dziuma et al. the definitions of value opacity and VWC both use an ambiguous value-based reads-from relation, and both definitions consider live transactions. It presents the same conclusion [31]. Any value opaque history is also virtual world consistent as the overall witness implies that there exists a witness containing all committed transactions and one witness for each aborted transaction. An example history which is virtual world consistent but not value opaque can be seen in Figure 3.8.
Overall, there is no serialization that makes both reads legal, but there is a witness containing both committed transactions and a witness for each reading transaction as each reading transaction has only the transaction it reads from in its causal past.

In the paper presenting TMS 1 and 2 Doherty et al. stated that they believe but have not proven that TMS 1 implies the explicitly prefix-closed version of value

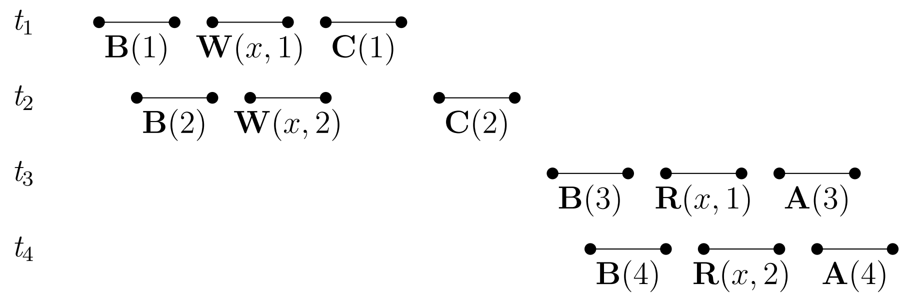


Figure 3.8: A history that is virtual world consistent but not value opaque.

opacity [29]. They give an example history that is correct under TMS 1, but not value opaque. This history is shown in Figure 3.9.

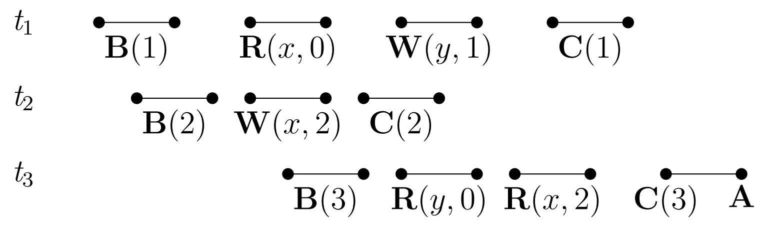


Figure 3.9: A history that is correct under TMS1 but not value opaque.

It is not value opaque as for a potential witness transactions 1 and 2 must be ordered as $1, 2$, and additionally 3 must be ordered after transaction 2. Then the only potential transaction order is $1, 2, 3$ which is not a legal history. Thus, no witness exists. Under TMS 1 this execution is correct as if transaction 1 would have aborted, transaction 3 would be legal. At a later time, Lesani et al. showed that a prefix-closed version of value opacity modelled as an automaton implies TMS1 [62].

Value opacity is implied by TMS2 [61], but value opacity does not imply TMS 2. Doherty et al. showed the latter via an example history [29] which can be seen in Figure 3.10.

At the commit of transaction 2, its read no longer matches the current memory state; thus, this execution is not correct under TMS 1, but the witness with transaction order $2, 1$ proves its value opacity.

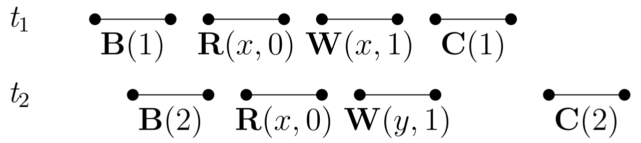


Figure 3.10: A history that is value opaque but not correct under TMS2.

TMS 1 has been shown to be weaker than TMS 2 via a mechanical proof [29]. Additionally, it is incomparable to VWC. The witnesses for aborted transactions in VWC do not need to preserve the real-time order of transactions, and thus they can ignore committed transactions real-time ordered before it, which TMS 1 does not allow. TMS 1, on the other hand, allows transactions to read from other commit invoked transactions which VWC does not.

3.2 The Membership Problem for Value Opacity is NP-Complete

We prove the NP-completeness of the membership problem for opacity by reducing the membership problem for SR (see Section 2.3.3) to it and showing that given a g-history it can be determined in polynomial time if it is an OP-witness for another g-history. The first step proves the problem to be NP-hard and the second step proves it to be a member of NP, showing that it is NP-complete overall. We will first discuss the reduction function in length. The second step is then briefly discussed in the description of Lemma 5 at the end of this section as it is.

As proven by Papadimtriou [72, 73], the membership problem is NP-complete even if no transactions are dead. A history of SR can trivially be reduced to an instance of the membership problem of SR^+ where each thread contains exactly one transaction. Then there is no internal thread order imposed on transactions, which is the only meaningful difference to SR. Also, the solution of an instance of a membership problem for SR^+ can be checked in polynomial time as p-equivalence can be checked in polynomial time [73] and thread order preservation trivially can as well. To avoid unnecessary introduction of new notation, we will reduce to the SR^+ problem under the above assumption.

Our reduction is based on a function f that maps a p-history ph to a g-history

$$\begin{aligned}
ph^{SR} &= \mathbf{T_w}\mathbf{R}^{11}_{t_1}[]\mathbf{R}^{21}_{t_2}[]\mathbf{W}^{11}_{t_1}[var, var'']\mathbf{W}^{21}_{t_2}[var]\mathbf{R}^{31}_{t_3}[var]\mathbf{W}^{31}_{t_3}[var]\mathbf{R}^{41}_{t_4}[var']\mathbf{W}^{41}_{t_4}[var']\mathbf{T_r}\\
ph^{SR}_{s1} &= \mathbf{T_w}\mathbf{R}^{11}_{t_1}[]\mathbf{W}^{11}_{t_1}[var, var'']\mathbf{R}^{21}_{t_2}[]\mathbf{W}^{21}_{t_2}[var]\mathbf{R}^{31}_{t_3}[var]\mathbf{W}^{31}_{t_3}[var]\mathbf{R}^{41}_{t_4}[var']\mathbf{W}^{41}_{t_4}[var']\mathbf{T_r}\\
ph^{SR}_{s2} &= \mathbf{T_w}\mathbf{R}^{41}_{t_4}[var']\mathbf{W}^{41}_{t_4}[var']\mathbf{R}^{11}_{t_1}[]\mathbf{W}^{11}_{t_1}[var, var'']\mathbf{R}^{21}_{t_2}[]\mathbf{W}^{21}_{t_2}[var]\mathbf{R}^{31}_{t_3}[var]\mathbf{W}^{31}_{t_3}[var]\mathbf{T_r}\\
f(ph^{SR}) &= \mathbf{T_w}\mathbf{B}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{B}^{31}_{t_3}\mathbf{B}^{41}_{t_4}\mathbf{W}^{11}_{t_1}(var, 11)\mathbf{W}^{11}_{t_1}(var'', 11)\mathbf{W}^{21}_{t_2}(var, 21)\\
&\quad \mathbf{R}^{31}_{t_3}(var, 21)\mathbf{W}^{31}_{t_3}(var, 31)\mathbf{R}^{41}_{t_4}(var', tr_w)\mathbf{W}^{41}_{t_4}(var', 41)\mathbf{C}^{11}_{t_1}\mathbf{C}^{21}_{t_2}\mathbf{C}^{31}_{t_3}\mathbf{C}^{41}_{t_4}\mathbf{T_r}\\
f(ph^{SR}_{s2}) &= \mathbf{T_w}\mathbf{B}^{41}_{t_4}\mathbf{R}^{41}_{t_4}(var', tr_w)\mathbf{W}^{41}_{t_4}(var', 41)\mathbf{C}^{41}_{t_4}\mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(var, 11)\mathbf{W}^{11}_{t_1}(var'', 11)\mathbf{C}^{11}_{t_1}\\
&\quad \mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(var, 21)\mathbf{C}^{21}_{t_2}\mathbf{B}^{31}_{t_3}\mathbf{R}^{31}_{t_3}(var, 21)\mathbf{W}^{31}_{t_3}(var, 31)\mathbf{C}^{31}_{t_3}\mathbf{T_r}
\end{aligned}$$

Figure 3.11: Examples illustrating the reduction function

$f(ph)$ such that $f(ph)$ is opaque under OP iff ph is serializable under SR. Both problems are about deciding whether for a given history a serial history (witness) exists fulfilling certain constraints. A serial history can be viewed as a total order on the transactions of the original history. Whether it is a witness or not is then determined by if this total order adheres to the constraints. In both problems the constraints are solely based on the original history. In the remainder of this section, we call the constraints SR-constraints or OP-constraints depending on the problem. The reduction function tries to carry over the constraints for ph to $f(ph)$ by ensuring three properties:

1. There is a one-to-one mapping of transactions between ph and $f(ph)$,
2. the OP-constraints on the transactions of $f(ph)$ must contain the SR constraints on the corresponding transactions of ph (preservation of SR constraints)
3. and no other OP-constrains exists in $f(ph)$ (preventing additional OP constraints).

In the further, we will describe how f realizes each of the above properties. This will be explained in an incremental manner where each step provides the input for the next. As f is a function, its definition will execute these steps as one, not being incremental but yielding the same result. For better clarity, ph^{SR} from Figure 3.11 will be used as an example. Note that $\mathbf{T_w}$ and $\mathbf{T_r}$ share the same colour for both being augmented transactions despite not being one transaction.

One-to-One mapping of transactions The first goal is achieved by a syntactical replacement of symbols, which keeps the write and read sets for each transaction identical. This is done by the following replacements:

$$\mathbf{R}_t^{tr}[var_0, \ldots, var_n] \rightarrow \mathbf{B}_t^{tr}\mathbf{R}_t^{tr}(var_0, ?) \ldots \mathbf{R}_t^{tr}(var_n, ?)$$

$$\mathbf{W}_t^{tr}[var_0, \ldots, var_n] \rightarrow \mathbf{W}_t^{tr}(var_0, ?) \ldots \mathbf{W}_t^{tr}(var_n, ?)\mathbf{C}_t^{tr}$$

The resulting sequences are not interleaved with other transactions keeping the atomicity of each read and write event in a p-history. The write set and read set is kept identical for each transaction. Additionally, a begin or, respectively, commit is added to conform to the transaction standards of a g-history. Note that instead of values we used a question mark as the replacement of values will be explained in the next step. The replacement of tr_w and tr_r yields a transaction writing to all variables and a transaction reading from all variables, respectively. Their notation ($\mathbf{T_w}, \mathbf{T_r}$) in the examples remains identical even after the change to avoid obfuscating our notation. For our example, this syntactical replacement yields the following result:

$$\begin{aligned}
&\mathbf{T_w}\mathbf{R}_{t_1}^{11}[]\mathbf{R}_{t_2}^{21}[]\mathbf{W}_{t_1}^{11}[var, var'']\mathbf{W}_{t_2}^{21}[var]\\
&\mathbf{R}_{t_3}^{31}[var]\mathbf{W}_{t_3}^{31}[var]\mathbf{R}_{t_4}^{41}[var']\mathbf{W}_{t_4}^{41}[var']\mathbf{T_r}\\
&\rightarrow\\
&\mathbf{T_w}\mathbf{B}_{t_1}^{11}\mathbf{B}_{t_2}^{21}\mathbf{W}_{t_1}^{11}(var, ?)\mathbf{W}_{t_1}^{11}(var'', ?)\mathbf{C}_{t_1}^{11}\mathbf{W}_{t_2}^{21}(var, ?)\mathbf{C}_{t_2}^{21}\\
&\mathbf{B}_{t_3}^{31}\mathbf{R}_{t_3}^{31}(var, ?)\mathbf{W}_{t_3}^{31}(var, ?)\mathbf{C}_{t_3}^{31}\mathbf{B}_{t_4}^{41}\mathbf{R}_{t_4}^{41}(var', ?)\mathbf{W}_{t_4}^{41}(var', ?)\mathbf{C}_{t_4}^{41}\mathbf{T_r}.
\end{aligned}$$

The colors represent the different transactions. As one can see, each transaction still consists of two atomic parts and has the same read and write sets (only considering variables) under both definitions.

Preservation of *SR* constraints Now given that syntactical conversion, we explain how the constraints for *SR*-witnesses can be preserved. The *SR* definition imposes two sets of constraints:

- constraints regarding the augmented transactions
- and the reads-from constraints.

The first set of constraints requires tr_w to happen before all other transactions and tr_r to happen after all other transactions. Given the previous step, these constraints are already preserved as in the resulting g-history tr_w is real-time ordered before all other transactions and tr_r after all other transactions.

The second set of constraints requires that the reads-from relation is identical for the (augmentations) of a p-history and its SR-witnesses. In our example p-history ph^{SR}, transaction 31 reads var from transaction 21. Let ph_s be an arbitrary SR-witness of ph^{SR}. As a reads-from relation between two transactions is only given when the writing transaction's write is the last write before the reading transaction's read, this imposes two constraints on the SR-witness:

1. a **before constraint**, the writing transaction must come before the reading transaction, for the example this is $21 \prec_{ph_s} 31$
2. and a **not-in-between constraint**, any transaction that would interrupt the read cannot be ordered in between, for the example this is $\neg(21 \prec_{ph_s} 11 \prec_{ph_s} 31)$.

For the OP-problem a reads-from relation does not exist, but in an OP-witness each transaction must be legal. This means that for any read on a variable the value that was read must be the last written value to that variable at the point of the read. One may think this is equivalent to the reads-from relation as a read reads the same value in a g-history and its OP-witness, and thus in both also reads from the same transaction writing that value. But there is a difference: In a g-history multiple transactions can write the same value. If that value is read; in the absence of other constraints; there is no clearly defined writer to the read. Any of these writing transactions may be the last writer before the read in the serial g-history.

$$h^{op} = \mathbf{B}_{t_1}^{11}\mathbf{W}_{t_1}^{11}(var, val)\mathbf{B}_{t_2}^{21}\mathbf{W}_{t_2}^{21}(var, val)\mathbf{C}_{t_2}^{21}\mathbf{C}_{t_1}^{11}\mathbf{B}_{t_1}^{12}\mathbf{R}_{t_1}^{12}(var, val)\mathbf{A}_{t_1}^{12}$$
$$h_{s1}^{op} = \mathbf{B}_{t_1}^{11}\mathbf{W}_{t_1}^{11}(var, val)\mathbf{C}_{t_1}^{11}\mathbf{B}_{t_2}^{21}\mathbf{W}_{t_2}^{21}(var, val)\mathbf{C}_{t_2}^{21}\mathbf{B}_{t_1}^{12}\mathbf{R}_{t_1}^{12}(var, val)\mathbf{A}_{t_1}^{12}$$
$$h_{s2}^{op} = \mathbf{B}_{t_2}^{21}\mathbf{W}_{t_2}^{21}(var, val)\mathbf{C}_{t_2}^{21}\mathbf{B}_{t_1}^{11}\mathbf{W}_{t_1}^{11}(var, val)\mathbf{C}_{t_1}^{11}\mathbf{B}_{t_1}^{12}\mathbf{R}_{t_1}^{12}(var, val)\mathbf{A}_{t_1}^{12}$$

Figure 3.12: Example showing reads-from ambiguity in OP

The g-history h^{OP} in Figure 3.12 with its OP-witnesses h_{s1}^{op} and h_{s2}^{op} illustrate both properties. It is not clear whether transaction 12 reads from 11 or 21. Thus, in h_{s1}^{op}, 21 is read by 12, and in h_{s2}^{op} 11 is being read by 12.

This ambiguity is avoided by f by making each value written to a variable unique with regard to other values written to that variable. For writes we chose the transaction ID of the writer as the written value as threads have at most one transaction in SR and write at most once to a variable. Then, for a read on a variable, the read value is set to the transaction ID of the transaction it reads that variable from in the original p-history. As we have a transaction at the beginning initializing all variables, this is well-defined. For the augmented transactions we again keep the $\mathbf{T_w}$ and $\mathbf{T_r}$ symbols after the modifications. Still, all changes discussed above apply to these transactions. We show how the values are inserted in our example, continuing with the result of the first step:

$$\mathbf{T_w}\mathbf{B}_{t_1}^{11}\mathbf{B}_{t_2}^{21}\mathbf{W}_{t_1}^{11}(var, ?)\mathbf{W}_{t_1}^{11}(var'', ?)\mathbf{C}_{t_1}^{11}\mathbf{W}_{t_2}^{21}(var, ?)\mathbf{C}_{t_2}^{21}$$
$$\mathbf{B}_{t_3}^{31}\mathbf{R}_{t_3}^{31}(var, ?)\mathbf{W}_{t_3}^{31}(var, ?)\mathbf{C}_{t_3}^{31}\mathbf{B}_{t_4}^{41}\mathbf{R}_{t_4}^{41}(var', ?)\mathbf{W}_{t_4}^{41}(var', ?)\mathbf{C}_{t_4}^{41}\mathbf{T_r}$$
$$\rightarrow$$
$$\mathbf{T_w}\mathbf{B}_{t_1}^{11}\mathbf{B}_{t_2}^{21}\mathbf{W}_{t_1}^{11}(var, 11)\mathbf{W}_{t_1}^{11}(var'', 11)\mathbf{C}_{t_1}^{11}\mathbf{W}_{t_2}^{21}(var, 21)\mathbf{C}_{t_2}^{21}$$
$$\mathbf{B}_{t_3}^{31}\mathbf{R}_{t_3}^{31}(var, 21)\mathbf{W}_{t_3}^{31}(var, 31)\mathbf{C}_{t_3}^{31}\mathbf{B}_{t_4}^{41}\mathbf{R}_{t_4}^{41}(var', tr_w)\mathbf{W}_{t_4}^{41}(var', 41)\mathbf{C}_{t_4}^{41}\mathbf{T_r}.$$

Under this construction each transaction in an OP-witness is legal iff it reads each variable from the same transaction as in the original p-history. This preserves the constraints of SR.

Preventing additional OP-constraints Next, we address how to prevent additional OP-constraints to occur in the generated g-history. Opacity imposes a real-time order constraint on the OP-witnesses of a g-history, meaning if transactions have no overlap in a g-history then its OP-witnesses must preserve the order of these transactions. We discussed in the previous step that the real-time order ensures the constraints regarding tr_w and tr_r. On the other hand, for every other transaction SR-constraints do not include anything resembling a real-time order. Histories h^{op} and ph^{SR} in the example illustrate this issue. In h^{op} transactions 11 and 12 have no overlap and are thus real-time ordered. Therefore, in the OP-witnesses h^{op}_{s1} and h^{op}_{s2} they keep their order. In contrast, in ph^{SR} transaction 41 is "real-time ordered" (if that concept would exist in SR) after any other transaction in that p-history. But in the example SR-witness ph^{SR}_{s2}, transaction 41 is ordered before any other transaction.

Thus, for the real-time order to impose no additional constraints on the g-history, each non-augmented transaction must be concurrent. The augmented transactions are kept at the start or at the end of the history, respectively. For the non-augmented transactions the begin event is put directly after tr_w and the commit event (aborts do not exist in g-histories generated by the reduction function) right before tr_r. We show this along our example:

$$\mathbf{T_w}\mathbf{B}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{11}_{t_1}(var, 11)\mathbf{W}^{11}_{t_1}(var'', 11)\mathbf{C}^{11}_{t_1}\mathbf{W}^{21}_{t_2}(var, 21)\mathbf{C}^{21}_{t_2}$$
$$\mathbf{B}^{31}_{t_3}\mathbf{R}^{31}_{t_3}(var, 21)\mathbf{W}^{31}_{t_3}(var, 31)\mathbf{C}^{31}_{t_3}\mathbf{B}^{41}_{t_4}\mathbf{R}^{41}_{t_4}(var', tr_w)\mathbf{W}^{41}_{t_4}(var', 41)\mathbf{C}^{41}_{t_4}\mathbf{T_r}$$
$$\rightarrow$$
$$\mathbf{T_w}$$
$$\mathbf{B}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{B}^{31}_{t_3}\mathbf{B}^{41}_{t_4}$$
$$\mathbf{W}^{11}_{t_1}(var, 11)\mathbf{W}^{11}_{t_1}(var'', 11)\mathbf{W}^{21}_{t_2}(var, 21)$$
$$\mathbf{R}^{31}_{t_3}(var, 21)\mathbf{W}^{31}_{t_3}(var, 31)\mathbf{R}^{41}_{t_4}(var', tr_w)\mathbf{W}^{41}_{t_4}(var', 41)$$
$$\mathbf{C}^{11}_{t_1}\mathbf{C}^{21}_{t_2}\mathbf{C}^{31}_{t_3}\mathbf{C}^{41}_{t_4}$$
$$\mathbf{T_r}.$$

Overall the reduction function thus preserves the constraints of SR while avoiding additional constraints imposed by OP. This means if a OP-witness exists, that same order of transactions also is a SR-witness for the original. For our example the transaction order for the OP-witness $f(ph^{SR}_{s2})$ is also the transaction order of the SR-witness ph^{SR}_{s2}.

Defining f We are now ready to formally define f. As noted previously, it does not follow the incremental structure of the description above, but the result is identical. It consists of two parts:

- It places all transactional events (begin and commits). At the beginning of the g-history the transactional events of tr_w are placed, at the end the ones of tr_r and in between them first all other begins and then all other commits.
- For each p-event in the input p-history it inserts the corresponding non-transactional events between the begin and commit of its transaction.

These steps relate as follows to the properties f needs to achieve: The first step prevents additional constraints and preserves the constraints related to the augmented transactions, the second preserves the reads-from constraints and both steps together establish the one-to-one mapping of transactions. While the first step is a simple matter of definition, the second step is realized via a subfunction r.

The subfunction is parametrized by a p-history ph and takes a p-event pev as input and returns a sequence of events.

- If pev is a write to a variable set, r returns a sequence of writes; one to each variable in the set; writing $tr_{ph}(pev)$ as its value. We illustrate this along our example:

$$r(ph^{SR}, \mathbf{W}^{11}_{t_1}[var, var']) = \mathbf{W}^{11}_{t_1}(var, 11)\mathbf{W}^{11}_{t_1}(var'', 11).$$

- If pev is a read from transaction tr on a variable set, r returns a sequence of reads ; one from each variable var in the set. The value read by each read depends on the reads-from relation of ph. It is the transaction ID of the transaction that transaction tr read var from in the augmentation of ph. Using the augmentation serves the purpose that transactions reading from the initial state also have a defined value. We illustrate this along our example:

$$r(ph^{SR}, \mathbf{R}^{31}_{t_3}[var]) = \mathbf{R}^{31}_{t_3}(var, 21).$$

Let $rf_{ph}(tr_1, var)$ return the transaction tr_2 s.t. $(tr_2, tr_1, var) \in ph.RF$. Then the definition of r is as follows.

Definition 12 (Subfunction r)**.** *Let ph be a p-history and pev a p-event in ph. Let Var' be the set of variables pev writes to/reads from. Let tr be the transaction of pev and t be its thread. Then r is defined as follows:*

$$r_{ph}(pev) = \begin{cases} \bullet_{var \in Var'} \mathbf{W}^{tr}_{t}(var, tr), & \text{if } pev \in PEv_{Wr} \\ \bullet_{var \in Var'} \mathbf{R}^{tr}_{t}(var, rf_{ph}(tr, var)), & \text{if } pev \in PEv_{Rd}. \end{cases}$$

Now we can define f. As discussed, f starts with the converted tr_w, then places a begin for each non-augmented transaction, inserts the event sequences determined by r afterwards, places a commit for each non-augmented transaction and then ends with the converted tr_r. Note that the first and last event of a history are an empty read and an empty write, respectively, belonging to tr_w and tr_r. As the application of r to them would result in no events, they are discarded when computing f. Let $Var(ph)$ be the set of all variables occuring in ph.

Definition 13 (Reduction function f)**.** *Given a p-history $ph = pev_0 \ldots pev_n$, the reduction function f is defined as follows:*

$$\begin{array}{lccccccl} f(ph) = & & \mathbf{B}^{tr_w}_{thr(tr_w)} & \cdot & r_{ph}(pev_1) & \cdot & \mathbf{C}^{tr_w}_{thr(tr_w)} & \mid (tr_w) \\ & \cdot & \left(\bullet_{tr \in tr^-(ph)} \mathbf{B}^{tr}_{thr(tr)}\right) & \cdot & \bullet_{2 \leq k \leq n-2} r_{ph}(pev_k) & \cdot & \left(\bullet_{tr \in tr^-(ph)} \mathbf{C}^{tr}_{thr(tr)}\right) & \mid (\text{Non-aug. T.}) \\ & \cdot & \mathbf{B}^{tr_r}_{thr(tr_r)} & \cdot & r_{ph}(pev_{n-1}) & \cdot & \mathbf{C}^{tr_r}_{thr(tr_r)} & \mid (tr_r.) \end{array}$$

We will now use this reduction function to prove the NP-completeness of the membership problem for OP.

Proving NP-completeness The proof of the following lemmas and the concluding Theorem 1 can be found in Appendix A. We will summarize the proof structure here and present the lemmas used in the proof resulting in Theorem 1. Overall the proof consists of two steps:

- Show that f is a polynomial reduction function from the membership problem of *SR* to the membership problem of *OP*.
- Show that is possible to determine in polynomial time whether one g-history is an *OP*-witness of another g-history.

Lemmas 1 to 4 are concerned with the first point, Lemma 1 stating the claim and lemmas 2 to 4 being building blocks to its proof. Lemma 5 is concerned with the second point.

Lemma 1. *Given an arbitrary p-history ph, it holds that*

1.

$$f(ph) \textit{ is opaque under } OP$$
$$\leftrightarrow$$
$$ph \textit{ is serializable under } SR$$

2. and f is computable in polynomial time.

The second statement will only be discussed in the appendix as its correctness is straightforward to see. The proof of the first part of the lemma is based on three sub-lemmas, which reflect the design goals of the reduction function discussed at the beginning of this section. First off, there must be a one-to-one mapping between transactions, meaning the transaction sets of both histories are identical.

Lemma 2 (One-to-One mapping of transactions)**.** *For an arbitrary p-history ph, it holds that*

$$tr(ph) = tr(f(ph)) \textit{ (transaction sets identical)} .$$

The correctness of the lemma follows directly from the definition of f.

Secondly, the constraints from the original p-history must be preserved. This includes the reads-from constraints and the constraints for the augmented transactions. For the reads-from relations we require that each write and read in the g-history has a corresponding p-event in the input p-history. This is not a one-to-one relation as one p-event possibly spawns multiple events in the output g-history. Each write must write to its transaction's ID, and each read must read the transaction ID of the last writer on its variable in the input p-history. Also, no other event may exist. For the constraints involving the augmented transactions, we require that after applying f each non-augmented transaction of the original p-history is ordered after tr_w and before tr_r. In the lemma, these requirements are split into the requirements for writes, the requirements for reads and the requirements for the augmented transactions

Lemma 3 (Preservation of *SR*-constraints)**.** *For an arbitrary p-history ph, it holds that*

$$\mathbf{W}^{tr_1}_{thr(tr_1)}(var, val) \in f(ph) \rightarrow val = tr_1 \wedge var \in WS^{vo}_{ph}(tr_1)$$
$$\mathbf{R}^{tr_1}_{thr(tr_1)}(var, val) \in f(ph) \rightarrow \exists tr_2 \in Tr : (tr_2, tr_1, var) \in ph.RF \wedge val = tr_2$$
$$\forall tr \in tr^{-}(ph) : tr_w \prec_{f(ph)} tr \wedge tr \prec_{f(ph)} tr_r.$$

The correctness of this lemma mainly follows from the definition of r.

Lastly f should not introduce new constraints that are not present in the input p-history. This boils down to the real-time order not introducing new constraints as there are no other constraints that can be introduced. That means no non-augmented transactions should be real-time ordered.

Lemma 4 (Preventing additional *OP*-constraints)**.** *For an arbitrary p-history ph, it holds that*

$$\forall tr, tr' \in tr^{-}(ph) : \neg(tr' \prec_{f(ph)} tr \vee tr \prec_{f(ph)} tr').$$

By definition of f, all transactions of its output g-history besides tr_w and tr_r are concurrent; thus they are not real-time ordered with regard to each other.

Combining lemmas 2 to 4, we can deduce that any *OP*-witness of the g-history must fulfil the same constraints with regard to transaction order as an *SR*-witness of the input p-history. Thus, it can only exist iff an *SR*-witness for the input exists proving Lemma 1 correct. While this proves the NP-hardness of the membership problem for *OP*, we need to prove that *OP* is in NP as well. We show this by proving that we can given a g-history and a witness (in the complexity sense, which in this case is also the witness in the value opacity sense) verify in polynomial time whether the g-history is opaque under *OP*. This is sufficient to show that *OP* is in NP as an NTM can guess the correct witness (it has the same length as the g-history) if it exists, and then verify it in polynomial time.

Lemma 5. *Given two g-histories h and h_s, it is determinable in polynomial time whether h_s is an OP-witness of h or not.*

This proof is trivial as it is doable in polynomial time one needs to check for each read whether the last committed write on it has a matching value and compare the real-time orders of the g-histories. Combining Lemma 1 and Lemma 5, proves the NP-completeness of the membership problem for *OP*.

Theorem 1 (Complexity of the membership problem for *OP*)**.** *The membership problem for OP is NP-complete.*

3.3 Comparison of Conflict Opacity and Value Opacity

In this section, we will be comparing the g-histories accepted by the membership problem for *CO* and for *OP* and give assumptions under which *OP* is equivalent to *CO*. This is based and expands on previous work of us [56]. In general, both value opacity and conflict opacity are incomparable because of very specific cases that are not necessarily relevant in actual TMs. We try to obtain more meaningful results by making assumptions over the set of g-histories to exclude

$$\begin{aligned}
h_1 &= \mathbf{B}_3^{31}\mathbf{W}_3^{31}(x,1)\mathbf{W}_3^{31}(y,1)\mathbf{C}_3^{31}\mathbf{B}_1^{11}\mathbf{W}_1^{11}(x,2)\mathbf{W}_1^{11}(y,2)\mathbf{B}_2^{21}\mathbf{R}_2^{21}(x,1)\mathbf{Inv}_1^{11}(\mathbf{C})\mathbf{R}_2^{21}(y,2)\mathbf{C}_2^{21}\\
h_2 &= \mathbf{B}_1^{11}\mathbf{W}_1^{11}(x,1)\mathbf{C}_1^{11}\mathbf{B}_2^{21}\mathbf{R}_2^{21}(x,2)\mathbf{C}_2^{21}\\
h_3 &= \mathbf{B}_1^{11}\mathbf{B}_2^{21}\mathbf{W}_1^{11}(x,1)\mathbf{W}_2^{21}(x,1)\mathbf{B}_3^{31}\mathbf{W}_1^{11}(y,2)\mathbf{W}_2^{21}(y,3)\mathbf{C}_1^{11}\mathbf{R}_3^{31}(x,1)\mathbf{C}_2^{21}\mathbf{R}_3^{31}(y,3)\mathbf{C}_3^{31}\\
h_{3,s} &= \mathbf{B}_1^{11}\mathbf{W}_1^{11}(x,1)\mathbf{W}_1^{11}(y,2)\mathbf{C}_1^{11}\mathbf{B}_2^{21}\mathbf{W}_2^{21}(x,1)\mathbf{W}_2^{21}(y,3)\mathbf{C}_2^{21}\mathbf{B}_3^{31}\mathbf{R}_3^{31}(x,1)\mathbf{R}_3^{31}(y,3)\mathbf{C}_3^{31}
\end{aligned}$$

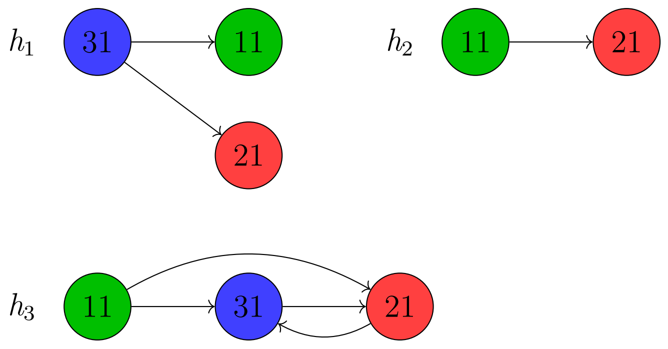


Figure 3.13: G-Histories showing that CO does not imply OP (h_1, h_2) and OP does not imply CO (h_3 with its OP-witness $h_{3,s}$) and their respective conflict graphs.

the cases where the issues occur. We will first compare the conditions under no assumptions and then under two increasingly stricter sets of assumptions. The first set of assumptions establishes a most-recent reads-from relation for value opacity, and the second set is a restriction on write and read sets. This restriction is that all transactions must read each variable they write to. Under the first set of assumptions, we show that CO implies OP but not vice versa and under the second we show that OP is equal to CO. The latter is a new result that is not contained in our previous work.

No assumptions Without any assumptions value opacity and conflict opacity have an intersection, but they are not subsets of each other. We will illustrate this relation by giving an example g-history (shown in Figure 3.13) for each type of g-history that is correct under only one of the conditions.

One can easily see CO does not imply OP because of two issues. For one, CO does not consider commit invokes to be a conflicting event, but OP allows for commit invoked transactions to be considered committed and thus their writes to be readable. OP, on the other hand, allows transactions to read the values of commit invoked transactions.

$$\mathbf{B}_3^{31}\mathbf{W}_3^{31}(x,1)\mathbf{W}_3^{31}(y,1)\mathbf{C}_3^{31}\mathbf{B}_1^{11}\mathbf{W}_1^{11}(x,2)\mathbf{W}_1^{11}(y,2)\mathbf{B}_2^{21}\mathbf{R}_2^{21}(x,1)\mathbf{C}_1^{11}\mathbf{R}_2^{21}(y,1)\mathbf{C}_2^{21}$$

Figure 3.14: Realistic g-history example for values deciding value opacity of a history

An example of this is h_1 which is obviously not value opaque, but it would be conflict opaque as the invoke does not cause any conflicts.

Second, CO does not factor in values as can be seen in its definition of conflicts. An example of this is h_2 in Figure 3.13. As one can see, its conflict graph is acyclic, but the only possible OP-witness is the g-history itself, which is obviously not legal. This leads to the following observation:

Observation 1. *If a g-history is conflict opaque, it is not necessarily value opaque.*

So summarizing, the main issue lies in that CO does not include values in its definition and does not consider commit invoked transactions to be readable. While the above g-histories are not overly realistic, a g-history similar to the one shown in Figure 3.14 can be generated by snapshot based TMs, such as the one presented by Riegel et al. [81]. In this g-history it is relevant what value of y is read to determine whether the g-history is value opaque or not. However, conflict opacity considers it not opaque in any case. In general, for a g-history its conflict order is not necessarily the same as the order of transactions in its OP-witnesses.

For the other direction, OP also does not imply CO without further assumptions. An example of this is h_3 in Figure 3.13. This g-history is opaque as can be seen by looking at g-history $h_{3,s}$. Its conflict graph is cyclic as 21 writes on x and y and commits in between a read of x and y from transaction 31. From this follows the observation.

Observation 2. *If a g-history is value opaque, it is not necessarily conflict opaque.*

Here the main culprit lies in the fact that value opacity does not factor in where values originated from. While in CO the assumption is made that it is always clear which transaction is being read, in value opacity an ambiguous value-based reads-from relation is used. In h_3 in Figure 3.13 transaction 11 writes 1 to x and transaction 21 does the same. If these values were not the same, $h_{3,s}$ would not be legal.

First set of assumptions In our previous work, we made two assumptions to avoid these fairly specific issues: The unique writers assumption and the no-out-of-thin-air reads assumption. The first requires each written value to be unique and the second requires that only values from the most recent writer on a variable are being read. Both together establish a most-recent reads-from relation for opacity.

The no-out-of-thin-air reads assumption requires that the values read by a transaction are the values from the most recent writer to the respective variables. That means if a transaction reads on x, it reads the value written by the transaction that most recently committed before the read event and wrote to x. This is the implicit assumption behind the conflict order of CO.

Assumption 2 (No Out-Of-Thin-Air Reads)**.** *A g-history has no out-of-thin-air reads when the value read by a read is always the value of the transaction which committed most recently before the read and wrote to the variable of the read.*

We define the most recent committed transaction for a read event as follows:

Definition 14. *We define the last committed writer for a read event $\mathbf{R}_t^{tr}(var, val)$ in g-history h, denoted as $LWC_h(\mathbf{R}_t^{tr}(var, val))$, as the transaction tr' s.t.*

1. $x \in WS_h^{vo}(tr')$,
2. $co_h(tr')$,
3. $\mathbf{C}_{thr(tr')}^{tr'} <_h \mathbf{R}_t^{tr}(var, val)$
4. *and* $\neg(\exists tr'' \in h : x \in WS_h^{vo}(tr'') \wedge \mathbf{C}_{thr(tr')}^{tr'} <_h \mathbf{C}_{thr(tr'')}^{tr''} <_h \mathbf{R}_t^{tr}(var, val))$.

Note that as mentioned in Chapter 2, we assume one transaction to not read the same value multiple time from the same variable. Thus, the above definition is well-defined for a given g-event and g-history. For example, the following g-history falls under the assumption:

$$\mathbf{B}_3^{31}\mathbf{B}_1^{11}\mathbf{W}_1^{11}(x, 2)\mathbf{W}_3^{31}(x, 1)\mathbf{C}_3^{31}\mathbf{C}_1^{11}\mathbf{B}_2^{21}\mathbf{R}_2^{21}(x, 2).$$

If transaction 21 were to read 1 for variable x, it would no longer fall under this assumption.
The unique writer assumption assumes that there are no duplicate values. This leaves no ambiguity to which write is read by a read. This is a reasonable assumption when for instance timestamps are used.

Assumption 3 (Unique Writers)**.** *A g-history has unique writers whenever the value written by each of its writes is pairwise different to the values written by each other write of the g-history.*

Given these assumptions, we can show that conflict opacity implies value opacity.

Lemma 6. *Conflict opacity does imply value opacity under the assumption of unique writers and no-out-of-thin-air-reads.*

Proof. A more formal proof was done in our previous work [56]. We will present a proof sketch here. If a g-history h is conflict opaque, a conflict order corresponding to a serial g-history h_s, in which each unfinished transaction is aborted, exists for it. This g-history is obviously equivalent to a g-history in $compl(h)$, preserves the real-time order of h and is serial. So it is left to show it is legal. We show that by proving $LWC_h(\mathbf{R}_t^{tr}(var, val)) = LWC_{h_s}(\mathbf{R}_t^{tr}(var, val))$. This in combination with the no-out-of-thin-air-reads assumption proves our point. This claim follows from the fact that for a variable the commits of all transactions writing on it and all reads on that variable are ordered in the conflict order as they are in h. Thus, if in h a transaction is the most recent writer on a variable before a read on that variable, it is also in h_s. By the no-out-of-thin-air-reads assumption, this also means that their values match; thus, h_s is legal. □

Even under these assumptions OP does not imply CO.

Lemma 7. *Value opacity does not imply conflict opacity even under the assumption of unique writers and no-out-of-thin-air-reads.*

Proof. See the g-history in Figure 3.15. Its conflict graph is trivially cyclic. But there exists an OP-witness with the serial order 31 21 11. □

For OP to imply CO, we need to make further assumptions which we do in the form of the read-before-update assumption.

$$\mathbf{B}_{1}^{11}\mathbf{B}_{3}^{31}\mathbf{B}_{2}^{21}\mathbf{W}_{3}^{31}(x,1)\mathbf{W}_{1}^{11}(x,2)\mathbf{C}_{3}^{31}\mathbf{R}_{2}^{21}(x,1)\mathbf{C}_{1}^{11}\mathbf{W}_{2}^{21}(x,3)\mathbf{C}_{2}^{21}$$

Figure 3.15: CO does not imply OP under assumptions.

$$\begin{aligned}
h_1 &= \mathbf{B}_{1}^{11}\mathbf{B}_{2}^{21}\mathbf{W}_{1}^{11}(x,1)\mathbf{R}_{2}^{21}(x,0)\mathbf{C}_{1}^{11}\mathbf{W}_{2}^{21}(x,2)\mathbf{C}_{2}^{21}\\
h_2 &= \mathbf{B}_{1}^{11}\mathbf{B}_{2}^{21}\mathbf{W}_{1}^{11}(x,1)\mathbf{R}_{1}^{11}(x,0)\mathbf{C}_{1}^{11}\mathbf{R}_{2}^{21}(x,1)\mathbf{W}_{2}^{21}(x,2)\mathbf{C}_{2}^{21}\\
h_3 &= \mathbf{B}_{1}^{11}\mathbf{B}_{2}^{21}\mathbf{R}_{2}^{21}(x,0)\mathbf{R}_{1}^{11}(x,0)\mathbf{C}_{1}^{11}\mathbf{W}_{2}^{21}(x,2)\mathbf{C}_{2}^{21}
\end{aligned}$$

Figure 3.16: Examples for read-before-update assumption.

Read-before-update-assumption This assumption - as with SSR and SR - leads to OP and CO being equivalent. We will first state the assumption and then discuss what properties it establishes for the OP-witness of g-histories.

Assumption 4. *A g-history h fulfils the read-before-update assumption whenever for each completed transaction tr it holds that*

$$RS_h^{vo}(tr) \supseteq WS_h^{vo}(tr).$$

In Figure 3.16, three examples are shown, h_1 does not fulfil the assumption, h_2 and h_3, on the other hand, do. In the following we will show that for a g-history under the no-out-of-thin-air, unique writers and read-before-update assumptions the real-time order of any OP-witness of it is a superset of the conflict order of it. We will do this by showing that two things hold for an arbitrary g-history:

1. The real-time order of any of its OP-witness is a superset of the conflict order of all write/write conflicts of it.
2. The real-time order of any of its OP-witness is a superset of the conflict order of all read/write and write/read conflicts of it.

We start by showing that for an arbitrary g-history two committed transactions writing to at least one shared variable are ordered the same in itself and any OP-witness. First, we look at why this is not the case without the read-before-update assumption. The following g-history is an example of this:

$$\mathbf{B}_{1}^{11}\mathbf{B}_{3}^{31}\mathbf{B}_{2}^{21}\mathbf{W}_{3}^{31}(x,1)\mathbf{C}_{3}^{31}\mathbf{R}_{2}^{21}(x,1)\mathbf{W}_{1}^{11}(x,2)\mathbf{C}_{1}^{11}\mathbf{W}_{2}^{21}(x,3)\mathbf{C}_{2}^{21}.$$

The write of transaction 11 is never read, and thus in an OP-witness it can be put in the beginning of an OP-witness as in:

$$\mathbf{B}_{1}^{11}\mathbf{W}_{1}^{11}(x,2)\mathbf{C}_{1}^{11}\mathbf{B}_{3}^{31}\mathbf{W}_{3}^{31}(x,1)\mathbf{C}_{3}^{31}\mathbf{B}_{2}^{21}\mathbf{R}_{2}^{21}(x,1)\mathbf{W}_{2}^{21}(x,3)\mathbf{C}_{2}^{21}.$$

But when the read-before-update assumption is introduced, this is no longer possible. This is because for each transaction that writes on a variable x this write is always being read in any OP-witness, except if the commit of the transaction is the last commit of all transactions writing to x. Modifying the previous g-history, one of the possible results is this g-history:

$$\mathbf{B}_{1}^{11}\mathbf{B}_{3}^{31}\mathbf{B}_{2}^{21}\mathbf{R}_{3}^{31}(x,0)\mathbf{W}_{3}^{31}(x,1)\mathbf{C}_{3}^{31}\mathbf{R}_{2}^{21}(x,1)\mathbf{R}_{1}^{11}(x,1)\mathbf{W}_{1}^{11}(x,2)\mathbf{C}_{1}^{11}\mathbf{W}_{2}^{21}(x,3)\mathbf{C}_{2}^{21}.$$

As every write is read in a witness except the last one, this g-history cannot be equivalent to any witness as two of its writes are not read. The next lemma states that the write/write part of the conflict order of a g-history is a subset of the real-time order of any of its OP-witnesses under our assumptions.

Lemma 8. *Given a g-history h fulfilling the no-out-of-thin-air-reads, unique writers and read-before-update assumptions, for any OP-witness h_s of it, it holds that*

$$\{(tr_1, tr_2) \mid tr_1, tr_2 \in h, \exists var \in WS_h^{vo}(tr_1) \cap WS_h^{vo}(tr_2), \mathbf{C}_{thr(tr_1)}^{tr_1} <_h \mathbf{C}_{thr(tr_2)}^{tr_2}\} \subseteq h_s.RT$$

Proof Sketch. We show this by first showing that h can only be opaque if for each committed transaction tr writing to a variable x there exists another transaction tr' that reads x from tr. The only exception to this is if tr is the transaction writing to x and committing last in h. We denote it tr_{lx} here. Assume this is not the case, for an OP-witness to be legal tr must be the last transaction writing to x and committing in h_s else it is read by the next transaction writing to x and committing by the read-before-update assumption. But then the write on x by tr_{lx} is being read in h_s as there is at least one transaction ordered after it writing to x and thus also reading from it.

Now we show that for each variable x, if two transactions writing on it are adjacent (meaning there is no other commit of a transaction writing on x in between), then they are also adjacent in h_s. For adjacent transactions tr_1 and tr_2 with tr_1 committing first it must hold that tr_2 reads x from tr_1 in h_s by our assumptions. Thus, they must also be adjacent in h_s else it is not legal by the unique writers assumption. This means that the order of commits of transactions writing on x is also their order in h_s. From this the lemma follows. $\square$

Now we show the second claim made above. In the conflict order of conflict opacity, a read on a variable is totally ordered with each commit of transactions writing to its variable. For value opacity without the read-before-update assumption, an OP-witness for a g-history does not necessarily follow that constraint. An example of this is the following g-history:

$$\mathbf{B}_{1}^{11}\mathbf{B}_{3}^{31}\mathbf{B}_{2}^{21}\mathbf{W}_{3}^{31}(x,1)\mathbf{W}_{1}^{11}(x,2)\mathbf{C}_{1}^{11}\mathbf{C}_{3}^{31}\mathbf{R}_{2}^{21}(x,1)\mathbf{C}_{2}^{21}.$$

For which the following g-history is an OP-witness:

$$\mathbf{B}_{3}^{31}\mathbf{W}_{3}^{31}(x,1)\mathbf{C}_{3}^{31}\mathbf{B}_{2}^{21}\mathbf{R}_{2}^{21}(x,1)\mathbf{C}_{2}^{21}\mathbf{B}_{1}^{11}\mathbf{W}_{1}^{11}(x,2)\mathbf{C}_{1}^{11}.$$

As one may note from the example, not only is the commit of transaction 11 ordered oppositely with the read on x but also the commit of transaction 31. The next g-history is one of the possible results of adding read events to the previous g-history so that the result fulfils the read-before-update assumption:

$$\mathbf{B}_{1}^{11}\mathbf{B}_{3}^{31}\mathbf{B}_{2}^{21}\mathbf{W}_{3}^{31}(x,1)\mathbf{R}_{1}^{11}(x,0)\mathbf{W}_{1}^{11}(x,2)\mathbf{C}_{1}^{11}\mathbf{R}_{3}^{31}(x,2)\mathbf{C}_{3}^{31}\mathbf{R}_{2}^{21}(x,1)\mathbf{C}_{2}^{21}.$$

For this g-history the only OP-witness is:

$$\mathbf{B}_{1}^{11}\mathbf{R}_{1}^{11}(x,0)\mathbf{W}_{1}^{11}(x,2)\mathbf{C}_{1}^{11}\mathbf{B}_{3}^{31}\mathbf{R}_{3}^{31}(x,2)\mathbf{W}_{3}^{31}(x,1)\mathbf{C}_{3}^{31}\mathbf{B}_{2}^{21}\mathbf{R}_{2}^{21}(x,1)\mathbf{C}_{2}^{21}.$$

As by the properties discussed in the previous paragraph, an OP-witness with transaction 11 placed at the end is no longer possible. For a given variable the write/write conflicts order all transactions writing to that variable. For an OP-witness to be legal, each read of that variable must read from the transaction that wrote to its variable and committed most recently to it in the g-history (because of the no-out-of-thin-air and unique writer assumptions). This means it must be ordered directly after it, which means it is ordered after each other writer on its variable that came before it in the g-history, and it is also ordered before each other writer on its variable that came after it in the g-history. Thus, we can state the following lemma, saying that the real-time order of any OP-witness for a given g-history contains the union of the R/W and W/R conflicts.

Lemma 9. *Given a g-history h fulfilling the no-out-of-thin-air-reads, unique writers and read-before-update assumptions, for any OP-witness h_s of it, it holds that*

$$\begin{gathered}\{(tr_2, tr_1) \mid tr_1, tr_2 \in h, \exists var \in RS_h^{vo}(tr_1) \cap WS_h^{vo}(tr_2), \mathbf{C}_{thr(tr_2)}^{tr_2} <_h \mathbf{R}_{thr(tr_1)}^{tr_1}(var)\} \\ \cup \\ \{(tr_1, tr_2) \mid tr_1, tr_2 \in h, \exists var \in WS_h^{vo}(tr_1) \cap RS_h^{vo}(tr_2), \mathbf{R}_{thr(tr_1)}^{tr_1}(var) <_h \mathbf{C}_{thr(tr_2)}^{tr_2}\} \\ \subseteq \\ h_s.RT.\end{gathered}$$

Proof Sketch. Let h be a g-history fulfilling the no-out-of-thin-air, unique writers and read-before-update assumptions. Let h_s be an arbitrary OP-witness of it. Let $ev_r = \mathbf{R}_{thr(tr)}^{tr}(var, val)$ be an arbitrary read in h. Let $ev_c = \mathbf{C}_{thr(tr')}^{tr'}$ be a commit of a transaction writing to var. If tr' is the transaction ev_r reads from, then tr is ordered after tr' in h_s, with no other writer on var in between. Else h_s would not be legal. Let tr'' be another writer on var. If its commit is ordered before ev_c is in h, then by Lemma 8 it is ordered before tr' and in h_s and thus also before tr. If its commit is ordered after ev_c, then by the facts that there is no other writer on var in between tr' and tr, and that tr'' must be ordered after tr' by Lemma 8 it is ordered after tr. □

Combining both lemmas, we can state the following lemma:

Lemma 10. *Given a g-history h fulfilling the no-out-of-thin-air-reads and read-before-update assumptions, for any OP-witness h_s of it, it holds that*

$$\{(tr_1, tr_2) \mid tr_1, tr_2 \in h, \exists var \in WS_h^{vo}(tr_{\mathit{1}}) \cap WS_h^{vo}(tr_{\mathit{2}}), \mathbf{C}_{thr(tr_{\mathit{1}})}^{tr_{\mathit{1}}} <_h \mathbf{C}_{thr(tr_{\mathit{2}})}^{tr_{\mathit{2}}}\}$$
$$\cup$$
$$\{(tr_2, tr_1) \mid tr_1, tr_2 \in h, \exists var \in RS_h^{vo}(tr_{\mathit{1}}) \cap WS_h^{vo}(tr_{\mathit{2}}), \mathbf{C}_{thr(tr_{\mathit{2}})}^{tr_{\mathit{2}}} <_h \mathbf{R}_{thr(tr_{\mathit{1}})}^{tr_{\mathit{1}}}(var)\}$$
$$\cup$$
$$\{(tr_1, tr_2) \mid tr_1, tr_2 \in h, \exists var \in WS_h^{vo}(tr_{\mathit{1}}) \cap RS_h^{vo}(tr_{\mathit{2}}), \mathbf{R}_{thr(tr_{\mathit{1}})}^{tr_{\mathit{1}}}(var) <_h \mathbf{C}_{thr(tr_{\mathit{2}})}^{tr_{\mathit{2}}}\}$$
$$\subseteq$$
$$h_s.RT.$$

This together with the previous paragraph leads to the following theorem:

Theorem 2. *Under the unique writers, no-out-of-thin-air-reads and read-before-update assumptions a g-history is opaque iff it is conflict opaque.*

4 Correctness Problem

In this chapter, we will present our results regarding the correctness problem for strict state serializability and value opacity. The result for strict state serializability is based on previous work of us [58], while the result for value opacity is a new contribution. We will first present the related work, then the result for state serializability and finally the result for value opacity.

4.1 Related Work

In this section, we will discuss the related work regarding the correctness problem. As in the previous chapter, we will divide the related work into publications regarding derived correctness conditions and publications regarding correctness conditions specifically designed for TMs. Overall, the related work is a lot more sparse in terms of theoretical results compared to the membership problem for the same conditions. Still, there has been a good number of model checking approaches presented for TMs.

Derived correctness conditions The correctness problem for sequential consistency for arbitrary implementations is undecidable [1]. Alur et al. showed

Condition	Complexity of correctness problem
Sequential consistency	Undecidable [3]
Linearizability	EXPSPACE [3]
View serializability	Unknown
Strict view serializability	Unknown
State serializability	Unknown
Strict state serializability	Decidable [a] (Section 4.2)
Conflict serializability	PSPACE [37]
Causal serializability	Unknown
Snapshot isolation	Unknown
Opacity	Decidable [b] (Section 4.3)
Conflict opacity	PSPACE (reduction to conflict serializability)
DU opacity	Unknown
TMS 1	Unknown
TMS 2	Unknown
VWC	Unknown

[a]Under assumptions, see Section 4.2

[b]Under assumptions, see Section 4.3

Table 4.1: Complexity of the correctness problems for different correctness conditions

that the problem is still undecidable under the assumption of read/write objects and more than 3 threads [3]. There are several works showing that under specific assumptions for the implementation, system, number of threads, variables and values and similar aspects sequential consistency is decidable [49, 78, 20, 17, 12].

The correctness problem for linearizability is in EXPSPACE as shown by Alur et al. [3]. Bouajjani et al. showed that for an unbounded number of threads this problem is undecidable [15]. However, it is EXPSPACE-complete whenever the linearization point of each operation is known beforehand. As with sequential consistency, there are several works showing that, even for an unbounded number of threads, the problem is decidable for specific types of programs/implementations [16, 89, 7, 68, 18, 66].

For view serializability there is no work regarding the general correctness problem or subclasses of it. As Farzan et al. put it, the correctness problem for view serializability is hard, and the condition itself is not “well-behaved” [36] as it for example is not monotonic (if a history is correct each projection on a subset of transactions must also be correct).

We have shown strict state serializability to be decidable under assumptions in previous work [58], for which we present an improved version in this thesis in Section 4.2 following this section. As with view serializability there is no work with regard to other subclasses of this problem.

The most studied serializability criterion with regard to the correctness problem is conflict serializability. Model checking a DFA with a bounded number of threads for conflict serializability is in PSPACE. This was shown by Farzan et al. who fixed the errors of an earlier proof published by Alur et al. [37, 3]. Boujjani et al. showed that the variant of the correctness problem with an unbounded number of threads is EXPSPACE-complete [15]. Guerraoui et al. gave a reduction theorem making it possible to verify conflict serializability with a real-time order on TMs that fulfil certain properties [42]. There is some work that is more practical with regard to verifying serializability variants for TMs [71, 19, 33].

There is no work for the correctness problem of causal serializability and snapshot isolation or the verification of these conditions for TMs.

Correctness conditions specifically designed for TMs For value opacity we present a proof that value opacity with an unambiguous value-based reads-from relation is decidable under assumptions in Section 4.3. There is no further work with regard to this specific problem. There has been work about making value opacity easier to verify for TMs. Armstrong et al. developed a method where opacity verification was simplified by using existing results for linearizability [4]. They showed that to verify the value opacity of an implementation it is sufficient to show that it linearizes to a value opaque abstraction. Lesani and Palsberg presented a condition called markability, which is equivalent to explicitly prefix-closed value opacity, but breaks it down into 3 subconditions. These conditions, according to the authors, are easier to prove for TM algorithms as they are closer to the design intuitions of TMs compared to value opacity [63]. We have published research about the application of data independence for checking value opacity [57]. There also have been several proofs of opacity for specific TM implementations [25, 28, 83, 63, 24].

With regard to conflict opacity, there is no work for the correctness problem definition we use in this thesis. The results of Farzan et al. and Bouajjani et al. for conflict serializability can be used here [37, 15] as conflict opacity can be reduced to conflict serializability by transforming each aborted transaction into a read only committed transaction. Guerraoui et al. gave a reduction theorem making it possible to verify conflict opacity on TMs fulfilling certain properties [42].

There is no work for the theoretical correctness problem for DU-Opacity, VWC and TMS 1 and 2. TMS 2 was proven for specific TMs [28, 24, 83], mostly as a means to prove their value opacity.

4.2 The Correctness Problem for SSR^- Is Decidable

In this section, we show decidability for SSR^-, a subclass of SSR^+. In SSR^- we are restricted to p-histories in which all transactions are live or unfinished. Note that with the definition of the correctness problem used in this thesis the number of threads and variables used in each implementation is finite, as implementations are modeled as DFAs.

The decidability of the correctness problem follows from the fact that we can construct (approximations of) equivalence classes of p-histories. The equivalence classes capture the strict state serializability of p-histories and their extensions. Then we can reduce the given implementation automaton to a finite automaton whose states contain these equivalence classes. The language of this automaton is empty if and only if all p-histories generated by the implementation automaton are strictly serializable. The assumption of all transactions being live or unfinished limits the information that characterizes the states of the equivalence class automaton.

In the following we assume (1) all p-histories to contain live or unfinished transactions only and (2) an implementation automaton to only accept words (p-histories) in which all transactions are finished. We can therefore employ a notion of equivalence of two p-histories meaning both share (a) the same set of transactions and

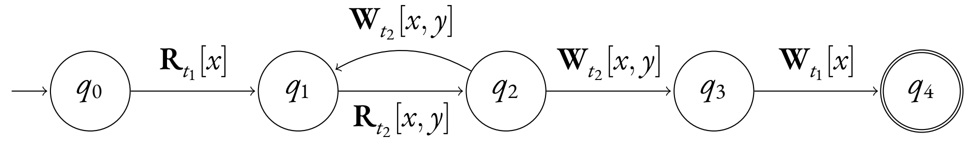

$$\begin{tikzcd}
\arrow[r] & q_0 \arrow[r, "\mathbf{R}_{t_1}[x]"] & q_1 \arrow[r, "\mathbf{R}_{t_2}[x,y]"'] & q_2 \arrow[l, bend right, "\mathbf{W}_{t_2}[x,y]"'] \arrow[r, "\mathbf{W}_{t_2}[x,y]"] & q_3 \arrow[r, "\mathbf{W}_{t_1}[x]"] & q_4
\end{tikzcd}$$

Figure 4.1: Implementation automaton example: I_{ex}.

(b) same reads-from relation (for all transactions, not just live ones). The notion of SSR^{-}-witness used in the sequel is based on this adapted equivalence definition.

4.2.1 Compact Representation

We start by looking at a naive approach for generating all p-histories of an implementation automaton and explain how to compress these infinitely many p-histories to some finite structure. In this section, we use the example implementation automaton I_{ex} which can be seen in Figure 4.1. It accepts the language

$$L(I_{ex}) = \mathbf{T_w}\mathbf{R}_{t_1}[x]\left(\mathbf{R}_{t_2}[x,y]\mathbf{W}_{t_2}[x,y]\right)^{+}\mathbf{W}_{t_1}[x]\mathbf{T_r},$$

where an entire transaction of the form $\mathbf{R}^{j}_{t_i}[x]\mathbf{W}^{j}_{t_i}[x]$ is for brevity denoted as $\mathbf{T}^{j}_{t_i}[x]$. As one can easily see, no p-history it produces is serializable. Given an arbitrary p-history produced by it, in any SSR^{-}-witness of this history either all transactions of t_2 must happen before or after the transaction of t_1 as else the transactions of t_2 do not read x from each other. Let tr_1 be the first transaction of t_1 and tr_2 be the last transaction of t_2. In the first case, tr_1 does not read x from tr_w but from tr_2. In the second case, tr_r does not read x from tr_1 but from tr_2. Thus, there can be no SSR^{-}-witness for any p-history in $L(I_{ex})$.

Given such an implementation automaton, a naive approach would be to simply explore the entire state space of the implementation, i.e. to generate all of its p-histories and check them for strict serializability. An excerpt of the state space (shown as a graph) of the implementation automaton I_{ex} can be seen in Figure 4.2. The upper half of each node shows the current state of the automaton. The lower half shows the p-history of the p-events executed so far and the set of SSR^{-}-witnesses for that p-history. In this example, all transactions are not coloured, we

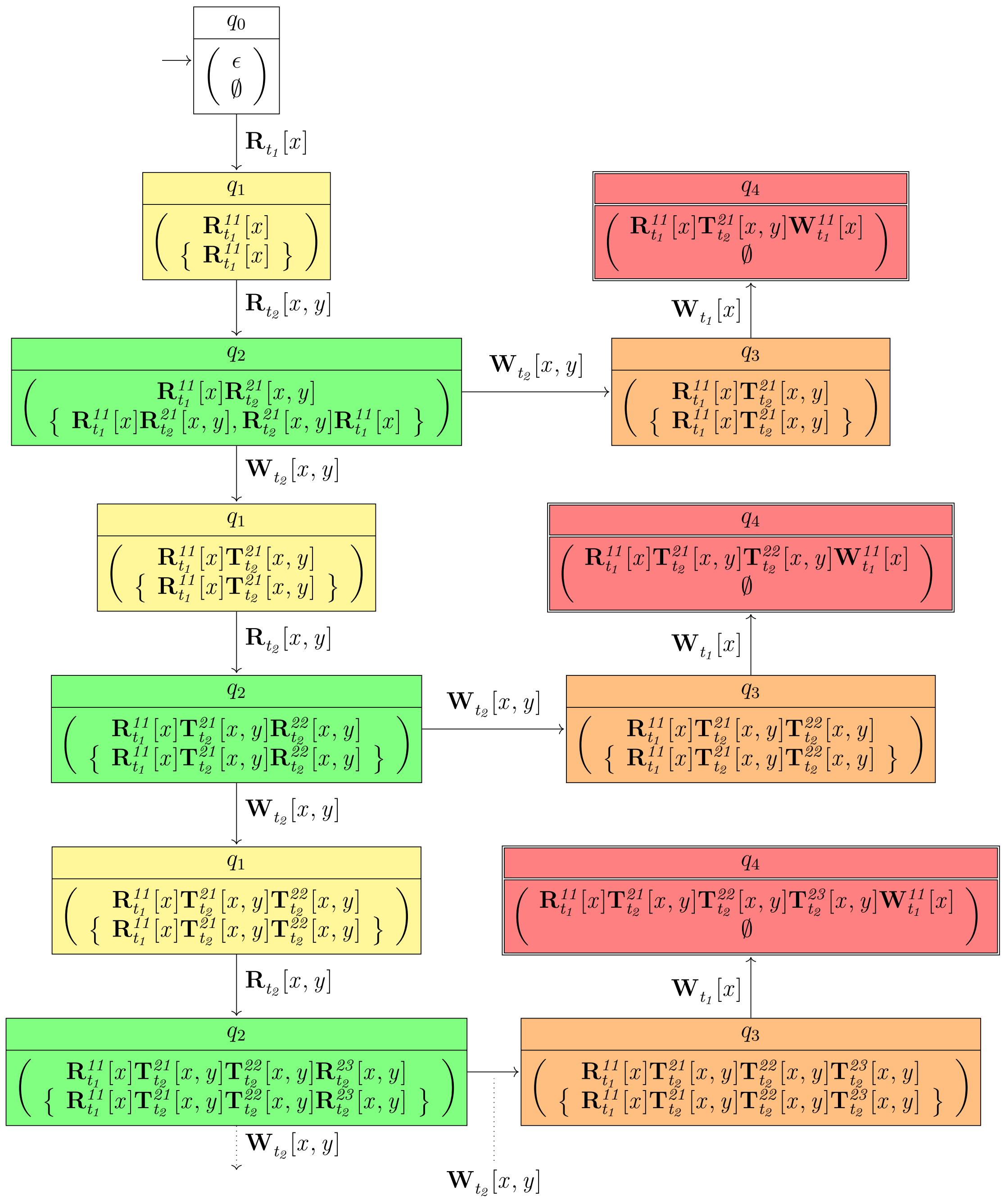

q_0
ϵ
$\emptyset$
$\mathbf{R}_{t_1}[x]$
q_1
$\mathbf{R}^{11}_{t_1}[x]$
$\{ \mathbf{R}^{11}_{t_1}[x] \}$
$\mathbf{R}_{t_2}[x,y]$
q_2
$\mathbf{R}^{11}_{t_1}[x]\mathbf{R}^{21}_{t_2}[x,y]$
$\{ \mathbf{R}^{11}_{t_1}[x]\mathbf{R}^{21}_{t_2}[x,y], \mathbf{R}^{21}_{t_2}[x,y]\mathbf{R}^{11}_{t_1}[x] \}$
$\mathbf{W}_{t_2}[x,y]$
q_3
$\mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y]$
$\{ \mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y] \}$
$\mathbf{W}_{t_1}[x]$
q_4
$\mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y]\mathbf{W}^{11}_{t_1}[x]$
$\emptyset$
$\mathbf{W}_{t_2}[x,y]$
q_1
$\mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y]$
$\{ \mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y] \}$
$\mathbf{R}_{t_2}[x,y]$
q_2
$\mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y]\mathbf{R}^{22}_{t_2}[x,y]$
$\{ \mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y]\mathbf{R}^{22}_{t_2}[x,y] \}$
$\mathbf{W}_{t_2}[x,y]$
q_3
$\mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y]\mathbf{T}^{22}_{t_2}[x,y]$
$\{ \mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y]\mathbf{T}^{22}_{t_2}[x,y] \}$
$\mathbf{W}_{t_1}[x]$
q_4
$\mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y]\mathbf{T}^{22}_{t_2}[x,y]\mathbf{W}^{11}_{t_1}[x]$
$\emptyset$
$\mathbf{W}_{t_2}[x,y]$
q_1
$\mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y]\mathbf{T}^{22}_{t_2}[x,y]$
$\{ \mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y]\mathbf{T}^{22}_{t_2}[x,y] \}$
$\mathbf{R}_{t_2}[x,y]$
q_2
$\mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y]\mathbf{T}^{22}_{t_2}[x,y]\mathbf{R}^{23}_{t_2}[x,y]$
$\{ \mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y]\mathbf{T}^{22}_{t_2}[x,y]\mathbf{R}^{23}_{t_2}[x,y] \}$
q_3
$\mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y]\mathbf{T}^{22}_{t_2}[x,y]\mathbf{T}^{23}_{t_2}[x,y]$
$\{ \mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y]\mathbf{T}^{22}_{t_2}[x,y]\mathbf{T}^{23}_{t_2}[x,y] \}$
$\mathbf{W}_{t_1}[x]$
q_4
$\mathbf{R}^{11}_{t_1}[x]\mathbf{T}^{21}_{t_2}[x,y]\mathbf{T}^{22}_{t_2}[x,y]\mathbf{T}^{23}_{t_2}[x,y]\mathbf{W}^{11}_{t_1}[x]$
$\emptyset$
$\mathbf{W}_{t_2}[x,y]$
$\mathbf{W}_{t_2}[x,y]$

Figure 4.2: Excerpt of state space of I_{ex} (Figure 4.1) (augmented transactions not shown)

also left out tr_w and tr_r for brevity.

The problem with this approach is easy to see. The state space of implementations can be infinite as there are infinitely many p-histories. So our approach is to reduce the state space by *merging* nodes which behave similarly. In the graph in Figure 4.2, these are marked with the same color. For example, consider the green states (first column, third, fifth and seventh state). Whenever we execute $\mathbf{W}_{t_2}[x, y]\mathbf{W}_{t_1}[x]$ from a green node, we end up in a node with implementation state q_4 and an empty SSR^--witness, i.e. the current p-history is not strictly serializable. Whenever we execute $\mathbf{W}_{t_2}[x, y]$, we either end up in a node with implementation state q_3 or q_1. In both cases the corresponding p-history is strictly serializable. So summarizing, we consider two nodes as behaving similarly whenever:

- they contain the same implementation automaton state
- and when after appending identical p-events to their respective histories, the resulting histories are both strictly serializable or are both not.

Merging two such nodes into one does not change the accepted language of the automaton. We show our main result by proving that such a graph with merged nodes has (a) a finite number of nodes (and thus is representable as a finite automaton) and (b) this automaton is effectively constructable.

We start by formalizing the above similarity on p-histories. Recall that the concatenation of two histories does not include their augmented transactions. Instead, their non-augmented transactions are concatenated, and then the new history is assumed to be augmented again.

Definition 15 (*SSR*-extension equivalence)**.** *Two p-histories $ph, ph' \in \mathcal{PH}$ are SSR^--extension equivalent ($ph \equiv_{ext} ph'$) iff $\forall n \in \mathbb{N}, \forall pev_0 \dots pev_n \in PEv^{n+1}$ either:*

- *$ph \cdot pev_0 \dots pev_n$ and $ph' \cdot pev_0 \dots pev_n$ are both strictly serializable*
- *or $ph \cdot pev_0 \dots pev_n$ and $ph' \cdot pev_0 \dots pev_n$ are both not strictly serializable.*

The question is how to determine whether two p-histories are SSR^{-}-extension equivalent. The general idea is to reduce a p-history to the essential information needed to determine whether appending p-events keeps the p-history strictly serializable or not. We call this information *SSR^{-}-data*. Whenever two p-histories have the same SSR^{-}-data, they are SSR^{-}-extension equivalent. Below, we will show that there are only finitely many different (valid) SSR^{-}-data for a given number of threads and variables. This is key to our decidability result.

The remainder of this section is structured as follows. We first present *candidate sets* - sets of potential SSR^{-}-witnesses - which are an explicit representation of the implicit data needed to compute the strict serializability of a p-history. Based on this concept, we present a compression method for pairs of p-histories and candidates which compresses them into equivalence classes. Then we show how this method can be used to compress complete candidate sets and their p-histories to SSR^{-}-data. Finally, using SSR^{-}-data we present the automata reduction construction which proves the decidability of SSR^{-}.

Candidate set The (witness) candidate set is a summarization of the implicit information in a p-history that is needed to compute its serializability. Candidate sets possess a property called the *supersequence property*, which enables their compression and is discussed later on. In this paragraph, we will define candidate sets and introduce the supersequence property and its implications for the extension of candidate sets for p-histories. This will lead to the compression presented in the next paragraph.

The candidate set of a p-history overapproximates the set of its SSR^{-}-witnesses and is defined as follows.

Definition 16 (Candidate set). *The candidate set of a p-history ph, C_{ph} is defined as the set of all p-histories, which*

- *contain the same events as ph,*
- *are serial*
- *and preserve the real-time order of ph.*

$$ph'_e = \mathbf{T}^{11}_{t_1}[y]\mathbf{R}^{21}_{t_2}[x,y]\mathbf{R}^{12}_{t_1}[x] \quad \rightarrow \quad ph_e = \mathbf{T}^{11}_{t_1}[y]\mathbf{R}^{21}_{t_2}[x]$$

$$C_{ph'_e} = \left\{ \begin{matrix} \mathbf{T}^{11}_{t_1}[y]\mathbf{R}^{21}_{t_2}[x,y]\mathbf{R}^{12}_{t_1}[x] \\ \mathbf{T}^{11}_{t_1}[y]\mathbf{R}^{12}_{t_1}[x]\mathbf{R}^{21}_{t_2}[x,y] \end{matrix} \right\} \rightarrow C_{ph_e} = \left\{ \mathbf{T}^{11}_{t_1}[y]\mathbf{R}^{21}_{t_2}[x,y] \right\}$$

Figure 4.3: Example of the supersequence property for candidate sets.

So basically the candidate set of ph is the set of p-histories that fulfil every condition to be an SSR^{-}-witness of ph, except having an equal reads-from relation. Candidate sets possess the supersequence property. This property expresses that given a history and an extension of it, each candidate of the extended p-history is a supersequence of a candidate of the unextended p-history.

Proposition 1 (Supersequence property)**.** *Given a p-history ph and an arbitrary extension ph' of it, it holds that*

$$\forall ph'_c \in C_{ph'}, \exists ph_c \in C_{ph} : ph_c \sqsubseteq ph'_c.$$

Note that the supersequence and subsequence definitions do not include augmented transactions. We disregard augmented transactions completely for this paragraph as their order in each p-history is completely identical and given by definition. Also, note that the additional events of ph'_c in the definition are exactly the events by which ph is extended.

We will argue why the supersequence property holds for the extension by one p-event pev, which by induction implies the property for arbitrary extensions. For this explanation let ph be a p-history and ph' be an extension of it. We use the example from Figure 4.3 to illustrate the argument. First, note that the real-time order of ph' consists of the real-time order of ph united with the set containing all rt-elements involving pev, denoted RT_{pev}, which is formalized as follows

$$ph'.RT = ph.RT \cup RT_{pev}.$$

The set RT_{pev} is non-empty iff pev is a read, and both sets are obviously disjoint.

In Figure 4.3, ph'_e has the real-time order of ph_e plus the rt-elements involving 12. In this case, there is only one rt-element (11, 12). By definition, any candidate of ph', denoted ph'_c, preserves its real-time order:

$$ph.RT \cup RT_{pev} \subseteq ph'_c.RT.$$

Now, if we remove pev, all rt-elements involving it are removed as well which results in

$$ph.RT \subseteq ph'_c.RT \setminus RT_{pev}.$$

Let ph'_{c-} be ph'_c with pev removed, then it holds that

$$ph.RT \subseteq ph'_{c-}.RT.$$

As ph'_{c-} is trivially serial, it is a candidate for ph. In the example, for the candidate $\mathbf{T}^{11}_{t_1}[y]\mathbf{R}^{12}_{t_1}[x]\mathbf{R}^{21}_{t_2}[x,y]$ becomes $\mathbf{T}^{11}_{t_1}[y]\mathbf{R}^{21}_{t_2}[x,y]$, which is a candidate of ph_e.

This property allows us to define a function for the extension of candidate sets that models how a candidate set changes when its p-history is extended. We will do this for extensions of one event. This can inductively be expanded to extensions by multiple events. Assume we have a p-history ph that is extended to ph' by pev. Each candidate ph_c of ph can spawn a number of candidates for ph', which are its supersequence. All of its supersequences can be generated by inserting pev at arbitrary points in the p-history, but not all of them are candidates. In the following, we define which insertion points lead to candidates. We distinguish between the case where pev is a write and where it is a read. First, the extension by a write event will be covered, and then the extension by a read event will be covered.

For a write there is only one insertion point that results in a candidate for the extended p-history. It must be inserted directly after the last read of its transaction, otherwise the resulting p-history is not serial. If it is inserted that way, new rt-elements can be added to the real-time order of the candidate but none are removed. This preserves the real-time order of ph' because the real-time order of ph' is equal to the real-time order of ph. An example of the extension of

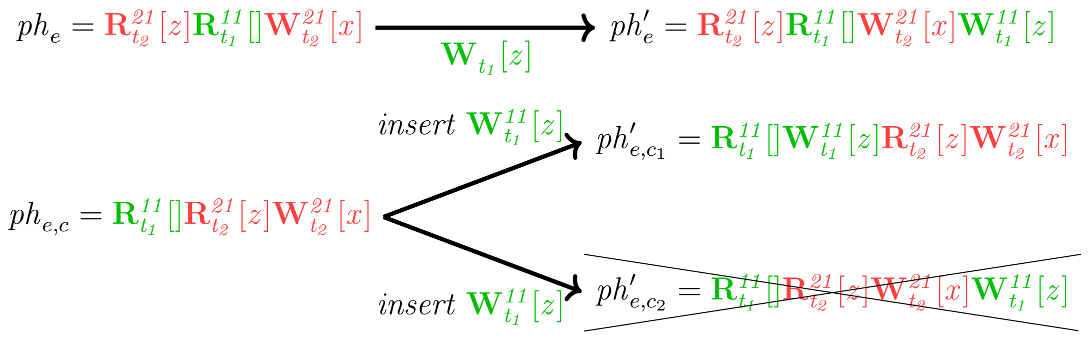

$ph_e = \mathbf{R}^{21}_{t_2}[z]\mathbf{R}^{11}_{t_1}[]\mathbf{W}^{21}_{t_2}[x]$
$\mathbf{W}_{t_1}[z]$
$ph'_e = \mathbf{R}^{21}_{t_2}[z]\mathbf{R}^{11}_{t_1}[]\mathbf{W}^{21}_{t_2}[x]\mathbf{W}^{11}_{t_1}[z]$
$insert\ \mathbf{W}^{11}_{t_1}[z]$
$ph'_{e,c_1} = \mathbf{R}^{11}_{t_1}[]\mathbf{W}^{11}_{t_1}[z]\mathbf{R}^{21}_{t_2}[z]\mathbf{W}^{21}_{t_2}[x]$
$ph_{e,c} = \mathbf{R}^{11}_{t_1}[]\mathbf{R}^{21}_{t_2}[z]\mathbf{W}^{21}_{t_2}[x]$
$insert\ \mathbf{W}^{11}_{t_1}[z]$
$ph'_{e,c_2} = \mathbf{R}^{11}_{t_1}[]\mathbf{R}^{21}_{t_2}[z]\mathbf{W}^{21}_{t_2}[x]\mathbf{W}^{11}_{t_1}[z]$

Figure 4.4: A p-history and its candidates extended with a write event

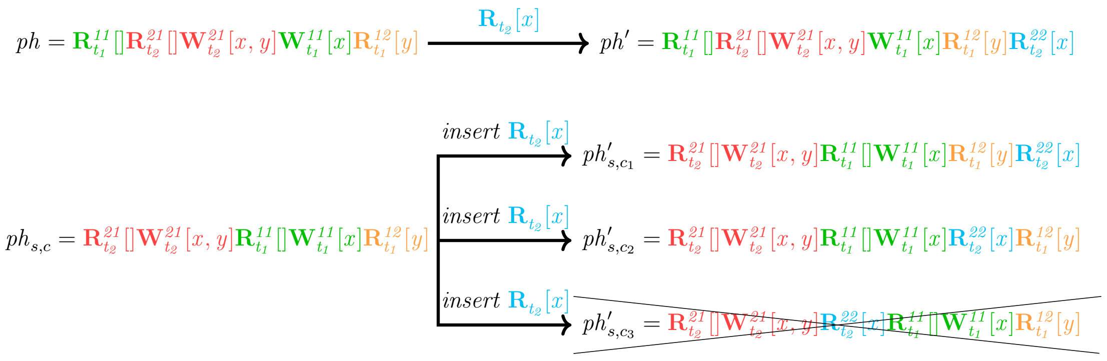

$ph = \mathbf{R}^{11}_{t_1}[]\mathbf{R}^{21}_{t_2}[]\mathbf{W}^{21}_{t_2}[x,y]\mathbf{W}^{11}_{t_1}[x]\mathbf{R}^{12}_{t_1}[y]$
$\mathbf{R}_{t_2}[x]$
$ph' = \mathbf{R}^{11}_{t_1}[]\mathbf{R}^{21}_{t_2}[]\mathbf{W}^{21}_{t_2}[x,y]\mathbf{W}^{11}_{t_1}[x]\mathbf{R}^{12}_{t_1}[y]\mathbf{R}^{22}_{t_2}[x]$
$insert\ \mathbf{R}_{t_2}[x]$
$ph'_{s,c_1} = \mathbf{R}^{21}_{t_2}[]\mathbf{W}^{21}_{t_2}[x,y]\mathbf{R}^{11}_{t_1}[]\mathbf{W}^{11}_{t_1}[x]\mathbf{R}^{12}_{t_1}[y]\mathbf{R}^{22}_{t_2}[x]$
$ph_{s,c} = \mathbf{R}^{21}_{t_2}[]\mathbf{W}^{21}_{t_2}[x,y]\mathbf{R}^{11}_{t_1}[]\mathbf{W}^{11}_{t_1}[x]\mathbf{R}^{12}_{t_1}[y]$
$insert\ \mathbf{R}_{t_2}[x]$
$ph'_{s,c_2} = \mathbf{R}^{21}_{t_2}[]\mathbf{W}^{21}_{t_2}[x,y]\mathbf{R}^{11}_{t_1}[]\mathbf{W}^{11}_{t_1}[x]\mathbf{R}^{22}_{t_2}[x]\mathbf{R}^{12}_{t_1}[y]$
$insert\ \mathbf{R}_{t_2}[x]$
$ph'_{s,c_3} = \mathbf{R}^{21}_{t_2}[]\mathbf{W}^{21}_{t_2}[x,y]\mathbf{R}^{22}_{t_2}[x]\mathbf{R}^{11}_{t_1}[]\mathbf{W}^{11}_{t_1}[x]\mathbf{R}^{12}_{t_1}[y]$

Figure 4.5: A p-history and its candidates extended with a read event.

the candidates by a write is shown in Figure 4.4. There is only one valid insertion point of $\mathbf{W}_{t_1}[z]$ in $ph_{e,c}$, which is behind the last read of transaction 11, resulting in ph_{e,c_1}. The real-time orders of ph_e and ph'_e are equal as a write is appended at the end. In ph'_{e,c_1} transaction 11 is now real-time ordered before 21, which it is not in $ph_{e,c}$. It preserves the real-time order of ph'_e, which is empty (excluding the augmented transactions). Any other insertion point does not yield a serial p-history, as exemplarily shown in ph_{e,c_2}.

When appending a read to ph, the transaction of the read is real-time ordered after every other finished transaction in ph'. Thus, to preserve the real-time order of ph', the read must be inserted after the last write in ph_c. Adding a read at that point always results in a serial p-history. In the example in Figure 4.5, ph'_{s,c_1} and ph'_{s,c_2} show such insertions for $ph_{s,c}$. The insertion for ph'_{s,c_3} shows an example of an insertion not following the rules described above. In that case, it does not preserve the real-time order of ph'_s.

Next, we define the insertion function which upon input of a candidate and a p-event returns a set containing all valid insertions following the above rules. We define an insertion function *ins* for writes and reads separately and combine them. Before that, we introduce some notation that is needed in the further text.

Definition 17 (Notation)**.** *Let ph be an arbitrary p-history and tr be an arbitrary transaction of it, then let*

- *$st(ph)$ be the subsequence of ph that starts at the first p-event of ph and ends at the last write of ph,*
- *$en(ph)$ be the subsequence of ph which includes all p-events after the last write,*
- *$add(ph, pev, n)$ be ph with pev inserted at index n,*
- *$pev_{ph}^{tr,rd}$ be the read event of tr in ph,*
- *$pev_{ph}^{tr,wr}$ be the write event of tr in ph, if it exists*
- *and $lsInd(ph)$ be the last index of ph.*

Given this notation, we can define the insertion function for a read, a write and the overall insertion function.

Definition 18 (Insertion function read)**.** *The insertion function for reads is denoted $ins_r : \mathcal{PH} \times PEv \to 2^{\mathcal{PH}}$. On input of a serial p-history ph_c and a read p-event pev, it returns the set*

$$\{st(ph_c) \cdot add(en(ph_c), pev, n) \mid 0 \leq n \leq lsInd(en(ph_c))\}.$$

Definition 19 (Insertion function write)**.** *The insertion function for writes is denoted $ins_w : \mathcal{PH} \times PEv \to 2^{\mathcal{PH}}$. On input of a serial p-history $ph_c =$*

$pev_0 \dots pev^{tr,rd}_{ph_c} \dots pev_n$ and a write p-event pev of transaction tr, it returns the set

$$\{pev_0 \dots pev^{tr,rd}_{ph_c} pev \dots pev_n\}.$$

Definition 20 (Insertion function)**.** *The insertion function is denoted $ins : \mathcal{PH} \times PEv \to 2^{\mathcal{PH}}$. On input of a serial p-history ph_c and a p-event pev, it returns a set of p-histories $\mathcal{PH}_c$ which is either*

1. *$ins_w(ph_c, pev)$, if pev is a write,*
2. *or $ins_r(ph_c, pev)$, if pev is a read.*

We also extend *ins* to arbitrary p-event sequences by first applying it to the input p-history and the first p-event of the sequence, then applying it to every element of the resulting set of the first application and the second event of the sequence and so on.

As we have stated above, inserting a p-event in such a way into a candidate of a p-history yields a candidate of this p-history extended by that p-event. We show that given an arbitrary p-history *ph* and its extension *ph′* by an arbitrary p-event *pev*, applying the insertion function with *pev* to each candidate of *ph* yields a number of candidate sets whose union is equal to the candidate set of *ph′*.

Proposition 2 (Generation of candidates by insertion function)**.** *Given a p-history ph and a p-event pev, it holds that*

$$\bigcup_{ph_c \in C_{ph}} ins(ph_c, pev) = C_{ph \cdot pev}.$$

The proof of this proposition can be found in Appendix B.

History candidate pairs The next question is then how to compress a p-history and its candidate set to SSR^--data in such a way that if this SSR^--data is identical for two p-histories, they are SSR^--extension equivalent. The first step towards this is to look at single candidates with their respective p-history instead of looking at complete candidate sets with their respective p-history. A p-history ph and its candidate ph_c are treated as a pair (ph, ph_c) called *hc-pair* (history candidate pair). An extension of an hc-pair refers to the extension of both its members. An hc-pair is called consistent iff its members are p-equivalent. We group multiple hc-pairs into an equivalence class if they share a broken down notion of extension equivalence. The notion is defined as follows.

Definition 21 (Extension equivalence for hc-pairs)**.** *Given two p-histories ph, ph' and candidates $ph_c \in C_{ph}$ and $ph'_c \in C_{ph'}$, the hc-pair (ph, ph_c) is extension equivalent to the hc-pair (ph', ph'_c), denoted as $(ph, ph_c) \equiv_{ext} (ph', ph'_c)$ iff for any arbitrary p-event sequence seq it holds that*

$$\exists ph_{c,ins} \in ins(ph_c, seq) : ph \cdot seq \equiv ph_{c,ins} \leftrightarrow \exists ph'_{c,ins} \in ins(ph'_c, seq) : ph' \cdot seq \equiv ph'_{c,ins}.$$

We can approximate the equivalence classes for this notion. Our approximation uses the fact that we can divide all rf-elements not involving tr_r in a reads-from relation of any p-history into two categories: *fixed* rf-elements and *interruptible* rf-elements. A fixed rf-element is in the reads-from relation of any extension of the p-history or the candidate. On the other hand, for an interruptible rf-element there exists an extension of the p-history or the candidate where a (new or already existing) transaction has a write in between the writing and reading transaction of the rf-element and is now read by the reading transaction instead. We exclude rf-elements involving tr_r from the second category as then any interruption of an rf-element in an hc-pair results in it being not consistent. As we will see later on, interruptible elements then only exist in candidates. Thus, we will not define them for p-histories. We will first define fixed rf-elements for p-histories and candidates, then define interruptible rf-elements for candidates and then give conditions of when rf-elements belong to one of these categories. We formally define fixed elements and related notation.

Definition 22 (Fixed reads-from elements for p-histories)**.** *Given a p-history ph,* *an rf-element* $(tr, tr', x) \in ph.RF$ *is called fixed iff*

$$\forall seq \in PEv^* : (tr, tr', x) \in (ph \cdot seq).RF.$$

Definition 23 (Fixed reads-from elements for candidates)**.** *Given a serial p-history* ph_c*, an rf-element* $(tr, tr', x) \in ph.RF$ *is called fixed iff*

$$\forall seq \in PEv^*, \forall ph_{c,ins} \in ins(ph_c, seq) : (tr, tr', x) \in ph_{c,ins}.RF.$$

In the second case, such fixed elements exist because of the supersequence property. If an rf-element $rf = (tr, tr', x)$ is element of $ph.RF^{fix}$ for an arbitrary p-history (or candidate) ph, we say $fix_{ph}(rf)$. Let $ph.RF^{fix}$ be the set of fixed rf-elements of an arbitrary p-history ph. Next we formally define interruptible elements in candidates and introduce additional notation related to this definition.

Definition 24 (Interruptible rf-element)**.** *Given a candidate* ph_c*, an rf-element* $(tr, tr', x) \in ph_c.RF$ *with* $tr' \neq tr_r$ *is called interruptible iff*

$$\exists seq \in PEv^*, \exists ph_{c,ins} \in ins(ph_c, seq), \exists tr'' \in Tr : (tr'', tr', x) \in ph_{c,ins}.RF \wedge tr'' \neq tr.$$

Note that all rf-elements involving tr_r are not fixed **and** not interruptible by the above definitions. If an rf-element $rf = (tr, tr', x)$ is interruptible in candidate ph_c, this is denoted $int_{ph_c}(rf)$. An unfinished transaction tr'' is called interrupting for an interruptible rf in the context of a candidate whenever its read is in between the write of tr and the read of tr'. This is denoted $int_{ph_c}(tr'', rf)$. For an unfinished transaction tr'', a variable x is called an interrupting write whenever there exists an rf-element (tr, tr', x) s.t. tr'' is interrupting for this element. This is denoted $int_{ph_c}(tr'', x)$. The *interrupting write set* of an unfinished transaction in a candidate is the set of all interrupting writes of it. We define this formally. Note that the set of all unfinished transactions of a p-history/candidate ph_c ($unfin(ph_c)$) is bounded in size by $|T|$.

Definition 25 (Interrupting write set)**.** *Given a candidate* ph_c *and a transaction*

$$ph_e = \mathbf{T_w}\mathbf{R}_{t_1}^{11}[x]\mathbf{R}_{t_2}^{21}[]\mathbf{W}_{t_1}^{11}[y,z]\mathbf{R}_{t_1}^{12}[x,z]\mathbf{W}_{t_1}^{12}[x]\mathbf{R}_{t_3}^{31}[x]\mathbf{T_r}$$
$$ph_{ce} = \mathbf{T_w}\mathbf{R}_{t_2}^{21}[]\mathbf{R}_{t_1}^{11}[x]\mathbf{W}_{t_1}^{11}[y,z]\mathbf{R}_{t_1}^{12}[x,z]\mathbf{W}_{t_1}^{12}[x]\mathbf{R}_{t_3}^{31}[x]\mathbf{T_r}$$

Figure 4.6: Example showing fixed and interruptible rf-elements

$tr \in unfin(ph_c)$, *its interrupting write set is defined as*

$$IWS_{ph_c}(tr) = \{x \in Var \mid int_{ph_c}(tr, x)\}.$$

For an arbitrary candidate ph_c, the mapping of unfinished transactions to their respective interrupting write set is denoted IWS_{ph_c}. Next, we will discuss when rf-elements are fixed or interruptible and then how we can use these conditions to construct the equivalence classes. We start by giving the conditions for fixed rf-elements in p-histories and then the conditions for interruptible and fixed rf-elements in candidates.
See Figure 4.6 for the example we will use to illustrate the conditions. The reads-from relation of ph_e (and ph_{ce}) is

$$\{(tr_w, 11, x), (tr_w, 12, x), (11, 12, z), (12, 31, x), (11, tr_r, y), (11, tr_r, z), (12, tr_r, x)\}.$$

In a p-history rf-elements are fixed whenever they do not involve tr_r. The reasoning is that any extension appends p-events at the end. These new p-events are not located before any read except the one of tr_r. Thus, existing rf-elements are not "interrupted". The augmented transaction tr_r does not follow this condition as new p-events are placed before it.

Lemma 11 (Conditions for fixed rf-elements in p-histories)**.** *Given a p-history ph, two arbitrary transactions tr, tr' and an arbitrary variable x, it holds for all $(tr, tr', x) \in ph.RF$ that*

$$(tr, tr', x) \in ph.RF_{fix} \leftrightarrow tr' \neq tr_r.$$

The proof of this lemma can be found in Appendix B. In Figure 4.6 $(tr_w, 11, x)$, $(tr_w, 12, x)$, $(11, 12, z)$, and $(21, 31, x)$ are fixed rf-elements. As every element that does not involve tr_r is fixed in any p-history, there exists no interruptible rf-elements in p-histories.

RF-elements in a candidate are fixed iff they do not involve tr_r **and** the rf-element is not interruptible. Trivially, if both conditions hold, the rf-element is fixed. Now if an rf-element is fixed, it cannot be interruptible by definition, and also if it involves tr_r, it cannot be fixed as a write at the end of the candidate can overwrite the writer of the rf-element before tr_r.

Lemma 12 (Conditions for fixed rf-elements in candidates)**.** *Given a serial p-history ph_c, two arbitrary transactions tr, tr' and an arbitrary variable x, it holds for all $rf = (tr, tr', x)$ s.t. $rf \in ph_c.RF$:*

$$(tr, tr', x) \in ph_c.RF_{fix}$$
$$\leftrightarrow$$
$$\neg int_{ph_c}(rf) \wedge tr' \neq tr_r.$$

The proof of this lemma can be found in Appendix B. In the example in Figure 4.6, if 21 wrote to x, $(tr_w, 11, x)$ would be interrupted by the inserted write and be replaced with $(21, 11, x)$ in the extended candidate. Also, $(11, 12, z)$ is the only fixed rf-element in ph_{ce}.

Next, we give conditions for interruptible rf-elements in candidates. Assume an arbitrary p-history, a candidate of it, and an arbitrary rf-element (tr, tr', x) of the candidate. An rf-element is interruptible iff one of the following two conditions holds:

1. there is an unfinished transaction in between tr and tr'
2. or there exists a thread without an event after the read of tr' and in addition there exists no write after the read of tr'.

Assume the candidate is extended by an arbitrary sequence seq. If the first condition holds and seq is a write of the unfinished transaction on x, then the rf-element is interrupted. If the second condition holds and *seq* contains an arbitrary read of the thread that has no event after the read of tr', then the first condition holds in the extension, and the rf-element is interruptible in it and the original candidate. Assume both conditions are untrue, meaning there is no unfinished transaction in between tr and tr', and either there does not exist a thread without an event after the read of tr' or there is a write after the read of tr'. These conditions stay the same in any extension of the candidate as all p-events in the candidate are present in any extension of it and keep their relative order. Then, a write cannot be directly inserted in between both transactions as there is no unfinished transaction in between them. Additionally, a read cannot be inserted in between them, which would enable inserting a write, because either all threads have an event after tr', meaning the new read is inserted after, or there is a write after the read of tr', meaning the new read is inserted after it. Thus, the rf-element is not interruptible. Let $pev^{t,ls}_{ph_c}$ be the last event of t in ph_c.

Lemma 13 (Conditions for interruptible rf-elements in candidates)**.** *Given a candidate ph_c, an rf-element $(tr, tr', x) \in ph_c.RF$ with $tr' \neq tr_r$ is interruptible iff*

$$\begin{gathered}
\exists tr'' \in Tr : unfin_{ph_c}(tr'') \land pev^{tr,wr}_{ph_c} <_{ph_c} pev^{tr'',rd}_{ph_c} <_{ph_c} pev^{tr',rd}_{ph_c} \\
\lor \\
\left(\begin{gathered}
\neg(\exists tr'' \in Tr : fin_{ph_c}(tr'') \land pev^{tr',rd}_{ph_c} <_{ph_c} pev^{tr'',wr}_{ph_c}) \\
\land \\
\exists t \in T : pev^{t,ls}_{ph_c} <_{ph_c} pev^{tr',rd}_{ph_c}
\end{gathered} \right).
\end{gathered}$$

The proof of this lemma can be found in Appendix B. In Figure 4.6 in ph_{ce}, the first case applies to $(tr_w, 11, x)$ because if 21 writes to x this write is inserted directly after its read event and interrupts the rf-element. The second case applies to $(12, 31, x)$, if ph were to be extended by the p-event $\mathbf{R}_{t_1}[x]$ then one of the possible extensions of the candidate would be

$$ph'_{ce} = \mathbf{T_w}\mathbf{R}^{21}_{t_2}[]\mathbf{R}^{11}_{t_1}[x]\mathbf{W}^{11}_{t_1}[y,z]\mathbf{R}^{12}_{t_1}[x,z]\mathbf{W}^{12}_{t_1}[x]\mathbf{R}^{13}_{t_1}[x]\mathbf{R}^{31}_{t_3}[x]\mathbf{T_r}$$

where the first case applies to $(12, 31, x)$ and a further addition of a write event on x of transaction 13 would interrupt the rf-element. But if in the original candidate there is a write after $\mathbf{R}_{t_3}^{31}[x]$ or all other threads had an p-event after it, then $\mathbf{R}_{t_1}^{13}[x]$ cannot be inserted before it, making the rf-element not interruptible.

Compression In the following, we will explain how hc-pairs are grouped into equivalence classes. An equivalence class is uniquely identified by a triple of the format $\mathcal{PH} \times \mathcal{PH} \times (\mathit{Tr} \to 2^{\mathit{Var}})$ or a special symbol **DM**. The triple contains a (compressed) p-history, a (compressed) candidate and all interrupting write sets of all unfinished transactions in the candidate of the hc-pair. The special symbol **DM** summarizes all hc-pairs that are non-consistent and for which each extension is also non-consistent. We will give an algorithmic description of how to derive the equivalence class of an hc-pair. Afterwards, we give the formal definitions, which are not algorithmic but produce an equivalent result.

To explain the compression, we need to introduce the notion of *mutually exclusive* rf-elements. We call two arbitrary rf-elements $(\mathit{tr}, \mathit{tr}', x)$, $(\mathit{tr}'', \mathit{tr}', x)$ mutually exclusive iff $\mathit{tr} \neq \mathit{tr}''$. For rf-elements rf and rf' we denote this $\mathit{mutex}(\mathit{rf}, \mathit{rf}')$. This means they cannot coexist in the same reads-from relation as a transaction cannot read a variable from multiple transactions. The compression of an hc-pair $(\mathit{ph}, \mathit{ph}_c)$ is done in 3 steps.

1. Check whether one fixed rf-element in ph and one fixed element in ph_c are mutually exclusive or one fixed rf-element of the p-history and one interruptible rf-element of its candidate are mutually exclusive. If yes, put the hc-pair into the **DM** equivalence class. Skip the remaining steps.

2. Determine and save the interrupting write set for each unfinished transaction.

3. Remove all transactions for which in the p-history all rf-elements are fixed and in the candidate all rf-elements are fixed or interruptible. For each read in the p-history and the candidate, if it reads a set of variables from a removed transaction, remove all these variables from the read.

For the first step, we argue why these hc-pairs can be classified as **DM**. The first case where both rf-elements are fixed is trivial. For the second case, we shortly argue why even a change in the mutually exclusive rf-element by interruption yields an rf-element that is mutually exclusive to the rf-element in the p-history. Assume (tr, tr', x) is the fixed read in the p-history and (tr'', tr', x) the interruptible read in the candidate. If now the p-history and candidate are extended by a write of transaction tr_i which interrupts the rf-element in the candidate, the candidate then contains (tr_i, tr', x) and the p-history contains (tr, tr', x) with $tr \neq tr_i$. Thus, the rf-elements are mutually exclusive. If this read is interruptible itself, the above argument holds again, so the hc-pair is doomed. Consider the following example:

$$ph_e = \mathbf{T_w}\mathbf{R}^{31}_{t_3}[x, z]\mathbf{R}^{11}_{t_1}[x]\mathbf{R}^{21}_{t_2}[]\mathbf{W}^{11}_{t_1}[y, z]\mathbf{W}^{31}_{t_3}[x]\mathbf{T_r}$$
$$ph_{ce} = \mathbf{T_w}\mathbf{R}^{31}_{t_3}[x, z]\mathbf{W}^{31}_{t_3}[x]\mathbf{R}^{21}_{t_2}[]\mathbf{R}^{11}_{t_1}[x]\mathbf{W}^{11}_{t_1}[y, z]\mathbf{T_r}.$$

The reads-from relation of ph_e contains the fixed rf-element $(tr_w, 11, x)$. The reads-from relation of ph_{ce} contains the interruptible rf-element $(31, 11, x)$. If the write on x of 21 occurs, then the old rf-element is replaced by $(21, 11, x)$, which is still mutually exclusive to $(tr_w, 11, x)$. Also, it is fixed since no more unfinished transactions are in between the reading and writing transaction. So, in both cases, the hc-pair will stay non-consistent for any extension no matter if 21 interrupts the rf-element or not. This means the hc-pair belongs to the **DM** equivalence class.

In the second step, all interruptible rf-elements in the candidate are taken into account. As the hc-pair is not in **DM**, all of these have an identical rf-element in the p-history of the hc-pair. Whenever a transaction can interrupt an arbitrary rf-element (tr, tr', x), x is added to the interrupting write set of the transaction, if it is not already present. All extensions of the hc-pair where the transaction writes to x are not consistent. This is because the interrupting write set of a transaction is identical, no matter whether a transaction potentially can interrupt one or multiple rf-elements containing the same variable.

We use the hc-pair of Figure 4.7 as an example. Its candidate has the following

$$
\begin{aligned}
ph_e &= \mathbf{T}_{\mathbf{w}}\mathbf{R}^{11}_{t_1}[x]\mathbf{R}^{21}_{t_2}[\,]\mathbf{W}^{11}_{t_1}[z]\mathbf{T}^{31}_{t_3}[z]\mathbf{R}^{12}_{t_1}[x,z]\mathbf{W}^{12}_{t_1}[x]\mathbf{T}_{\mathbf{r}} \\
ph_{ce} &= \mathbf{T}_{\mathbf{w}}\mathbf{R}^{21}_{t_2}[\,]\mathbf{R}^{11}_{t_1}[x]\mathbf{W}^{11}_{t_1}[z]\mathbf{T}^{31}_{t_3}[z]\mathbf{R}^{12}_{t_1}[x,z]\mathbf{W}^{12}_{t_1}[x]\mathbf{T}_{\mathbf{r}}
\end{aligned}
$$

Figure 4.7: Example for equivalence class grouping.

two interruptible rf-elements:

$$(tr_w, 11, x) \text{ and } (tr_w, 12, x).$$

Thus, the interrupting write set of 21 is $\{x\}$. There are no other interruptible reads in the example. All interrupting write sets are saved.

In the third step, we first compare the fixed rf-elements of the p-history with the fixed and interruptible rf-elements of the candidate. If there exists a non-augmented transaction for which all rf-elements it occurs in are fixed in the p-history and fixed or interruptible in the candidate, we delete it in both. We also modify each read that reads a variable from such a transaction to no longer read from this variable. This effectively removes all rf-elements the transaction was occurring in. The rf-elements of such a transaction do not change in any extension of the hc-pair and thus this part of the reads-from relation is identical between the candidate and the p-history in any extension. If an rf-element is interruptible in the candidate but fixed in the p-history, we can assume it to be fixed under the condition it is not interrupted. In this case, the interrupting write set already contains this information. Because of these facts, transactions for which all rf-elements involving them are fixed in the p-history and fixed or interruptible in the candidate can be forgotten. In the example of Figure 4.7,

$$\{(tr_w, 11, x), (tr_w, 12, x), (11, 31, z), (31, 12, z), (31, tr_r, z), (12, tr_r, x)\}$$

is the reads-from order for the p-history and the candidate. For the p-history the following rf-elements are fixed:

$$(tr_w, 11, x), (tr_w, 12, x), (11, 31, z), (31, 12, z),$$

for the candidate the fixed rf-elements are

$$(11, 31, z), (31, 12, z)$$

and the interruptible rf-elements are

$$(tr_w, 11, x), (tr_w, 12, x).$$

Thus, we can remove the transaction 11. The end result would then be the tuple

$$\begin{pmatrix} \mathbf{T_w}\mathbf{R}^{21}_{t_2}[]\mathbf{R}^{31}_{t_3}[]\mathbf{W}^{31}_{t_3}[z]\mathbf{R}^{11}_{t_1}[x,z]\mathbf{W}^{11}_{t_1}[x]\mathbf{T_r}, \\ \mathbf{T_w}\mathbf{R}^{21}_{t_2}[]\mathbf{R}^{31}_{t_3}[]\mathbf{W}^{31}_{t_3}[z]\mathbf{R}^{11}_{t_1}[x,z]\mathbf{W}^{11}_{t_1}[x]\mathbf{T_r}, \\ 21 \rightarrow \{x\}. \end{pmatrix}$$

Note that the naming of the transaction is "changed" as the deletion of other transactions has changed their relative place. This only is done in examples. Identifiers such as *tr* are kept consistent in formal notation. Another hc-pair that is a member of this equivalence class for example is

$$\begin{pmatrix} \mathbf{T_w}\mathbf{R}^{11}_{t_1}[x]\mathbf{R}^{21}_{t_2}[]\mathbf{W}^{11}_{t_1}[z]\mathbf{T}^{31}_{t_3}[z]\mathbf{R}^{32}_{t_3}[x,z]\mathbf{W}^{32}_{t_3}[z]\mathbf{R}^{12}_{t_1}[x,z]\mathbf{W}^{12}_{t_1}[x]\mathbf{T_r}, \\ \mathbf{T_w}\mathbf{R}^{21}_{t_2}[]\mathbf{R}^{11}_{t_1}[x]\mathbf{W}^{11}_{t_1}[z]\mathbf{T}^{31}_{t_3}[z]\mathbf{R}^{32}_{t_3}[x,z]\mathbf{W}^{32}_{t_3}[z]\mathbf{R}^{12}_{t_1}[x,z]\mathbf{W}^{12}_{t_1}[x]\mathbf{T_r} \end{pmatrix}.$$

We are now going to formally define this compression and then discuss how to apply it to complete candidate sets. For the compression function we need a definition of the transaction set that is to be removed, a definition of a removal function for transactions and the overall compression.

Definition 26 (Fixed transactions)**.** *The set of fixed transactions in an hc-pair $hc = (ph, ph_c)$ is defined as*

$$\begin{aligned} fix(hc) = \{tr \in Tr \mid fin_{ph}(tr), \quad & \forall rf \in ph.RF : tr \in rf \rightarrow fix_{ph}(rf), \\ & \forall rf \in ph_c.RF : tr \in rf \rightarrow fix_{ph_c}(rf) \vee int_{ph_c}(rf)\}. \end{aligned}$$

Definition 27 (Removal of transactions)**.** *The transaction removal of a set of* *transactions Tr^- for a p-history (or candidate) $ph = pev_0 \dots pev_n$ is defined as*

$$ph \backslash Tr^- = pev'_0 \dots pev'_n, \text{ s.t. for } 0 \leq i \leq n :$$

$$pev'_i = \begin{cases} \epsilon & \text{if } tr_{ph}(pev) \in Tr^- \\ \mathbf{R}_t^{tr}[V'] & \text{if } pev_i = \mathbf{R}_t^{tr}[V], V' = V \backslash \{x \mid \exists tr' \in Tr^- : (tr', tr, x) \in ph.RF\} \\ pev_i & \text{else.} \end{cases}$$

To avoid convoluted index modifications, some p-events are defined as the empty word ϵ. These p-events are treated as non-existent in the resulting p-history or candidate. An important property of this removal function is that on input of a p-history and a set of transactions it removes all rf-elements involving any transaction of that set.

Lemma 14 (Removal function correctness)**.** *Given a p-history or candidate ph and a set of transactions Tr^-, it holds that*

$$(ph \backslash Tr^-).RF = ph.RF \backslash \{rf \in ph \mid \exists tr \in Tr^- : tr \in rf\}.$$

The proof of this lemma can be found in Appendix B. For the compression let $doomed(ph, ph_c)$ be true iff

$$\exists rf \in ph.RF, \exists rf' \in ph_c.RF : mutex(rf, rf') \wedge fix_{ph}(rf) \wedge (fix_{ph_c}(rf') \vee int_{ph_c}(rf')).$$

Then we can define the compression of hc-pairs.

Definition 28 (Compression of hc-pairs)**.** *The compressed hc-pair representation of an hc-pair $hc = (ph, ph_c)$ is denoted $cmp(hc)$ or $cmp(ph, ph_c)$ s.t*

$$cmp(ph, ph_c) = \begin{cases} \mathbf{DM} & \text{iff } doomed(hc) \\ (ph \backslash fix(hc), ph_c \backslash fix(hc), IWS_{ph_c}) & \text{else.} \end{cases}$$

We call such a compression consistent whenever it is not **DM** and its p-history and candidate compression are equivalent.

Lemma 15 (Upper limit of hc-pairs)**.** *The number of compressed hc-pairs for a given Var and T is finite.*

The proof of this lemma can be found in Appendix B. In the next lemma, we formalize that hc-pairs that are compressed into an equal representation are extension equivalent.

Lemma 16 (Compression represents an equivalence class)**.** *Given two arbitrary hc-pairs (ph, ph_c) and (ph', ph'_c), it holds that*

$$cmp(ph, ph_c) = cmp(ph', ph'_c) \rightarrow (ph, ph_c) \equiv_{ext} (ph', ph'_c).$$

The proof of this lemma can be found in Appendix B. Its reasoning is that given an hc-pair we can divide the reads-from relation of the p-history and the candidate into two parts, the fixed part and the non-fixed part. The fixed part is the same in all extensions of the hc-pair for both. So if one fixed rf-element of the p-history and one fixed rf-element of the candidate are mutually exclusive, they stay so in all extensions, and the hc-pair is and remains non-consistent. Assume the overlap of the fixed rf-elements of both members of the hc-pair is identical. Then for extensions only the remaining rf-elements that are not fixed are relevant for computing equivalence. Note that a transaction which is involved only in fixed rf-elements cannot be read by new transactions in an extension neither in the p-history nor the candidate. Else it would be read by tr_r which would imply the existence of a non-fixed rf-element. So all transactions with only fixed rf-elements in a p-history and candidate do not define the equivalence class for extension equivalence when the hc-pair is not doomed.

Interruptible rf-elements are behaving as fixed if they are not interrupted, and if they are interrupted, the hc-pair is doomed. The restricted write sets cover this distinction. Thus, interruptible rf-elements can be treated as fixed for the purpose of extension equivalence.

Compressing candidate sets We extend this notion to candidate sets by representing a p-history and its candidate set by a set of hc-pairs, one for each candidate. Then we apply the previous compression for each hc-pair in the set. The result is the SSR^{-}-data of a p-history. The extension equivalence notion for hc-pairs can be used to determine SSR^{-}-extension equivalence between two p-histories. Two p-histories ph, ph' are extension equivalent iff for each candidate ph_c in C_{ph} there exists an candidate ph'_c in $C_{ph'}$ such that (ph, ph_c) is extension equivalent to (ph', ph'_c). We formulate this in Lemma 17. Its proof can be found in Appendix B. Let HC_{ph} be the set of hc-pairs of a p-history and its candidate set.

Lemma 17. *Two p-histories ph, ph' are SSR^{-}-extension equivalent iff the following two conditions hold:*

1. $\forall (ph, ph_c) \in HC_{ph}, \exists (ph', ph'_c) \in HC_{ph'} : (ph, ph_c) \equiv_{ext} (ph', ph'_c)$,
2. *and* $\forall (ph', ph'_c) \in HC_{ph'}, \exists (ph, ph_c) \in HC_{ph} : (ph', ph'_c) \equiv_{ext} (ph, ph_c)$.

We define SSR^{-}-data.

Definition 29 (SSR^{-}-data)**.** *The SSR^{-}-data for an arbitrary p-history ph is defined as*

$$ssr(ph) = \{cmp(hc) \mid hc \in HC_{ph}\}.$$

If the SSR^{-}-data for two p-histories is identical, then conditions 1 and 2 of Lemma 17 are true for these p-histories. Thus, the two p-histories are SSR^{-}-extension equivalent.

Lemma 18. *Two p-histories ph, ph' are SSR^{-}-extension equivalent if $ssr(ph) = ssr(ph')$.*

The proof of this lemma can be found in Appendix B. Additionally, the number of SSR^{-}-data is finite for a given Var and T. Then the number of sets containing these classes is also finite when given these parameters as there is only a finite number of equivalence classes for hc-pairs in this case.

Automaton construction   Finally, we give the automaton construction. To formalize it, we let $SSR^-_{T,Var}$ be the set of all SSR^--data with thread identifiers from T and variables from Var. We furthermore let $SSR^-_{\emptyset\ T,Var}$ be the set of all SSR^--data containing only non-consistent compressed hc-pairs (which includes the **DM** class).

Definition 30. *Let $I = (Q, \delta, q_0, F)$ be an implementation automaton. The SSR^--*automaton of *I ($E(I)$) is the automaton $(Q_E, \delta_E, q_{0,E}, F_E)$ such that*

- $Q_E = Q \times SSR^-_{\emptyset\ T,Var}$,
- $q_{0,E} = (q_0, (ssr(\epsilon)))$,
- $F_E = F \times SSR^-_{\emptyset\ T,Var}$

and $((q, ssr), pev, (q', ssr')) \in \delta_E$ iff
$(q, pev, q') \in \delta$ *and* $\exists ph \in \mathcal{PH} : ssr(ph) = ssr \wedge ssr(ph \cdot pev) = ssr'$.

Algorithm 1 Algorithm to construct automaton

```
procedure AutConstr($I = (Q, \Sigma, \delta, q_0, F)$ )
    Init Compressed Automaton $(Q \times SSR^-_{T,Var}, \delta_E = \emptyset, (q_0, ssr(\epsilon)), F \times SSR^-_{\emptyset\ T,Var})$
    Init Queue $P$ with $((q_0, \epsilon), (q_0, ssr(\epsilon)))$
    $visited = \emptyset$
    while $P$ is not empty do
        $((q, ph), q_E) = POP(P)$
        for $q'$ s.t. $(q, pev, q') \in \delta$ do
            $\delta_E = \delta_E \cup \{(q_E, pev, (q', ssr(ph \cdot pev))\}$
            if $(q', ssr(ph \cdot pev)) \notin visited$ then
                $visited = visited \cup (q', ssr(ph \cdot pev))$
                Add $((q', ph \cdot pev), (q', ssr(ph \cdot pev)))$ to $P$
    return $(Q_E, \delta_E, q_{0,E}, F_E)$
```

The constructed automaton is a finite automaton since we only have finitely many different SSR^--data.

We can derive strict serializability of the implementation automaton from the language of the SSR^--automaton.

Theorem 3. *Let I be an implementation automaton. Then I only produces strictly serializable p-histories iff $L(E(I)) = \emptyset$.*

The proof can be found in Appendix B. For decidability of the overall problem it is required that this automaton is constructable. This can be done using Algorithm 1. As all possible states of the compressed automaton (and thus also all final states) are known in advance for a given input because of Lemma 15, the algorithm only needs to add all edges. It does so by performing a modified BFS on the naive state space using a queue containing pairs of naive states and their corresponding compressed automaton states. The queue is initialized with a pair containing the starting states of the naive state space and the compressed automaton, respectively. When a pair of naive and compressed state is taken from the queue, each successor of the naive state is iterated through. In each iteration, first, an edge between the compression of the naive state taken from the queue and the compression of its successor state is added to the compressed automaton. Second, the pair containing the successor state and its compression is added to the queue if the compression was not part of any pair that was already in the queue. This condition ensures that the algorithm terminates. This is because $Q \times SSR^{-}_{T,Var}$ is finite. This means Line 11 is executed a finite number of times. This in turn makes the while loop in Line 5 terminate after a finite number of steps. That the algorithm generates all edges can be shown by applying the following lemma, which follows from a part of the proof of Lemma 16.

Lemma 19. *Given two arbitrary p-histories ph and ph', it holds that*

$$\forall seq \in PEv^* : ssr(ph) = ssr(ph') \rightarrow ssr(ph \cdot seq) = ssr(ph' \cdot seq).$$

The proof for this lemma can be found in Appendix B. It shows that the extensions of two p-histories with identical SSR^{-}-data again have identical SSR^{-}-data; thus, the algorithm only needs to explore the successor states of one of them.
This finally gives us the decidability of SSR^{-}.

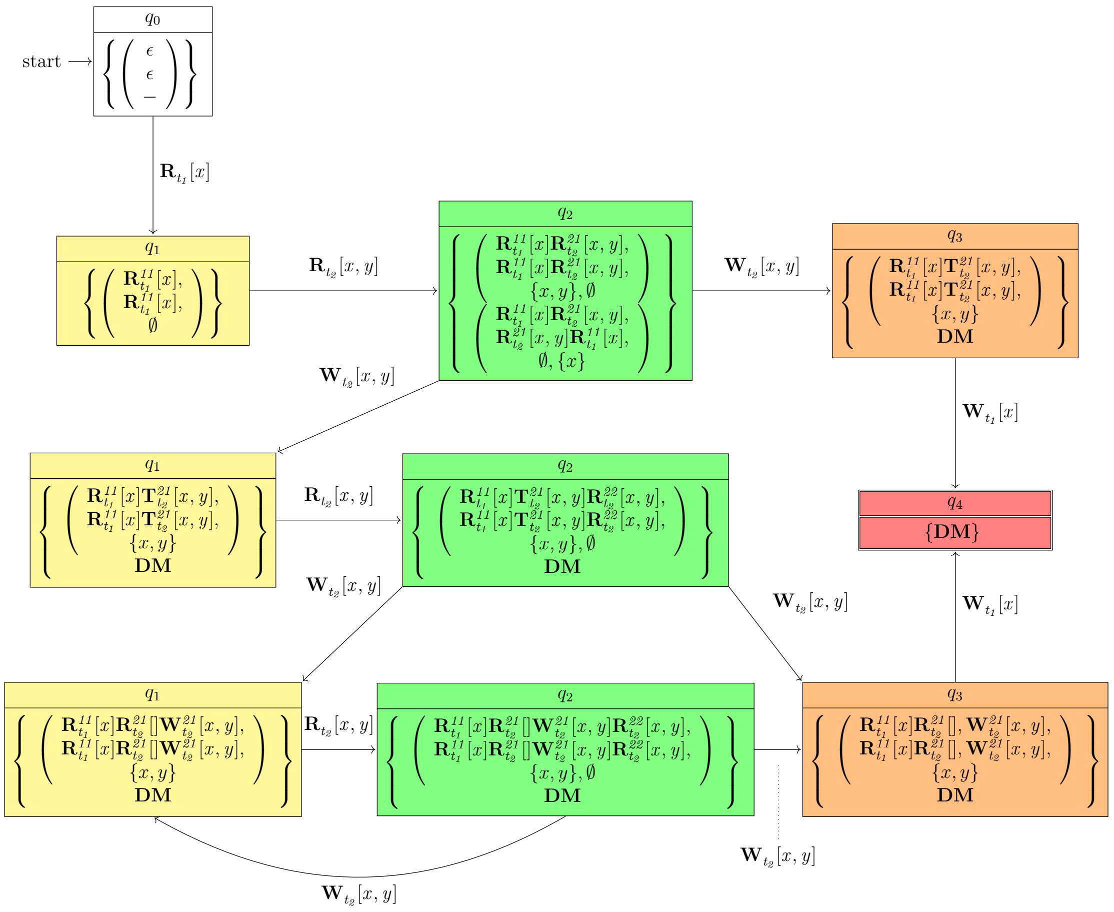


Figure 4.8: *SSR^-*-automaton of I_{ex} (of Figure 4.1), augmented transactions not shown, RWS ordered by transaction ID

Theorem 4 (Decidability of the correctness problem for SSR^-)**.** *The correctness problem for SSR^- is decidable.*

Figure 4.8 shows the result of the construction for our running example. The diagram only depicts the reachable states. Note that the standardized naming of transactions can lead to a "renaming" of transactions, and it does so for transaction 3 in one case. We see that the language of the SSR^--automaton is non-empty (the red state is accepting), and hence not all p-histories of the implementation automaton are strictly serializable. We also see that equivalence of SSR^--data only implies SSR^--extension equivalence. There are still two green and two yellow states which are SSR^--extension equivalent but have different SSR^--data, and thus could not be compacted to a single state.

4.3 The Correctness Problem for OP^- Is Decidable

In this section, we prove the decidability of a restricted version of the correctness problem for OP. In this version, for each implementation each read is justifiable by a write of a previous transaction, and it is possible to identify the writer for each read unambiguously (similar to an unambiguous value-based reads-from relation). We realize the first point with the *reasonable read assumption* and the second point with the *timeout assumption* and the *thread ID values assumption*. We describe and define these assumptions here, a discussion of why these assumptions were chosen can be found in Section 5.2.

The reasonable read assumption restricts reads to only read values from writing transactions which have not been overwritten by another writing transaction in between the read and the original writing transaction.

Assumption 5 (Reasonable read assumption (RR-assumption))**.** *A read on variable x by transaction tr' may only read a value val s.t.*

- *there exists a transaction tr with a write event, writing val to x,*
- *the transaction tr is either committed or commit pending and the commit invoke has happened before the read event*
- *and there is no other committed transaction real-time ordered in between tr and tr' writing on x.*

This assumption ensures that no values from the "future" can be read by the implementation. When only considering the subset of histories generated by an implementation with this restriction, value opacity is prefix-closed. This is because otherwise a scenario such as shown in Figure 4.9 is possible.
This g-history is opaque and adheres to constraint 1 and 2 of the RR-assumption. Consider the prefix of this g-history where transaction 4 is not committed. This

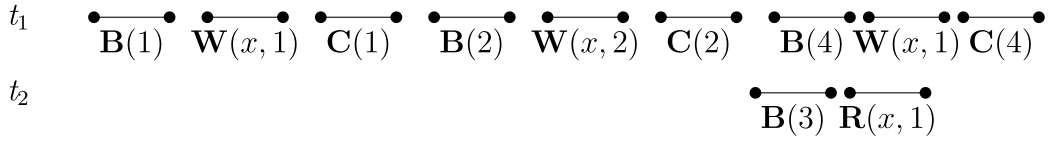


Figure 4.9: Example for the necessity of the third condition of the RR-assumption

prefix also adheres to constraint 1 and 2 of the RR-assumption, but it is not value opaque. So the third condition is necessary to prevent reads being justified by a future write.

Next we describe the assumptions necessary for enabling identifying the writer for each read unambiguously. The timeout assumption is that there is an upper bound to the length of all transactions.

Assumption 6 (Timeout assumption)**.** *There is a fixed upper bound of steps in which a transaction is finished.*

This also implies there is a fixed number of transactions of one thread that can be concurrent to any transaction. Let this number be *to*. Now we can make a timestamp assumption only using finitely many values.

Assumption 7 (Thread ID values assumption (TIV-assumption))**.** *For each write to a variable var of a transaction tr of thread t the value val consists of three fields containing the following information:*

- *the actual content to be written,*
- *t*
- *and a timestamp equal to the number of committed transactions of t writing to var up until tr taken mod to* $+ 1$*.*

The combination of these two assumptions ensures that there are only finitely many transactions that any read can possibly read from. This means each concurrent history has a well-defined reads-from relation where each read has one corresponding write.

In the following, we also exclude the case $|T| = 1$ because it would require several case distinctions in the proofs for this section. If this holds, each implementation is trivially correct because every g-history is serial and legal. Thus, the answer to the correctness problem is always "yes". Under all previous assumptions, we call OP OP^-. The structure of the proof and thus also this section is similar to the structure for the SSR^- problem. (Section 4.2)

Compact representation As in Section 4.2, we try to compress the naive state space of the given implementation by merging nodes which

- contain the same implementation automaton state
- and contain g-histories s.t. after appending identical g-events to both, either both resulting g-histories are opaque or both are not.

We start by formalizing the above similarity on g-histories. As before, we define extension equivalence for OP^-.

Definition 31 (OP^--extension equivalence)**.** *Two g-histories $h, h' \in \mathcal{H}$ are OP^--extension equivalent ($h \equiv_{ext} h'$) iff $\forall seq \in Ev^*$ either*

- *$h \cdot seq$ and $h' \cdot seq$ are both value opaque under OP^-*
- *or $h \cdot seq$ and $h' \cdot seq$ are both not value opaque under OP^-.*

Then, as before, using the equivalence classes implied by this notion, we are able to reduce naive state space automata. To determine whether two g-histories are OP^--extension equivalent, we use a similar concept as with SSR^-. We reduce a g-history to the necessary information to determine whether appending g-events keeps the g-history value opaque or not. We call this information OP^--data. If this data is identical for two g-histories, they are OP^--extension equivalent. The remainder of this section is structured similar to Section 4.2. We first adopt the notion of candidates for value opacity, and then present how to determine the value opacity of a g-history from its candidates. Then we show how to use this notion for compressing g-histories and their candidate sets to OP^--data. Using this data, we finally present the automata reduction construction.

Candidate set The candidates for an input g-history are all serial g-histories that are equivalent to it and preserve its real-time order. Note that these candidates, unlike their SSR^- counterparts, are not potential OP^--witnesses. They are subsequences of OP^--witnesses containing only the events of the original (not completed) g-history. As we will later see, each such subsequence of an OP^--witness is also a candidate. We can use this to determine whether a g-history is value opaque by checking the candidate set. But as with SSR^-, we first present candidates, discuss the supersequence property and give an insertion function to derive the candidates for an extension of a g-history. We start with the definition of candidate sets.

Definition 32 (Candidate set)**.** *The candidate set for a given input g-history h of OP^-, C_h, is the set of all serial g-histories h_c which*

- *are equivalent to h*
- *and preserve its real-time order.*

Similar to SSR^-, these candidate sets have a supersequence property.

Proposition 3 (Supersequence property)**.** *Given a g-history h and an arbitrary extension h' of it, it holds that*

$$\forall h'_c \in C_{h'}, \exists h_c \in C_h : h_c \sqsubseteq h'_c.$$

The reason of why this property holds is similar to the one for SSR^-. We will argue why this is the case for the extension by one g-event ev. This implies the property for arbitrary extensions by induction. For this explanation, let h be an arbitrary g-history, and h' be an extension of it. Let $\mathbf{T}_t^{tr}(var, val, val')$ denote a committed transaction tr of thread t writing val to var and reading val' from the same variable. We use the example from Figure 4.10 to illustrate the argument.

$$h'_e = \mathbf{T}^{11}_{t_1}(y, 0, 1)\mathbf{B}^{21}_{t_2}\mathbf{B}^{12}_{t_1} \quad \rightarrow \quad h_e = \mathbf{T}^{11}_{t_1}(y, 0, 1)\mathbf{B}^{21}_{t_2}$$

$$C_{h'_e} = \left\{ \begin{array}{l} \mathbf{T}^{11}_{t_1}(y, 0, 1)\mathbf{B}^{21}_{t_2}\mathbf{B}^{12}_{t_1} \\ \mathbf{T}^{11}_{t_1}(y, 0, 1)\mathbf{B}^{12}_{t_1}\mathbf{B}^{21}_{t_2} \end{array} \right\} \rightarrow C_{h_e} = \left\{ \ \mathbf{T}^{11}_{t_1}(y, 0, 1)\mathbf{B}^{21}_{t_2} \ \right\}$$

Figure 4.10: Example of the supersequence property for candidate sets

First, note that the real-time order of h' consists of the real-time order of h unified with the set containing all rt-elements induced by ev in h', denoted $RT_{h'}(ev)$. This is formalized as

$$h'.RT = h.RT \cup RT_{h'}(ev).$$

The set $RT_{h'}(ev)$ is non-empty iff ev is a begin. In the example, h'_e has the real-time order of h_e plus the rt-elements involving 12, namely $(11, 12)$. By definition, any arbitrary candidate of h', denoted h'_c, preserves its real-time order:

$$h.RT \cup RT_{h'}(ev) \subseteq h'_c.RT.$$

Let h'_{c-} be h'_c with ev removed. Its real-time order is the one of h'_c with all rt-elements induced by ev in h'_c, denoted $RT_{h'_c}(ev)$, removed:

$$h.RT \cup RT_{h'}(ev) \subseteq h'_{c-}.RT \cup RT_{h'_c}(ev).$$

Any rt-element in $RT_{h'_c}(ev)$ is either in $RT_{h'}(ev)$ or not contained in $h'.RT$ (and thus also not in $h.RT$) at all as in both g-histories all transactions share the same events and event order. Thus, it holds that

$$h.RT \subseteq h'_{c-}.RT.$$

As h'_{c-} is trivially serial, it is a candidate for h. In the example, for the candidate $\mathbf{T}^{11}_{t_1}[y]\mathbf{B}^{12}_{t_1}\mathbf{B}^{21}_{t_2}$ becomes $\mathbf{T}^{11}_{t_1}[y]\mathbf{B}^{21}_{t_2}$ which is a candidate of h_e.

Next, we define the insertion function which shows how the extension of candidates is derived. Before that, we need to briefly define additional needed notation.

Definition 33 (Notation)**.** *Let h be an arbitrary g-history. In the following we* *define additional notation.*

- *Let $st(h)$ be the subsequence of h starting at its first g-event, and ending at the last g-event that is either a commit or abort.*
- *Let $en(h)$ be the subsequence of h including all g-events after the last g-event that is either a commit or abort.*
- *Let $add(h, ev, n)$ return h with ev inserted at the end of h, if n is its last index, or in between indices n and $n + 1$, if not.*
- *Let $ev_h^{tr,ls}$ be the last event of tr in h.*
- *Let TrI_h be the set of all indices in h at which the last g-event of a transaction is located.*

Then the insertion function is a combination of two functions, one for the insertion of a begin event and one for the insertion of any other event.

Definition 34 (Insertion function begin)**.** *The insertion function for begin events is denoted $ins_b : \mathcal{H} \times Ev \to 2^{\mathcal{H}}$. On input of a serial g-history h_c and a begin g-event ev, it returns the set*

$$\{st(h_c) \cdot add(en(h_c), ev, n) \mid n \in TrI_{en(h_c)}\}.$$

Definition 35 (Insertion function write/read/commit/abort)**.** *The insertion function for write, abort, read and commit events is denoted $ins_{warc} : \mathcal{H} \times Ev \to 2^{\mathcal{H}}$. On input of a serial g-history $h_c = ev_0 \ldots ev_{h_c}^{tr,ls} \ldots ev_n$ and a write, abort, read, or commit g-event ev of transaction tr, it returns the set*

$$\{ev_0 \ldots ev_{h_c}^{tr,ls} ev \ldots ev_n\}.$$

Definition 36 (Insertion function)**.** *The insertion function is denoted $ins : \mathcal{H} \times Ev \to 2^{\mathcal{H}}$. On input of a serial g-history h_c and a g-event ev, it returns a set of g-histories which is either*

$$\begin{tikzcd}
h_e = \mathbf{B}^{21}_{t_2}\mathbf{B}^{11}_{t_1}\mathbf{W}^{21}_{t_2}(x,5) \arrow[r, "\mathbf{W}^{11}_{t_1}(z,1)"'] & h'_e = \mathbf{B}^{21}_{t_2}\mathbf{B}^{11}_{t_1}\mathbf{W}^{21}_{t_2}(x,5)\mathbf{W}^{11}_{t_1}(z,1)
\end{tikzcd}$$

$$\begin{tikzcd}
 & h'_{e,c_1} = \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(z,1)\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(x,5) \\
h_{e,c} = \mathbf{B}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(x,5) \arrow[ur, "insert\ \mathbf{W}^{11}_{t_1}(z,1)"] \arrow[dr, "insert\ \mathbf{W}^{11}_{t_1}(z,1)"'] & \\
 & \xcancel{h'_{e,c_2} = \mathbf{B}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(x,5)\mathbf{W}^{11}_{t_1}(z,1)}
\end{tikzcd}$$

Figure 4.11: A g-history and its candidates extended with a write event

$$\begin{tikzcd}
h = \mathbf{B}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(x,5)\mathbf{W}^{11}_{t_1}(x,3)\mathbf{B}^{12}_{t_1} \arrow[r, "\mathbf{B}^{22}_{t_2}"] & h' = \mathbf{B}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(x,5)\mathbf{W}^{11}_{t_1}(x,3)\mathbf{B}^{12}_{t_1}\mathbf{B}^{22}_{t_2}
\end{tikzcd}$$

$$\begin{tikzcd}
 & h'_{s_1} = \mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(x,5)\mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(x,3)\mathbf{B}^{12}_{t_1}\mathbf{B}^{22}_{t_2} \\
h_s = \mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(x,5)\mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(x,3)\mathbf{B}^{12}_{t_1} \arrow[ur, "insert\ \mathbf{B}^{22}_{t_2}"] \arrow[r, "insert\ \mathbf{B}^{22}_{t_2}"] \arrow[dr, "insert\ \mathbf{B}^{22}_{t_2}"] & h'_{s_2} = \mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(x,5)\mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(x,3)\mathbf{B}^{22}_{t_2}\mathbf{B}^{12}_{t_1} \\
 & \xcancel{h'_{s_3} = \mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(x,5)\mathbf{B}^{22}_{t_2}\mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(x,3)\mathbf{B}^{12}_{t_1}}
\end{tikzcd}$$

Figure 4.12: A g-history and its candidates extended with a begin event

1. *$ins_b(h_c, ev)$, if ev is a begin*

2. *or $ins_{warc}(h_c, ev)$, else.*

This function conceptually is identical to the one for SSR^- (see Definition 20). Any event that does not start a new transaction must be appended directly at the end of its transaction, or else the resulting g-history is not serial. Any starting event of a transaction (a begin in this case) must be appended after the last (finishing) event (a commit or abort in this case) of a transaction to preserve the real-time order of the extended concurrent g-history. The two cases are shown in figures 4.11 and 4.12.

As we have stated above, inserting a g-event in such a way into a candidate of a g-history yields a candidate of this g-history extended by that g-event. Thus, the insertion function returns a subset of the candidate set of the extended g-history.

Proposition 4 (Generation of candidates by insertion function)**.** *Given a g-history h and a g-event ev, it holds that*

$$\bigcup_{h_c \in C_h} ins(h_c, ev) = C_{h \cdot ev}.$$

The proof of this proposition can be found in Appendix C. We also extend *ins* to arbitrary g-event sequences by applying the function for the first event to the input candidate, then apply the function to each candidate in the resulting set, then merge the resulting sets and repeat the last two steps for the remaining g-events of the sequence.

Determining value opacity from candidates In this section, we will discuss how we can use the candidates of a g-history to check whether a legal OP^--witness for it exists. We show that iff the set of all serial completions (completions that are a serial g-history) of all candidates for a g-history contains a legal g-history, then there exists an OP^--witness for this g-history. We present a method to efficiently determine whether a legal serial completion exists by (a) giving a simplified method for determining the legality of a serial completion and (b) showing that for determining the existence of a legal minimal completion it also suffices to check a certain serial completion (called *minimal completion*) of each candidate for legality.

We start by showing that if there exists an OP^--witness for a g-history, then the set of all serial completions of all candidates for a g-history contains a legal g-history. For a set of g-histories H', let $SC(H')$ be the set of all serial completions of g-histories in H'.

Lemma 20. *Given a g-history h, it holds that*

$$\exists h_s \in \mathcal{H} : h_s \in OW(h) \rightarrow \exists h_{c,comp} \in \mathcal{H} : h_{c,comp} \in SC(C_h) \land h_{c,comp} \text{ is legal.}$$

Concurrent history:

$$h_1 = \mathbf{B}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{11}_{t_1}(x, val)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})\mathbf{R}^{21}_{t_2}(x, val)$$

Completions:

$$\begin{aligned}
h_2 &= \mathbf{B}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{11}_{t_1}(x, val)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})\mathbf{A}^{11}_{t_1}\mathbf{R}^{21}_{t_2}(x, val)\mathbf{A}^{21}_{t_2}\\
h_3 &= \mathbf{B}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{11}_{t_1}(x, val)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})\mathbf{Resp}^{11}_{t_1}(\mathbf{C})\mathbf{R}^{21}_{t_2}(x, val)\mathbf{A}^{21}_{t_2}\\
h_4 &= \mathbf{B}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{11}_{t_1}(x, val)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})\mathbf{R}^{21}_{t_2}(x, val)\mathbf{A}^{11}_{t_1}\mathbf{A}^{21}_{t_2}\\
h_5 &= \mathbf{B}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{11}_{t_1}(x, val)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})\mathbf{R}^{21}_{t_2}(x, val)\mathbf{A}^{21}_{t_2}\mathbf{A}^{11}_{t_1}\\
h_6 &= \mathbf{B}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{11}_{t_1}(x, val)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})\mathbf{R}^{21}_{t_2}(x, val)\mathbf{Resp}^{11}_{t_1}(\mathbf{C})\mathbf{A}^{21}_{t_2}\\
h_7 &= \mathbf{B}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{11}_{t_1}(x, val)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})\mathbf{R}^{21}_{t_2}(x, val)\mathbf{A}^{21}_{t_2}\mathbf{Resp}^{11}_{t_1}(\mathbf{C})
\end{aligned}$$

Equivalent serial g-histories

$$\begin{aligned}
h_{s,1} &= \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(x, val)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})\mathbf{Resp}^{11}_{t_1}(\mathbf{C})\mathbf{B}^{21}_{t_2}\mathbf{R}^{21}_{t_2}(x, val)\mathbf{A}^{21}_{t_2}\\
h_{s,2} &= \mathbf{B}^{21}_{t_2}\mathbf{R}^{21}_{t_2}(x, val)\mathbf{A}^{21}_{t_2}\mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(x, val)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})\mathbf{Resp}^{11}_{t_1}(\mathbf{C})\\
h_{s,3} &= \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(x, val)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})\mathbf{A}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{R}^{21}_{t_2}(x, val)\mathbf{A}^{21}_{t_2}\\
h_{s,4} &= \mathbf{B}^{21}_{t_2}\mathbf{R}^{21}_{t_2}(x, val)\mathbf{A}^{21}_{t_2}\mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(x, val)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})\mathbf{A}^{11}_{t_1}
\end{aligned}$$

Candidates:

$$\begin{aligned}
h_{c,1} &= \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(x, val)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})\mathbf{B}^{21}_{t_2}\mathbf{R}^{21}_{t_2}(x, val)\\
h_{c,2} &= \mathbf{B}^{21}_{t_2}\mathbf{R}^{21}_{t_2}(x, val)\mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(x, val)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})
\end{aligned}$$

Minimal completions:

$$\begin{aligned}
h_{mc,1} &= \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(x, val)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})\mathbf{Resp}^{11}_{t_1}(\mathbf{C})\mathbf{B}^{21}_{t_2}\mathbf{R}^{21}_{t_2}(x, val)\mathbf{A}^{21}_{t_2}\\
h_{mc,2} &= \mathbf{B}^{21}_{t_2}\mathbf{R}^{21}_{t_2}(x, val)\mathbf{A}^{21}_{t_2}\mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(x, val)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})\mathbf{Resp}^{11}_{t_1}(\mathbf{C})
\end{aligned}$$

Figure 4.13: Example histories with corresponding completions, equivalent histories, candidates and minimal completions for value opacity

The proof can be found in Appendix C. We explain intuitively why this is the case, given a g-history h and one of its witnesses h_s. Consider a g-history h_c which is h_s without all g-events not occurring in h. We show it is a candidate for h. A candidate must be equivalent to h and preserve its real-time order. For it to be equivalent to h, it must contain the same transactions and the view of the history for each thread must be identical to h. The g-history h_c contains the same elements as h; thus, it also contains the same transactions. Also, h_s has the same order as h for all events in the same thread that are contained in both:

$$\forall ev, ev' \in h : thr(tr_h(ev)) = thr(tr_h(ev')) \rightarrow (ev <_{h_s} ev' \leftrightarrow ev <_h ev').$$

As h_c has the same order for all events in the same thread as h_s for all events contained in h, it has the same order for all events in the same thread as h :

$$\forall ev, ev' \in h : thr(tr_h(ev)) = thr(tr_h(ev')) \rightarrow (ev <_h ev' \leftrightarrow ev <_{h_c} ev').$$

Thus, the view of the history for each thread is identical to h. Because of this, it is equivalent to h. Secondly, a candidate must preserve the real-time order of h. It is given that h_s preserves the real-time order of h meaning $h.RT \subseteq h_s.RT$. Thus, if two given transactions tr and tr' are real-time ordered in h, they are as well in h_s. As h_c and h_s have the same order for all g-events and the commit or abort event of tr and the begin event of tr' are present in h, this implies $tr' \prec_{h_c} tr$ as well:

$$\forall tr, tr' \in Tr : tr' \prec_h tr \rightarrow tr' \prec_{h_s} tr \rightarrow tr' \prec_{h_c} tr.$$

Note that by construction, h_s is one of the serial completions of h_c.

As h_s is legal, this then means that itself is legal serial completion of h_c and thus of a candidate of h, proving the lemma. That means any OP^--witness of a g-history is thus the result of a serial completion of one of its candidates. Consider the example shown in Figure 4.13. The subsequence of the only witness $h_{s,1}$ just containing the elements of h_1 is the candidate $h_{c,1}$. Thus, $h_{s,1}$ is the serial completion of $h_{c,1}$.

We show the other direction. If the set of all serial completions of all candidates for a g-history contains a legal g-history, then there exists an OP^--witness for this g-history.

Lemma 21. *Given a g-history h, it holds that*

$$\exists h_{c,comp} \in \mathcal{H} : h_{c,comp} \in SC(C_h) \wedge h_{c,comp} \textit{ is legal } \rightarrow \exists h_s \in \mathcal{H} : h_s \in OW(h).$$

The proof of this lemma can be found in Appendix C. We give an intuitive description here. Given a candidate h_c and a completion $h_{c,comp}$ of it, if we can show $h_{c,comp}$ is equivalent to one completion of h and preserves the real-time order of h, the lemma is proven as in combination with its legality $h_{c,comp}$ is then a witness of h. Consider the completion h_{comp} of h where all transactions are completed by the same events as they are in $h_{c,comp}$. We first show that h_{comp} and $h_{c,comp}$ are equivalent. They trivially contain the same events as they contain all events of h plus the same finishing events. Thus, they also contain the same transactions.

The order of events for each thread is identical as it is identical to h for all events occurring in h, and all finishing events are at the end of their transactions/thread. We show that $h_{c,comp}$ preserves the real-time order of h. This holds because h_c already preserves the real-time order of h. By adding additional commits or aborts, only new rt-elements are generated. Thus, the serial completion preserves the real-time order of h as well. This proves the lemma.

Using both lemmas, we can state that an OP^--witness only exists for a g-history iff a serial completion of one of its candidates is legal.

Lemma 22. *Given a g-history h, it holds that*

$$\exists h_s \in \mathcal{H} : h_s \in OW(h) \leftrightarrow \exists h_c \in \mathcal{H} : h_c \in SC(C_h) \wedge h_c \textit{ is legal.}$$

This follows directly from lemmas 20 and 21.

Next, we give an efficient method for determining the legality of a serial completion. This method employs a reads-from relation for the completion similar to SR/SSR . This reads-from relation is compared with the reads-from relation implied by the values of the g-history. We can determine this relation unambiguously because of the assumptions made for OP^-. To ensure each read has a corresponding write, we add tr_w to the transaction pool which is an initial transaction writing each initial value on each variable. W.l.o.g. we assume each g-history to contain this transaction and that it is real-time ordered before all other transactions. It is not shown in examples. We first define the *conflict reads-from relation* for serial g-histories, and then the *value reads-from relation* for concurrent g-histories.

Definition 37 (Conflict reads-from relation)**.** *The conflict reads-from relation for a serial g-history h_s, where all transactions are finished, is defined as $h.RF_c \subseteq Tr \times Tr \times Var$. A tuple (tr, tr', var) is in $h_s.RF_c$ iff*

- $var \in WS_h^{vo}(tr)$,
- $var \in RS_h^{vo}(tr')$,
- $tr \prec_{h_s} tr'$,

- $co_{h_s}(tr)$
- *and* $\neg(\exists tr'' \in Tr : var \in WS^{vo}_{h_s}(tr'') \wedge co_{h_s}(tr'') \wedge tr \prec_{h_s} tr'' \prec_{h_s} tr')$.

In Figure 4.13 the conflict reads-from relation of witness $h_{s,1}$ has one element, namely $(11, 21, x)$. Next we define the value reads-from relation.

Definition 38 (Value reads-from relation)**.** *The value reads-from relation for a g-history h is defined as $h.RF_{val} \subseteq Tr \times Tr \times Var$. A tuple (tr, tr', var) is in $h.RF_{val}$ iff there exists $val \in Val$ s.t.*

- $(var, val) \in WS_h(tr)$,
- $(var, val) \in RS_h(tr')$,
- $\mathbf{Inv}^{tr}_{thr(tr)}(\mathbf{C}) <_h \mathbf{R}^{tr'}_{thr(tr')}(var, val)$
- *and* $\neg(\exists tr'' \in Tr : var \in WS^{vo}_{h}(tr'') \wedge co_{h}(tr'') \wedge tr \prec_{h} tr'' \prec_{h} tr')$.

In the example, the value reads-from relation of h has one element, namely $(11, 21, x)$.

A serially completed candidate is an OP^--witness for a g-history whenever its conflict reads-from relation is identical to the value reads-from relation of the g-history.

Lemma 23 (Legal)**.** *Given a g-history h and a serial completion h_s of one of its candidates, it holds that*

$$h_s \text{ is legal } \leftrightarrow h_s.RF_c = h.RF_{val}.$$

This lemma follows from the original definition of legality and three assumptions made at the beginning of this section. The proof can be found in Appendix C. In the example, $h_{mc,1}$ is legal as its conflict reads-from relation is identical to the value reads-from relation of h. It is thus an OP^--witness for h. This can be seen by the fact it is identical to $h_{s,1}$. There exists an rf-element for every read in both relations because of the existence of tr_w. Thus, the reads-from relations for a g-history and a candidate are identical iff there exists no two *mutually exclusive* rf-elements in them. We call two arbitrary rf-elements (tr, tr', x), (tr'', tr', x) mutually exclusive iff $tr \neq tr''$. For rf-elements rf and rf', we denote this $mutex(rf, rf')$.

Now having an efficient way to check for legality, we give a method to determine whether a legal serial completion exists for a candidate without checking all serial completions of that candidate. We can reduce the number of completions of candidates that need to be checked by checking only one completion, called *minimal completion*. A candidate has a legal completion iff its minimal completion is legal. We prove this fact below. Before that, we describe how minimal completions are defined and argue why this property holds.

We first need to define the *visibility* of a transaction in a g-history. Visibility expresses that a transaction has been read by another transaction, and thus it must be committed in any witness justifying the opacity of the original g-history.

Definition 39 (Visibility of a transaction). *We say tr is visible in a g-history h, denoted $vis_h(tr)$, iff there exists $tr' \in Tr$ and $var \in Var$ s.t.*

$$(tr, tr', var) \in h.RF_{val}.$$

In the example in Figure 4.13, transaction 11 is visible in h_1. Now we can define the minimal completion of a candidate. It only exists in the context of the g-history it is a candidate of. In a minimal completion all commit pending transactions that are visible in its g-history are committed, and all other commit pending transactions are aborted.

Definition 40 (Minimal completion). *The minimal completion of a candidate h_c of g-history h, denoted $mCl_h(h_c)$, is the serial g-history h_{mc} s.t.*

- $h_{mc} \in compl(h_c)$,
- $comP_{h_c}(tr) \wedge vis_h(tr) \rightarrow co_{h_{mc}}(tr)$,
- *and* $comP_{h_c}(tr) \wedge \neg(vis_h(tr)) \rightarrow ab_{h_{mc}}(tr)$.

If it is clear which g-history h a candidate h_c belongs to, $mCl_h(h_c)$ is shortened to $mCl(h_c)$. In the example, $h_{mc,1}$ and $h_{mc,2}$ are minimal completions of candidates $h_{c,1}$ and $h_{c,2}$, respectively. As we will only talk about the conflict reads-from relation of a serial completion of a candidate and its own reads-from relation is not relevant, from now on we will instead simply refer to it as the conflict reads-from relation of the candidate. Thus, for a candidate h_c of a g-history h, we denote $mCl(h_c).RF_c$ by $h_c.RF_c$ if it is clear that h_c is a candidate of h from context.

Now to determine opacity using minimal completions we proceed as follows. Instead of checking each completion, we check for each candidate whether its minimal completion is legal. We prove that iff the minimal completion of a candidate of a g-history is legal, then there exists a legal completion for that candidate. This combined with Lemma 22 then leads to the fact that a g-history has an OP^--witness iff one of its candidates has a legal minimal completion. We start with the first claim.

Lemma 24 (Minimal completion legal iff legal completion exists). *Given a g-history h and a candidate h_c, a legal serial completion for h_c exists iff $mCl(h_c)$ is legal.*

The proof can be found in Appendix C. Let h be an arbitrary g-history and h_c be an arbitrary candidate of it. It is obvious that a legal minimal completion of h_c implies a legal serial completion (the minimal completion itself) exists for h_c. For the other direction, we argue that if a serial completion $h_{c,comp}$ of h_c that is not the minimal completion is legal, then $mCl(h_c)$ is also legal. We do this by showing that each rf-element is identical between the serial completion and minimal completion. Assume a transaction tr reads the value val from x from another transaction tr' in the serial completion. The transaction tr must be committed in $h_{c,comp}$ as else the rf-element would not exist. As the completion is legal, tr must be visible in h. Thus, tr must be committed in $mCl(h_c)$. In $h_{c,comp}$ there cannot exist another committed transaction tr'' writing on x in between tr and tr'. In the case there simply exists no transaction with x in its write set in between tr and tr', this is clearly also the case in $mCl(h_c)$. In the case there exists an aborted transaction with x in its write set in between tr and tr', this transaction cannot be visible as else $h_{c,comp}$ would not be legal. Thus, this transaction is also aborted in $mCl(h_c)$. Consider this simple example of a serial completion in which h_c is a candidate, h_{sc} is a serial completion of the candidate and h_{mc} is the minimal completion.

$$\begin{aligned}
h_c &= \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(x,1)\mathbf{Inv}^{11}_{t_1}(\mathbf{C})\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(x,2)\mathbf{C}^{21}_{t_2}\mathbf{B}^{22}_{t_2}\mathbf{R}^{22}_{t_2}(x,2)\\
h_{sc} &= \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(x,1)\mathbf{C}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(x,2)\mathbf{C}^{21}_{t_2}\mathbf{B}^{22}_{t_2}\mathbf{R}^{22}_{t_2}(x,2)\\
h_{mc} &= \mathbf{B}^{11}_{t_1}\mathbf{W}^{11}_{t_1}(x,1)\mathbf{A}^{11}_{t_1}\mathbf{B}^{21}_{t_2}\mathbf{W}^{21}_{t_2}(x,2)\mathbf{C}^{21}_{t_2}\mathbf{B}^{22}_{t_2}\mathbf{R}^{22}_{t_2}(x,2).
\end{aligned}$$

It is easy to see that whether 11 is committed or aborted is irrelevant for the legality of any serial completion. If a legal minimal completion exists for any candidate of a g-history, then an OP^--witness exists for that g-history, too. This is formalized by the following lemma.

Lemma 25. *Given an input g-history h of OP^-, an OP^--witness for h exists iff there exists candidate h_c of h s.t. $mCl(h_c)$ is legal.*

The proof can be found in Appendix C. Conceptually this holds because of two facts. As implied by Lemma 20, each witness is a serial completion of a candidate. Secondly, as implied by Lemma 24, for a single candidate a legal minimal completion exists iff there is a legal serial completion. These legal completions are OP^-witnesses. Thus, there exists such a minimal completion for one candidate iff it exists for the g-history.

Compression We use a similar strategy as before in Section 4.2. It again utilizes hc-pairs. In comparison, this approach has slight differences in when transactions can be forgotten, and thus also by what data an equivalence class is characterized. An extension of an hc-pair means the extension of both its members. An hc-pair is called consistent iff its candidate's minimal completion is legal. We group multiple hc-pairs into an equivalence class if they share a broken down notion of extension equivalence. This notion is defined as follows.

Definition 41 (Extension equivalence for hc-pairs)**.** *Given g-histories h, h' and candidates $h_c \in C_h$ and $h'_c \in C_{h'}$, the hc-pair (h, h_c) is extension equivalent to* *hc-pair* (h', h'_c)*, denoted as* $(h, h_c) \equiv_{ext} (h', h'_c)$*, iff for all g-event sequences seq*

$$\exists h_{c2} \in ins(h_c, seq) : mCl(h_{c2}) \textit{ is legal} \leftrightarrow \exists h'_{c2} \in ins(h'_c, seq) : mCl(h'_{c2}) \textit{ is legal.}$$

We can approximate the equivalence classes for this notion. Our approximation uses the fact that certain rf-elements of certain transactions in an hc-pair are fixed in the value reads-from relation of the concurrent g-history and in the conflict reads-from relation of the minimal completion of the candidate, respectively, for any extension of the hc-pair. We first define fixed rf-elements for a concurrent g-history and their candidates.

Definition 42 (Fixed rf-elements for g-histories)**.** *An rf-element* $(tr, tr', x) \in h.RF_{val}$ *in a g-history* h *is called fixed iff*

$$\forall seq \in Ev^* : (tr, tr', x) \in (h \cdot seq).RF_{val}.$$

For a candidate this is defined as follows.

Definition 43 (Fixed rf-elements for candidates)**.** *Given a candidate* h_c *of g-history* h*, an rf-element* $(tr, tr', x) \in h.RF_c$ *is called fixed iff*

$$\forall seq \in Ev^* \forall h'_c \in ins(h_c, seq) : (tr, tr', x) \in h'_c.RF_c.$$

Let $h.RF^{fix}_{val}/h.RF^{fix}_{c}$ be the set of fixed rf-elements for the value reads-from relation of a g-history or the conflict reads-from relation of a minimal completion of a candidate, respectively.

A major difference to the relations for *SSR* is that there is no all reading transaction at the end of a g-history in OP^-. Thus, rf-elements are always fixed in the concurrent g-history, leading to the following lemma.

Lemma 26 (Conditions for fixed rf-elements in g-histories)**.** *Given a g-history* h*, it holds that* $h.RF^{fix}_{val} = h.RF_{val}$.

The proof for this can be found in Appendix C. The proof follows straightforward from the definition of the value reads-from relation and our assumptions for OP^-.

To give conditions for fixed rf-elements in candidates, one additional definition is needed. But first note that defining interruptible rf-elements as in the *SSR* construction is pointless. This is because each rf-element is interruptible iff it is not fixed and not abortable (the concept is explained below) as there is no all reading transaction. We still need a definition of an interrupting write set for the compression. In this chapter, we will give that definition right before the compression construction.

The new necessary definition for the conditions is the *abortable read*. An abortable read in a candidate is an rf-element for which the writing transaction is commit pending and may abort in extensions of the candidate, removing the rf-element. Thus, if in an extension the writing transactions aborts, the rf-element is not present in the conflict reads-from relation anymore.

Definition 44 (Abortable read). *Given a candidate h_c of g-history h, an rf-element*
$(tr, tr', x) \in h_c.RF_c$ with $tr' \neq tr_r$ is called abortable iff $comP_{h_c}(tr)$ holds.

Given this definition, we can give the conditions for fixed rf-elements in candidates. There are four cases in which an rf-element (tr, tr', x) can be removed in an extension of a candidate. First, tr can be commit pending and then abort in the extension. Second, there can be an unfinished transaction, which is not commit pending, in between tr and tr', and in an extension it can write to x and commit. Third, there can be a commit pending transaction, having x in its write set, which commits or becomes visible by being read by a later transaction. Fourth, there can be a thread without an event after the last event of tr', and neither tr' nor any other transaction after it has finished. Then this thread can start a transaction inserted between tr and tr' making the previous 2 conditions possible. Now if all of these conditions do not hold, then

1. no new transaction can be inserted in between tr and tr', no existing transaction can write to x in between tr and tr',

2. no commit pending transaction that intends to write to x can become visible or commit as it does not exist

3. and tr cannot abort.

This makes the rf-element fixed. All of these conditions do not change in any extension as no events are removed and their relative order stays identical. Let $ev_{h_c}^{rd,rf}$ be the read event belonging to the rf-element rf. Let $ev_{h_c}^{ls,t}$ be the last event of t in h_c.

Lemma 27 (Conditions for fixed rf-elements in candidates)**.** *Given a candidate h_c of g-history h and an arbitrary rf-element $rf \in h_c.RF$ with $rf = (tr, tr', var)$, rf is in $h_c.RF_c^{fix}$ iff*

1. rf is not abortable,

2. $\neg(\exists tr'' \in Tr : unfin_{h_c}(tr'') \wedge \neg(comP_{h_c}(tr'')) \wedge tr \prec_{h_c} tr'' \wedge \neg(tr' \prec_{h_c} tr''))$,

3. $\neg(\exists tr'' \in Tr : comP_{h_c}(tr'') \wedge x \in WS_{h_c}(tr) \wedge tr \prec_{h_c} tr'' \wedge \neg(tr' \prec_{h_c} tr''))$

4. and $(\exists tr'' : fin_{h_c}(tr'') \wedge ev_{h_c}^{rd,rf} <_{h_c} ev_{h_c}^{tr'',ls}) \vee (\forall t : ev_{h_c}^{rd,rf} <_{h_c} ev_{h_c}^{ls,t})$.

The proof for this can be found in Appendix C.

$$\begin{aligned}
h &= \mathbf{B}_{t_1}^{11}\mathbf{W}_{t_1}^{11}(x,1)\mathbf{C}_{t_1}^{11}\mathbf{B}_{t_1}^{12}\mathbf{B}_{t_2}^{21}\mathbf{W}_{t_1}^{12}(x,3)\mathbf{Inv}_{t_1}^{12}(\mathbf{C})\mathbf{R}_{t_2}^{21}(x,1)\mathbf{W}_{t_2}^{21}(z,1)\mathbf{Inv}_{t_2}^{21}(\mathbf{C})\mathbf{B}_{t_1}^{13}\mathbf{R}_{t_1}^{13}(z,1)\mathbf{W}_{t_1}^{13}(x,4)\mathbf{C}_{t_1}^{13}\\
h_c &= \mathbf{B}_{t_1}^{11}\mathbf{W}_{t_1}^{11}(x,1)\mathbf{C}_{t_1}^{11}\mathbf{B}_{t_1}^{12}\mathbf{W}_{t_1}^{12}(x,3)\mathbf{Inv}_{t_1}^{12}(\mathbf{C})\mathbf{B}_{t_2}^{21}\mathbf{R}_{t_2}^{21}(x,1)\mathbf{W}_{t_2}^{21}(z,1)\mathbf{Inv}_{t_2}^{21}(\mathbf{C})\mathbf{B}_{t_1}^{13}\mathbf{R}_{t_1}^{13}(z,1)\mathbf{W}_{t_1}^{13}(x,4)\mathbf{C}_{t_1}^{13}\\
h_{mc} &= \mathbf{B}_{t_1}^{11}\mathbf{W}_{t_1}^{11}(x,1)\mathbf{C}_{t_1}^{11}\mathbf{B}_{t_1}^{12}\mathbf{W}_{t_1}^{12}(x,3)\mathbf{A}_{t_1}^{12}\mathbf{B}_{t_2}^{21}\mathbf{R}_{t_2}^{21}(x,1)\mathbf{W}_{t_2}^{21}(z,1)\mathbf{C}_{t_2}^{21}\mathbf{B}_{t_1}^{13}\mathbf{R}_{t_1}^{13}(z,1)\mathbf{W}_{t_1}^{13}(x,4)\mathbf{C}_{t_1}^{13}
\end{aligned}$$

Figure 4.14: Example showing interruptible and abortable rf-elements

We will use a running example shown in Figure 4.14 to demonstrate the above lemma and the following definitions. In the minimal completion of candidate h_c there is only the fixed rf-element $(11, 21, x)$. For example, $(21, 13, z)$ is not fixed as 12 is abortable.

Before giving the description of the reduction construction for OP^-, we will give descriptions and definitions for the interrupting write set of a transaction and for a must-commit transaction. Consider an hc-pair (h, h_c). A variable x is called an interrupting write for an unfinished transaction tr'' whenever there exists an rf-element (tr, tr', x) in $h_c.RF_c$ s.t. tr'' is real-time ordered in between tr and tr' in $mCl(h_c)$, aborted in $mCl(h_c)$ and is either not commit pending or commit pending and has var in its write set. This is denoted $int_{h_c}(tr'', x)$. We also call tr'' interrupting for rf in the above case. Similar as with SSR^-, the interrupting write set of a transaction is the set of the variables such that if a transaction were to write on one or more of them, and then commit, it would interrupt an rf-element.

Definition 45 (Interrupting write set)**.** *Given a candidate h_c of g-history h and a transaction $tr \in unfin(h_c)$, which is not committed in $mCl(h_c)$, its interrupting write set is defined as*

$$IWS_{h_c}(tr) = \{x \in Var \mid int_{h_c}(tr, x)\}.$$

If a transaction tr is the writing transaction of an abortable rf-element in a candidate h_c, we say it is a must-commit transaction and denote it $mc_{h_c}(tr)$.

Definition 46 (Must-Commit transactions)**.** *Given a candidate h_c of g-history h, its must-commit transactions are the following set*

$$MC_{h_c} = \{tr \in Tr \mid mc_{h_c}(tr)\}.$$

In Figure 4.14, the must-commit transactions for h_c are $MC_{h_c} = \{21\}$ as this transaction is part of one abortable rf-element. In the following, we will explain how hc-pairs are grouped into equivalence classes. An equivalence class is uniquely identified by a quadruple of the format $\mathcal{H} \times \mathcal{H} \times (Tr \to 2^{Var}) \times 2^{Tr}$ or a special symbol **DM**. The former contains a compressed g-history, a compressed candidate, all interrupting write sets of all unfinished transactions of the candidate and all must-commit transactions of the candidate. The special symbol **DM** summarizes all hc-pairs that are non-consistent. This also implies that each extension is also non-consistent.

Now we will give the algorithmic description of how to derive the equivalence class of an hc-pair. Afterwards, we give the formal definitions. These are not algorithmic, but produce an equivalent result. The compression of an hc-pair (h, h_c) is done in 4 steps:

1. Check whether there are two rf-elements - one in the value reads-from relation of the g-history and one in the conflict reads-from relation - that are mutually exclusive. If this is the case, put the hc-pair into the **DM** equivalence class.

2. Determine and save the interrupting write set for each unfinished transaction in the candidate.

3. Determine and save the must-commit transactions for the candidate.

4. Do the following for all transactions which are not the last finished transaction in the candidate: Remove all aborted transactions. Remove all committed transactions and every read event reading them for which two things hold:

    (a) In the g-history and the candidate, all variables of their write set have been overwritten by committed transactions, which are real-time ordered after them

    (b) and none of these transactions is the most recent write on a variable in the candidate for an unfinished transaction that is not abort pending or commit pending.

We explain these steps for an arbitrary hc-pair (h, h_c). If a candidate is not legal in the first step, then no extension of the hc-pair can be consistent. We will argue why.

If the candidate is not legal, there exist two mutually exclusive rf-elements $\mathit{rf} = (\mathit{tr}', \mathit{tr}, x)$ and $rf_c = (\mathit{tr}'', \mathit{tr}, x)$ s.t. $x \in \mathit{Var}$ and $\mathit{tr}' \neq \mathit{tr}''$. It holds that rf is in the reads-from relation of the g-history and rf_c is in the reads-from relation of the candidate. The rf-element rf is fixed as shown in Lemma 26. We do a case distinction over rf_c. If rf_c is also fixed, the hc-pair is obviously not consistent when extended by any g-event sequence. We will now argue if rf_c is an interruptible or abortable rf-element, the same fact holds. If it is interruptible, it can only be interrupted by a commit of a transaction that is not tr'. This is because tr' is already finished in the minimal completion of the candidate as it is visible in the g-history. In an extension where it is interrupted, there will be a new rf-element which is mutually exclusive to rf. Thus, in an extension, the hc-pair stays non-consistent. If the new rf-element is also interruptible, by the same argument the hc-pair is non-consistent in extensions. If in an extension all interrupting transactions are finished, rf_c becomes fixed. This also makes such an extension non-consistent. If rf_c is abortable and mutually exclusive to rf, we must first note that tr'' is also read by another transaction than tr in h because else it would be aborted in the minimal completion of h_c. Thus, there exists another rf-element $(tr'', tr''', z) \in h.RF_{val}$. Then if tr'' aborts, $(tr'', tr''', z) \notin h_c.RF_c$, and also it is not in any extension of h_c as the transaction is finished. This makes any extension of the hc-pair non-consistent.

In the second step, the interrupting write set for each unfinished transaction is saved. One transaction can potentially interrupt an arbitrary number of rf-elements with a committed write to one variable. But as soon as one of them is interrupted in the extension of a legal candidate, this extension and any extension of it is not legal. This can be used for compression. For an example of that, consider two given candidates for a given history and a transaction present in both. If in both candidates there exists at least one interruptible rf-element which the transaction interrupts with a committed write on one variable, these candidates both are not legal for any extension where the transaction writes to that variable and commits. In Figure 4.14, there is one interruptible rf-element in h_c, namely $(11, 21, x)$. Thus, the only non-empty interrupting write set is the one of 21 which is $\{x\}$.

In the third step, each commit pending transaction that is a must-commit transaction is saved. A commit pending transaction may be read by arbitrarily many transactions. If there is at least 1 such transaction, and the commit pending transaction aborts in an extension of the candidate, this extension is not legal and in the **DM** equivalence class. This can be used for compression as given two candidates with an identical transaction that is a must-commit transaction, both of them are not legal in any extension in which that transaction is aborted. In Figure 4.14, h_c has 1 abortable rf-element namely $(21, 13, z)$. The must-commit transaction set thus is the set $\{21\}$.

In the last step, certain transactions and g-events are removed. All transactions that cannot be read in the g-history (given the assumptions of OP^{-}), and which cannot be read in the candidate by new read events anymore, are removed. Also, all read events reading from these transactions are removed in the g-history and candidate. The only exception is the last finished transaction of the candidate, even if it cannot be read (if it is aborted or has an empty write set), as its placement is relevant for the insertion of new begin events in the candidate. In any extension of the hc-pair, there can be no new rf-elements involving the removed transactions. If the hc-pair is consistent, these rf-element stay identical between g-history and candidate for any extension of the hc-pair. Thus, these extensions are legal iff the new rf-elements are also identical between the reads-from relations of g-history and candidate. If the candidate is not legal, the hc-pair has already been put into the **DM** equivalence class in step 1.

In Figure 4.14, transaction 11 has the transaction 13 real-time ordered after it in h. Both only write to x and commit. The same holds in the candidate h_c. Additionally, all unfinished transactions are commit pending, and thus they cannot read transaction 11. It is thus removed, and the tuple characterizing the equivalence class of the hc-pair of the example is as follows:

$$\begin{pmatrix} \mathbf{B}_{t_1}^{12}\mathbf{B}_{t_2}^{21}\mathbf{W}_{t_1}^{12}(x,3)\mathbf{Inv}_{t_1}^{12}(\mathbf{C})\mathbf{R}_{t_2}^{21}(x,1)\mathbf{W}_{t_2}^{21}(z,1)\mathbf{Inv}_{t_2}^{21}(\mathbf{C})\mathbf{B}_{t_1}^{13}\mathbf{R}_{t_1}^{13}(z,1)\mathbf{W}_{t_1}^{13}(x,4)\mathbf{C}_{t_1}^{13}, \\ \mathbf{B}_{t_1}^{12}\mathbf{W}_{t_1}^{12}(x,3)\mathbf{Inv}_{t_1}^{12}(\mathbf{C})\mathbf{B}_{t_2}^{21}\mathbf{R}_{t_2}^{21}(x,1)\mathbf{W}_{t_2}^{21}(z,1)\mathbf{Inv}_{t_2}^{21}(\mathbf{C})\mathbf{B}_{t_1}^{13}\mathbf{R}_{t_1}^{13}(z,1)\mathbf{W}_{t_1}^{13}(x,4)\mathbf{C}_{t_1}^{13}, \\ \{x\} \\ \{21\} \end{pmatrix}.$$

Note that the color of the interrupting write set indicates which transaction it is of. Another member of this equivalence class would be the hc-pair:

$$\begin{pmatrix} \mathbf{B}_{t_1}^{11}\mathbf{W}_{t_1}^{11}(x,1)\mathbf{C}_{t_1}^{11}\mathbf{B}_{t_1}^{12}\mathbf{B}_{t_2}^{21}\mathbf{W}_{t_1}^{12}(x,3)\mathbf{Inv}_{t_1}^{12}(\mathbf{C})\mathbf{R}_{t_2}^{21}(x,1)\mathbf{W}_{t_2}^{21}(z,1)\mathbf{Inv}_{t_2}^{21}(\mathbf{C})\mathbf{B}_{t_1}^{13}\mathbf{R}_{t_1}^{13}(x,1)\mathbf{A}_{t_1}^{13}\mathbf{B}_{t_1}^{14}\mathbf{R}_{t_1}^{14}(z,1)\mathbf{W}_{t_1}^{14}(x,4)\mathbf{C}_{t_1}^{14}, \\ \mathbf{B}_{t_1}^{11}\mathbf{W}_{t_1}^{11}(x,1)\mathbf{C}_{t_1}^{11}\mathbf{B}_{t_1}^{12}\mathbf{W}_{t_1}^{12}(x,3)\mathbf{Inv}_{t_1}^{12}(\mathbf{C})\mathbf{B}_{t_2}^{21}\mathbf{R}_{t_2}^{21}(x,1)\mathbf{W}_{t_2}^{21}(z,1)\mathbf{Inv}_{t_2}^{21}(\mathbf{C})\mathbf{B}_{t_1}^{13}\mathbf{R}_{t_1}^{13}(x,1)\mathbf{A}_{t_1}^{13}\mathbf{B}_{t_1}^{14}\mathbf{R}_{t_1}^{14}(z,1)\mathbf{W}_{t_1}^{14}(x,4)\mathbf{C}_{t_1}^{14} \end{pmatrix},$$

where the aborted transaction 13 is also removed. Also, here there are two interruptible rf-elements $(11, 21, x)$ and $(11, 13, x)$ in the candidate. Despite this, the interrupting write sets stay identical as transaction 12 would interrupt both rf-elements with a committed write on x.

We are now going to formally define this compression, and then discuss how to apply it to complete candidate sets. First, we need to define unreadable transactions for g-histories and for candidates. In a g-history or a candidate, a transaction is unreadable whenever any new read event in any arbitrary extension of the g-history cannot read any variable from that transaction. If a transaction tr occurs in an rf-element rf, i.e. it is either the first or second member of the triple, we say $tr \in rf$. So for a g-history an unreadable transaction is defined as follows:

Definition 47 (Unreadable transactions in a g-history). *For a g-history h a finished transaction $tr \in h$ is unreadable denoted $tr \in ur(h)$ iff there exist no sequence seq in which there exists a read $\mathbf{R}^{tr'}_{thr(tr')}(var, val)$ s.t.*

$$(tr, tr', var) \in (h \cdot seq).RF_{val}.$$

Next we give conditions such that a transaction is unreadable iff these conditions are given in a g-history.

Lemma 28 (Conditions for unreadable transactions in a g-history). *For a g-history h, a finished transaction $tr \in h$ is unreadable iff*

1. $ab_h(tr)$
2. *or* $\forall var \in WS^{vo}_h(tr), \exists tr' \in Tr : co_h(tr') \wedge var \in WS^{vo}_h(tr') \wedge tr \prec_h tr'$.

The proof can be found in Appendix C. It is trivial to see that if a transaction is aborted, it cannot be read. If for a transaction each variable of its write set is "overwritten" by some other committed transaction real-time ordered after it, it cannot be read any more by new reads appended at the end given the assumptions of OP^-.

Now, for candidates the definition of unreadable is basically identical except that the insertion function is used instead of appending the event sequence and the value reads-from relation is replaced by its conflict counterpart.

Definition 48 (Unreadable transactions in a candidate). *For a g-history h_c, a finished transaction $tr \in h_c$ is unreadable denoted $tr \in ur(h_c)$ iff there exist no sequence seq in which there exists a read $\mathbf{R}^{tr'}_{thr(tr')}(var, val)$ s.t.*

$$\forall h'_c \in ins(h_c, seq) : (tr, tr', var) \notin h'_c.RF_c.$$

We again give conditions such that a transaction is unreadable iff these conditions are given in a candidate.

Lemma 29 (Conditions for unreadable transactions in a candidate)**.** *Given a finished transaction tr occurring in candidate h_c of g-history h, tr is unreadable iff*

1. $ab_{h_c}(tr)$ *(Aborted)*
2. *or* $co_{h_c}(tr)$ *and:*
   - $\forall var \in WS^{vo}_{h_c}(tr), \exists tr' \in Tr :$
     $co_{h_c}(tr') \land var \in WS^{vo}_{h_c}(tr') \land tr \prec_{h_c} tr'$ *(Overwritten before end)*
   - *and for all unfinished and not commit pending or abort pending transactions tr' in h_c:*
     (a) $\neg(tr \prec_{h_c} tr')$ *(Ordered after unfin. tr.)*
     (b) *or* $\forall var \in WS^{vo}_{h_c}(tr), \exists tr'' \in Tr :$
     $co_{h_c}(tr'') \land var \in WS^{vo}_{h_c}(tr'') \land tr \prec_{h_c} tr'' \prec_{h_c} tr'$ *(Overwr. bef. unfin. tr.).*

The proof of this lemma can be found in Appendix C. While these conditions look much more complicated than the conditions for unreadable transactions in g-histories, the concept is similar. A transaction cannot be read when it is aborted. Additionally, each committed transaction must be overwritten before the point at which new g-events may be inserted. In this case, this point is not only the end of the candidate but at the end (or the start as a candidate is serial) of each unfinished transaction as well. This means, given an unreadable transaction and an unfinished transaction, the unreadable transaction must either be overwritten before the unfinished transaction or it must be ordered afterwards.

The following lemma is the key to the compression being able to remove unreadable transactions and to model their effects on the g-history with the interrupting write sets and must-commit transaction set.

Lemma 30. *Given an unreadable transaction tr in a candidate h_c in an hc-pair (h, h_c) and an arbitrary extension of it $(h \cdot seq, h'_c)$ by a sequence seq, the following property holds:*

$$\begin{aligned}&\{rf \in h_c.RF_c \mid tr \in rf\} \neq \{rf \in h'_c.RF_c \mid tr \in rf\}\\ &\to \forall seq' \in Ev^*, \forall h''_c \in ins(h'_c, seq') : (h \cdot seq \cdot seq', h''_c.RF_c) \textit{ is not consistent.}\end{aligned}$$

The proof can be found in Appendix C. Now we can define the set of unreadable transactions for an hc-pair. It is the intersection of the unreadable transactions of its members. We take the intersection instead of the union as we still need to detect whenever a new transaction reads differently in the g-history and the candidate. We also define the last finished transaction in the candidate h_c as $tr_{lf}(h_c)$, and remove it from the intersection for the reasons discussed previously.

Lemma 31 (Unreadable transaction set of an hc-pair)**.** *The set of unreadable transactions of an hc-pair $hc = (h, h_c)$ is defined as*

$$ur(hc) = (ur(h) \cap ur(h_c)) \backslash tr_{lf}(h_c).$$

Definition 49 (Removal of transactions)**.** *The transaction removal of a set of transactions Tr^- for a g-history (or candidate) $h = ev_0 \dots ev_n$ is defined as*

$$h \backslash Tr^- = ev'_0 \dots ev'_n, \textit{ s.t. for } 0 \leq i \leq n :$$

$$ev'_i = \begin{cases} \epsilon & \textit{if } tr_h(ev) \in Tr^- \\ \epsilon & \textit{if } ev_i = \mathbf{R}^{tr}_t(x, val), \exists tr' \in Tr^- : (tr', tr, x) \in h.RF_{val/c} \\ ev_i & \textit{else.} \end{cases}$$

For this definition, $RF_{val/c}$ denotes that according to whether h is a history or candidate the respective reads-from relation is used. To avoid convoluted index modifications, some g-events are defined as the empty word ϵ. These p-events are treated as non-existent in the resulting g-history or candidate. An important property of this removal function is that when applied to both members of an hc-pair with a set of transactions, it removes all rf-elements from the g-history and candidate in which any transaction of the transaction set is involved in.

Lemma 32 (Removal function correctness hc-pairs)**.** *Given an hc-pair (h, h_c) and a set of transactions Tr^-, it holds for $(h \backslash Tr^-, h_c \backslash Tr^-)$ that*

1. $(h \backslash Tr^-).RF_{val} = h.RF_{val} \backslash \{rf \in h.RF_{val} \mid \exists tr \in Tr^- : tr \in rf\}$
2. *and* $(h_c \backslash Tr^-).RF_c = h.RF_c \backslash \{rf \in h_c.RF_c \mid \exists tr \in Tr^- : tr \in rf\}$.

The proof can be found in Appendix C. Using this, we can define the compression of hc-pairs. The compression removes all unreadable transactions and read events reading them, except the last finished transaction, and stores the relevant information of them in the interrupting write sets and the set of must-commit transactions. For this definition, we need to define the set containing all interrupting write sets of a candidate h_c, which is

$$IWS_{h_c} = \{IWS_{h_c}(tr) \mid tr \in h_c\}.$$

Definition 50 (Compression of hc-pairs)**.** *The compressed hc-pair representation of an hc-pair $hc = (h, h_c)$, denoted $cmp(h, h_c)$, is defined as*

$$cmp(h, h_c) = \begin{cases} \mathbf{DM} & iff h_c.RF_c \neq h.RF_{val} \\ (h \backslash ur(hc), h_c \backslash ur(hc), IWS_{h_c}, MC_{h_c}) & else. \end{cases}$$

Lemma 33 (Finite number of hc-pairs)**.** *The number of compressed hc-pairs is finite for a given T, Var and Val.*

The proof can be found in the appendix in Appendix C. The next lemma formalizes that hc-pairs that are compressed into an equal representation are extension equivalent.

Lemma 34 (Compression represents an equivalence class). *Given two arbitrary hc-pairs (h, h_c) and (h', h'_c), it holds that*

$$cmp(h, h_c) = cmp(h', h'_c) \rightarrow (h, h_c) \equiv_{ext} (h', h'_c).$$

The proof can be found in Appendix C. Its reasoning is that given an hc-pair we can divide the reads-from relation of the g-history and the candidate into two parts, the fixed part and the non-fixed part. The fixed part is the same in all extensions of the hc-pair for both. Now whenever the candidate is non-legal, meaning there is a difference in reads-from relations, the hc-pair is in **DM** as we have discussed in the description above. This is a difference to the SSR^- construction as in it there can be mutually exclusive rf-elements involving tr_r in an hc-pair, but extensions of it can still be consistent. If the overlap of the fixed rf-elements of both members of the hc-pair is identical, then for extensions only the remaining rf-elements that are not fixed are relevant for computing equivalence. Note that a transaction is only removed iff it cannot be read in any extension of the hc-pair, neither in the g-history nor in the candidate. So, all transactions only involved in fixed rf-elements in a g-history and candidate do not define the equivalence class for extension equivalence when the hc-pair is not doomed. Interruptible rf-elements are behaving as fixed if they are not interrupted and if they are interrupted the hc-pair is doomed. The restricted write sets cover this distinction; thus interruptible rf-elements can be treated as fixed for the purpose of extension equivalence. The same can be applied for abortable rf-elements.

Compressing candidate sets We extend this notion to candidate sets by representing a g-history and its candidate set by a set of hc-pairs, one for each candidate. Then we apply the previous compression for each hc-pair in the set. The result of this is the OP^--data of a g-history. The extension equivalence notion for hc-pairs can be used to determine OP^--extension equivalence between two g-histories. Two g-histories h, h' are OP^--extension equivalent iff for each candidate h_c in C_h there exists a candidate h'_c in $C_{h'}$ such that (h, h_c) is extension equivalent to (h', h'_c). We formulate this in Lemma 35. Let HC_h be the set of hc-pairs of a g-history and its candidate set.

Lemma 35. *Two g-histories h, h' are OP^--extension equivalent iff the following two conditions hold:*

1. $\forall (h, h_c) \in HC_h, \exists (h', h'_c) \in HC_{h'} : (h, h_c) \equiv_{ext} (h', h'_c)$,
2. $\forall (h', h'_c) \in HC_{h'}, \exists (h, h_c) \in HC_h : (h', h'_c) \equiv_{ext} (h, h_c)$.

The proof of this lemma can be found in Appendix C. We define OP^--data.

Definition 51 (OP^--data)**.** *The OP^--data for an arbitrary g-history h is defined as*

$$op(h) = \{cmp(hc) \mid hc \in HC_h\}.$$

If the OP^--data for two g-histories is identical, then conditions 1 and 2 of Lemma 35 are true for these g-histories. Thus, the two g-histories are OP^--extension equivalent.

Lemma 36. *Two g-histories h, h' are OP^--extension equivalent if $op(h) = op(h')$.*

The proof can be found in Appendix C.

Automaton construction Finally, we give the automaton construction. To formalize it, we let $OP^-_{T,Var}$ be the set of all OP^--data with thread identifiers from T and variables from Var. We furthermore let ${OP^-_{\emptyset}}_{T,Var}$ be the set of all OP^--data of the format $\{\mathbf{DM}\}$.

Definition 52. *Let $I = (Q, \delta, q_0, F)$ be an implementation automaton. Then the* OP^--automaton of *I $(E(I))$ is the automaton $(Q_E, \delta_E, q_{0,E}, F_E)$ such that*

- $Q_E = Q \times OP^-_{T,Var}$,

- $q_{0,E} = (q_0, (\epsilon, \emptyset))$,
- $F_E = F \times OP^{-}_{\emptyset\ T,Var}$

and $((q, op), ev, (q', op')) \in \delta_E$ *iff*
$(q, ev, q') \in \delta$ *and* $\exists h \in \mathcal{H} : cmp(h) = op \wedge cmp(h \cdot ev) = op'$.

Algorithm 2 Algorithm to construct automaton

```
1: procedure AutConstr($I = (Q, \Sigma, \delta, q_0, F)$ )
2:     Init Compressed Automaton ($Q \times OP^{-}_{T,Var}, \delta_E = \emptyset, (q_0, op(\epsilon)), F \times OP^{-}_{\emptyset\ T,Var}$)
3:     Init Queue $P$ with $((q_0, \epsilon), (q_0, op(\epsilon)))$
4:     $visited = \emptyset$
5:     while $P$ is not empty do
6:         $((q, h), q_E) = POP(P)$
7:         for $q'$ s.t. $(q, ev, q') \in \delta$ do
8:             $\delta_E = \delta_E \cup \{(q_E, ev, (q', op(h \cdot ev))\}$
9:             if $(q', op(h \cdot ev)) \notin visited$ then
10:                $visited = visited \cup (q', op(h \cdot ev))$
11:                Add $((q', h \cdot ev), (q', op(h \cdot ev)))$ to $P$
12:    return $(Q_E, \delta_E, q_{0,E}, F_E)$
```

The automaton is a finite automaton since we only have finitely many different valid OP^{-}-data. We can derive opacity of the implementation automaton from the language of the OP^{-}-automaton.

Theorem 5. *Let I be an implementation automaton. Then I only produces g-histories opaque under OP^{-} iff $L(E(I)) = \emptyset$.*

The proof can be found in Appendix C. For the decidability of the overall problem, it is required that this automaton is constructable. This is possible via the algorithm shown in Algorithm 2. As all possible states of the compressed automaton (and thus also all final states) are known in advance for a given input because of Lemma 33, the algorithm only needs to add all edges. It does so by performing a modified BFS on the naive automaton using a queue containing pairs of naive states and their corresponding compressed automaton states. The queue is initialized with a pair containing the starting states of the naive and the compressed automaton, respectively. When a pair of naive and compressed state is taken from the queue, each successor of the naive state is iterated through. In each iteration, first, an edge between the compression of the naive state taken from the queue and the compression of its successor state is added to the compressed automaton. Second, the pair containing the successor state and its compression is added to the queue if the compression was not part of any pair that was already in the queue. This condition ensures that the algorithm terminates. This is because $Q \times OP^{-}_{T,Var}$ is finite, which means Line 11 is executed a finite number of times, which in turn makes the while loop in Line 5 terminate after a finite number of steps. That the algorithm generates all edges can be shown by applying the following lemma which follows from a part of the proof of Lemma 34.

Lemma 37. *Given two arbitrary g-histories h and h', it holds that*

$$\forall seq \in Ev^* : op(h) = op(h') \rightarrow op(h \cdot seq) = op(h' \cdot seq).$$

The proof for this lemma can be found in Appendix C. It implies that the extensions of two p-histories with identical OP^{-}-data again have identical OP^{-}-data; thus, the algorithm only needs to explore the successor states of one of them. This finally gives us the decidability of OP^{-}.

Theorem 6 (Decidability of the correctness problem for OP^{-})**.** *The correctness problem for OP^{-} is decidable.*

5 Discussion + Conclusion

In this chapter, we will first give a summary of the results of this thesis, discuss these results in detail (including future work) and then give our concluding thoughts.

5.1 Summary

The main goal of this thesis was to fill gaps in the existing literature with regard to the complexity of the membership and correctness problem for correctness conditions used in the context of TMs. Additionally, we aimed to provide an overview of the existing results. The thesis is divided into two parts: one concerning the membership problem and one concerning the correctness problem.

In the first part of the thesis, we gave an overview of the existing results (see Table 5.1) and presented results for two questions in this field. We showed that the membership problem for value opacity is NP-complete, and we compared the languages of conflict opacity and value opacity under two sets of assumptions.

For the first result (Section 3.2) we showed that the membership problem for value opacity is at least as hard as the membership problem for state serializability (which is NP-complete) by using a reduction from the latter problem to the former

Condition	Complexity of membership problem
Sequential consistency	NP-complete [40]
Linearizability	NP-complete [96, 40]
View serializability	NP-complete [97]
State serializability	NP-complete [73]
Strict state serializability	NP-complete [88]
Conflict serializability	P [73]
Causal serializability	Unknown
Snapshot isolation	NP-complete [13]
Value Opacity	NP-complete (Section 3.2)
Conflict opacity	P
DU opacity	Unknown
TMS 1	Unknown
TMS 2	Unknown
VWC	Unknown

Table 5.1: Complexity of the membership problems for different correctness conditions. Our contribution shown in green

problem. Then, we showed that the membership problem for value opacity belongs to the NP complexity class. The reduction converts a history that adheres to the syntax of histories state serializability is defined upon onto a history of the syntax of histories value opacity is defined upon. This was done in such a way that the constraints of state serializability are equivalently modelled by the constraints of value opacity in the converted history. Then we showed the membership problem for value opacity is in the NP complexity class by showing that it is possible to determine whether a serial history is a witness for another history in polynomial time. From this the overall result followed.

For the second result (Section 3.3), we compared the languages of conflict opacity and value opacity under two sets of assumptions. The first set of assumptions restricted the histories considered to those for which value opacity implicitly has a most-recent reads-from relation. Still, under this assumption, both conditions differ in how they handle unread writes. In conflict opacity such writes cause conflicts, and thus they restrict the ordering of a potential witness. In value opacity such an ordering restriction does not exist, and if the write is still unread in the witness the position of its transaction is only restricted according to the real-time order of the history it is part of. So, under this assumption conflict opacity implies value opacity, but the reverse claim does not hold. The second set of assumptions contained the first set of assumptions and in addition the read-before-update assumption. The latter assumption restricts histories to only contain transactions that read the values they write to beforehand. This assumption removes the difference in how unread writes are handled. Under the second set of assumptions, the languages of conflict opacity and value opacity are identical.

Condition	Complexity of correctness problem
Sequential consistency	Undecidable [3]
Linearizability	EXPSPACE [3]
View serializability	Unknown
Strict view serializability	Unknown
State serializability	Unknown
Strict state serializability [a]	Decidable (Section 4.2)
Conflict serializability	PSPACE [37]
Causal serializability	Unknown
Snapshot isolation	Unknown
Value Opacity[b]	Decidable (Section 4.3)
Conflict opacity	PSPACE (reduction to conflict serializability)
DU opacity	Unknown
TMS 1	Unknown
TMS 2	Unknown
VWC	Unknown

[a]Under assumptions, see Section 4.2
[b]Under assumptions, see Section 4.3

Table 5.2: Complexity of the correctness problems for different correctness conditions, our contribution shown in green

In the second part of the thesis concerned with the correctness problem, we first gave an overview of the existing results (see Table 5.2) and then presented results for two questions in this field. We determined that the respective correctness problem for (a) strict state serializability and (b) value opacity is decidable under assumptions. For strict state serializability (Section 4.2) we made the assumptions that implementations generate no dead transactions (transactions not affecting the end state) and only terminate if all transactions are finished. Our decidability proof consisted of an approach making it possible to decide the strict state serializability of any implementation under these assumptions. This approach involved exploring the naive state space of the given implementation in which each state consisted of the internal state of the implementation automaton and the history generated by the path from its start to the current state. We presented a compression of such a naive state space into equivalence classes. Two states are equivalent whenever the implementation is in the same internal state in both, and after any two given runs of the implementation starting in this state either the resulting extended histories are both serializable or both are not. Notably, there is an equivalence class **DM**, which contains all histories which are not state serializable and cannot become state serializable in any extension. We showed that there are only finitely many of these equivalence classes. This is the case because only a finite number of transactions can be read by future transactions. Then, we gave an algorithm exploring the naive state space of an implementation and constructing an equivalent state space using the equivalence classes as its states. This algorithm terminates because for a given problem instance there can only be a finite number of transitions and the number of possible equivalence classes is finite.

For our second result (Section 4.3) we employed a set of assumptions establishing an unambiguous value-based reads-from relation which notably limits all implementations to finish each transaction in a finite number of steps. This approach is analogue to the above approach, but it had to deal with value opacity allowing reads from commit invoked transactions which then later can be aborted. In the approach, this property was included in the equivalence classes. Additionally, as value opacity uses a different reads-from relation we derived new conditions of when transactions are readable or not. Given these changes, the remaining approach was identical to the approach for strict state serializability.

5.2 Discussion and Future Work

We will discuss the limitations, contributions and possible future work for each of the above results on its own in the following.

The membership problem for value opacity is NP-complete Value opacity as proposed by Guerraoui and Kapalka in 2008 ([46]) was the obvious choice for a gap to fill for the membership problem in the context of TMs as it was the first occurrence of opacity and also the definition of it that is most referred to. We did not choose the conflict-based variant of opacity since its differences to conflict serializability which is well studied are very little. The result itself was interesting given as it showed that while state serializability has a most-recent reads-from relation and value opacity an ambiguous value-based reads-from relation they still belong to the same complexity class. This contribution closes a gap in the literature regarding the membership problems of TM correctness conditions. It allows for realistic expectations for the run time of testing approaches. For future work it would be interesting to see if other conditions closely related to value opacity such as DU opacity, TMS 1, TMS 2 and VWC are also NP-complete or if some of the differences between these conditions and value opacity are significant enough to yield a higher or lower complexity. If not, the structure of the presented reduction of state serializability to value opacity could potentially serve as a blueprint for reduction of state serializability to these conditions.

Comparison of conflict opacity and value opacity The question of how these conditions compare to each other and under which assumptions they are equal, came up as conflict opacity was named opacity in the paper introducing it [42]. We discuss each of the three assumptions made and how applicable they may be to histories generated by actual TMs. The no-out-of-thin-air assumption (each read from a location reads the value written by the transaction that wrote to that location and committed most recently) is the most limiting assumption out of the three as it depends on the implementation of the TM and the memory model of the machine it runs. The assumption is not applicable for weaker memory models, where values read may not necessarily be those which were written most recently. Still, in stronger memory models or in implementations enforcing the assumption it is applicable. The unique writers assumptions (value are pairwise different) can be implemented using time stamps or a similar concept and the read-before update assumption (transactions also read each location they intend to write to) is very reasonable for many applications of TMs. The read-before-update assumption was adopted from the comparison of different serializability variants made by Papadimitriou in the paper presenting state serializability [73]. Often, reading values before modifying them is even necessary, for example for arithmetic operations. For future work, these assumptions can be used for a testing approach for an actual TM. Given they hold, it is possible to reduce checking the value opacity of a history to checking its conflict opacity. Additionally, it would be interesting to see how conflict opacity compares to other variants of opacity such as DU opacity, TMS 1, TMS 2 and VWC.

The correctness problem for SSR^{-} is decidable Besides the contribution of the result itself, we chose strict state serializability as a stepping stone towards determining the complexity of the correctness problem for value opacity. It was similar to value opacity in the sense that its constraints include a real-time order and do not constrain the order on a potential witness using conflicts as for example conflict serializability does. However, its most-recent reads-from relation made obtaining a result easier. To obtain the result for SSR, we made two assumptions. First, we assumed the implementation only terminates when all transactions are finished. This assumption allowed a simplified notion of equivalence where the reads of live transactions were also considered as they are expected to terminate. If one removes this assumption, the reads of live transactions are not relevant for the strict state serializability of a history, but they become so whenever the transaction finishes. Removing this assumption would imply the need for a potential new approach, if possible, to refine the equivalence classes - specifically the **DM** class - as now live transactions whose reads cannot be justified by any witness do not make a history permanently non-serializable except if they commit. As the number of threads and thus the number of live transactions in an instance of the problem is finite, and they can only read from a finite number of other transactions, it is likely that this would still lead to a finite number of equivalence classes. If this is the case, then the problem would likely still be decidable. Second, we assumed any implementation does not produce dead transactions. The issue with allowing dead transactions is that then state serializability is not prefix-closed. The reads of dead transactions are not considered for determining whether a history is strictly state serializable or not. Also, at any point, any previously not dead transaction may become dead. This leads to the problem that the **DM** equivalence class does not exist any more and any previously non strictly state serializable history may become strictly state serializable later on. Removing this assumption could lead to the problem being undecidable. A property which may still make this problem decidable is that the dead property propagates backwards. So if newer transactions become dead, older transactions whose liveness depends on these newer transactions also die. So, given a history which contains unreadable transactions which make the overall history not strictly state serializable, it may be possible to save which readable transactions need to become dead for the unreadable transaction to also become dead and make the history serializable again. If this is possible, unreadable transaction can still be forgotten. Still, even with its limitations this result is a significant step towards filling up the gaps in the related work. The approach also uses a reduction of histories into an equivalence classes which could be useful in other contexts if adapted accordingly. Future work could involve lifting the assumptions made, specifically by exploring if the previous approaches are feasible.

The correctness problem for OP^{-} is decidable This result was obtaining using the result for strict state serializability as a stepping stone. Compared to strict state serializability, the histories value opacity is defined upon can contain commit invoked transactions which can either be treated as committed or aborted in a witness. Also, a read does not necessarily read from the writer that most recently committed before it as is the case for strict state serializability. It can even read from writers that commit after it. Also, for a single read there can exist arbitrary many transactions of which each one can justify that read in a witness. This causes two issues for transferring the approach we used for strict state serializability to value opacity.

First, the possibility to read values from transactions that commit invoked after the read means value opacity in its original version is - as strict state serializability - not prefix-closed. This property causes a steep increase in complexity in any verification approach - if such an approach exists - thus often value opacity is defined as explicitly prefix-closed. Second, for one read there can be an arbitrary high number of potential writes justifying it. For an example of this issue, see Figure 5.1. Transaction 1 may read from any of the transactions of thread 2 if it invokes a read on x reading the value 1.

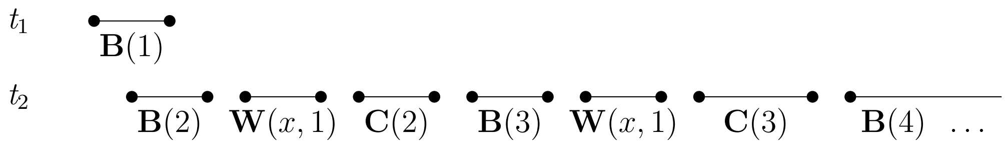


Figure 5.1: Issues with solving the correctness problem for value opacity

The problem with such a scenario is that an arbitrary high number of transactions can be read in the future, and thus cannot be forgotten in our approach. We first discuss our approach to addressing these issues, and then discuss possible other approaches for future work.

In this thesis, we opted for proving decidability of the correctness problem for implementations not producing histories containing such cases. We addressed the first issue by requiring that if a run of an implementation produces a non-opaque history, any extension of that run does as well. We did so via the reasonable-read constraint, which requires each read to read a value written by a previous committed or commit invoked transaction which can be the most recent writer on the respective variable before the read in a witness. This constraint as a side effect also excludes implementations in which values that no transaction can ever write in the implementation can be read. This is a fringe case which can also trivially be checked for in any implementation given as a DFA.

The second issue was harder to address. A first intuition would be to require that each history produced by an implementation has pairwise different values for each write, similar to the unique values assumption made in the comparison between value and conflict opacity. The issue with this is that a DFA implementation can only produce a finite number of write events with pairwise different values. Thus, this assumption would only include implementations that each have an upper bound to the number of writing committed transactions they can produce in a run. Every other transaction such an implementation could produce would be read-only which would limit the complexity of such histories given that read-only transactions can be forgotten in the style of approach we use. Because of this, we did not restrict the correctness problem to such implementations. Instead, we required each transaction to commit or abort in a fixed amount of time which made it possible to identify each writer for a read with a finite number of timestamps which we assumed to be the case. These assumptions still allow for arbitrary many writing transactions that are committed. Also, these assumptions keep the properties mentioned at the beginning of this paragraph mostly intact.

For future work, there are several interesting avenues. An obvious one would be to lift the constraints to determine the decidability of the correctness problem for value opacity for all possible implementations. If our approach were to be adopted to implementations without the reasonable-read constraint, it would mean that only histories in which a read cannot be justified by future writers in an extension of that history would belong to the **DM** equivalence class. Other non-opaque histories would belong to different equivalence classes. If the other constraints are still in place, the duration of each transaction is upper bounded by a fixed number and the number of concurrent transactions to one transaction is also upper bounded by a fixed number. This implies that after at most a fixed number of events the reading transaction is finished and each transaction concurrent to it as well, meaning its read cannot be justified by future writers in an extension anymore. This makes it very likely that the overall problem would still be decidable with only the RR-constraint lifted. Lifting the other two constraints restricting the duration of transactions and requiring timestamps would be challenging. It might very well make the problem undecidable given the issue presented in Figure 5.1. If it were to still be decidable, a promising avenue would be to group transactions into equivalence classes. The only way a transaction in an opaque history can contain arbitrary many events is if it repeatedly reads the same value from a location as multiple writes to the same location are not permitted. Reading different values from one location in the same transaction always means the history is not value opaque. Given these two facts, such a grouping into equivalence classes may be possible. Then these classes could be used to forget transactions in scenarios such as shown in Figure 5.1, if there is another transaction in the same equivalence class such that both fulfil conditions that imply they are interchangeable.

Besides lifting these constraints, it would also be interesting to conduct further research into different opacity variants and see if this approach is adaptable in some manner to these variants.

Concluding thoughts In this thesis, we have provided several results for the decidability of problems related to correctness conditions for TMs. The correctness of these results is supported by extensive proofs in this thesis. The results fill gaps in the existing results, and they can also be used as inspiration for practical approaches for verifying histories or TMs. Promising avenues for future work are expanding upon these results by removing their limitations and adapting them to different variants of the correctness conditions.

References

[1] IJsbrand Jan Aalbersberg and Hendrik Jan Hoogeboom. Characterizations of the decidability of some problems for regular trace languages. *Mathematical Systems Theory*, 22(1):1–19, 1989.

[2] Yehuda Afek, Alexander Matveev, and Nir Shavit. Pessimistic software lock-elision. In Marcos K. Aguilera, editor, *Distributed Computing - 26th International Symposium, DISC 2012, Salvador, Brazil, October 16-18, 2012. Proceedings*, volume 7611 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2012.

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

[4] Alasdair Armstrong, Brijesh Dongol, and Simon Doherty. Proving opacity via linearizability: A sound and complete method. In Ahmed Bouajjani and Alexandra Silva, editors, *FORTE 2017*, volume 10321 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 2017.

[5] Hagit Attiya, Sandeep Hans, Petr Kuznetsov, and Srivatsan Ravi. Safety of deferred update in transactional memory. In Mukaddim Pathan and Guiyi Wei, editors, *IEEE 33rd International Conference on Distributed Computing Systems, ICDCS 2013, 8-11 July, 2013, Philadelphia, Pennsylvania, USA*, pages 601–610. IEEE Computer Society, 2013.

[6] Hagit Attiya and Eshcar Hillel. Single-version stms can be multi-version permissive (extended abstract). In Marcos Kawazoe Aguilera, Haifeng Yu, Nitin H. Vaidya, Vikram Srinivasan, and Romit Roy Choudhury, editors, *Distributed Computing and Networking - 12th International Conference,*

ICDCN 2011, Bangalore, India, January 2-5, 2011. Proceedings, volume 6522 of *Lecture Notes in Computer Science*, pages 83–94. Springer, 2011.

[7] Josh Berdine, Tal Lev-Ami, Roman Manevich, G. Ramalingam, and Shmuel Sagiv. Thread quantification for concurrent shape analysis. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, volume 5123 of *Lecture Notes in Computer Science*, pages 399–413. Springer, 2008.

[8] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 1–10. ACM Press, 1995.

[9] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[10] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, 5(3):203–216, 1979.

[11] Eleni Bila, Simon Doherty, Brijesh Dongol, John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Defining and verifying durable opacity: Correctness for persistent software transactional memory. In Alexey Gotsman and Ana Sokolova, editors, *Formal Techniques for Distributed Objects, Components, and Systems - 40th IFIP WG 6.1 International Conference, FORTE 2020, Held as Part of the 15th International Federated Conference on Distributed Computing Techniques, DisCoTec 2020, Valletta, Malta, June 15-19, 2020, Proceedings*, volume 12136 of *Lecture Notes in Computer Science*, pages 39–58. Springer, 2020.

[12] Jesse D. Bingham, Anne Condon, Alan J. Hu, Shaz Qadeer, and Zhichuan Zhang. Automatic verification of sequential consistency for unbounded ad-

dresses and data values. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 427–439. Springer, 2004.

[13] Ranadeep Biswas and Constantin Enea. On the complexity of checking transactional consistency. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):165:1–165:28, 2019.

[14] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In Dean M. Tullsen and Brad Calder, editors, *34th International Symposium on Computer Architecture (ISCA 2007), June 9-13, 2007, San Diego, California, USA*, pages 81–91. ACM, 2007.

[15] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Verifying concurrent programs against sequential specifications. In Matthias Felleisen and Philippa Gardner, editors, *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*, volume 7792 of *Lecture Notes in Computer Science*, pages 290–309. Springer, 2013.

[16] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. On reducing linearizability to state reachability. *Information and Computation*, 261:383–400, 2018.

[17] Tim Braun, Anne Condon, Alan J. Hu, Kai S. Juse, Marius Laza, Michael Leslie, and Rita Sharma. Proving sequential consistency by model checking. In *Proceedings of the Sixth IEEE International High-Level Design Validation and Test Workshop 2001, Monterey, California, USA, November 7-9, 2001*, pages 103–108. IEEE Computer Society, 2001.

[18] Pavol Cerný, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. Model checking of linearizability of concurrent list imple-

mentations. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 465–479. Springer, 2010.

[19] Ariel Cohen, John W. O'Leary, Amir Pnueli, Mark R. Tuttle, and Lenore D. Zuck. Verifying correctness of transactional memories. In *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings*, pages 37–44. IEEE Computer Society, 2007.

[20] Anne Condon and Alan J. Hu. Automatable verification of sequential consistency. In Arnold L. Rosenberg, editor, *Proceedings of the Thirteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2001, Heraklion, Crete Island, Greece, July 4-6, 2001*, pages 113–121. ACM, 2001.

[21] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In Christian Scheideler and Jeremy T. Fineman, editors, *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures, SPAA 2018, Vienna, Austria, July 16-18, 2018*, pages 271–282. ACM, 2018.

[22] Luke Dalessandro, David Dice, Michael L. Scott, Nir Shavit, and Michael F. Spear. Transactional mutex locks. In Pasqua D'Ambra, Mario Rosario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing, 16th International Euro-Par Conference, Ischia, Italy, August 31 - September 3, 2010, Proceedings, Part II*, volume 6272 of *Lecture Notes in Computer Science*, pages 2–13. Springer, 2010.

[23] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In John Paul Shen and Margaret Martonosi, editors, *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, pages 336–346. ACM, 2006.

[24] John Derrick, Simon Doherty, Brijesh Dongol, Gerhard Schellhorn, Oleg Travkin, and Heike Wehrheim. Mechanized proofs of opacity: a comparison of two techniques. *Formal Aspects of Computing*, 30(5):597–625, 2018.

[25] John Derrick, Brijesh Dongol, Gerhard Schellhorn, Oleg Travkin, and Heike Wehrheim. Verifying opacity of a transactional mutex lock. In Nikolaj Bjørner and Frank S. de Boer, editors, *FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings*, volume 9109 of *Lecture Notes in Computer Science*, pages 161–177. Springer, 2015.

[26] Dave Dice and Nir Shavit. What really makes transactions fast? In *ACM Workshop*, 2006.

[27] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In Shlomi Dolev, editor, *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2006.

[28] Simon Doherty, Brijesh Dongol, John Derrick, Gerhard Schellhorn, and Heike Wehrheim. Proving opacity of a pessimistic STM. In Panagiota Fatourou, Ernesto Jiménez, and Fernando Pedone, editors, *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain*, volume 70 of *LIPIcs*, pages 35:1–35:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016.

[29] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25(5):769–799, 2013.

[30] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 155–165. ACM, 2009.

[31] Dmytro Dziuma, Panagiota Fatourou, and Eleni Kanellou. Consistency for transactional memory computing. In Rachid Guerraoui and Paolo Romano, editors, *Transactional Memory. Foundations, Algorithms, Tools, and Applications - COST Action Euro-TM IC1001*, volume 8913 of *Lecture Notes in Computer Science*, pages 3–31. Springer, 2015.

[32] Sameh Elnikety, Fernando Pedone, and Willy Zwaenepoel. Generalized snapshot isolation and a prefix-consistent implementation. Technical report, EPFL, 2004.

[33] Michael Emmi, Rupak Majumdar, and Roman Manevich. Parameterized verification of transactional memories. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 134–145. ACM, 2010.

[34] Robert Ennals. Software transactional memory should not be obstruction-free. In *Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report*, 2006.

[35] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[36] Azadeh Farzan and P Madhusudan. Algorithms for atomicity. Unpublished, 2007.

[37] Azadeh Farzan and P. Madhusudan. Monitoring atomicity in concurrent programs. In Aarti Gupta and Sharad Malik, editors, *Computer Aided*

Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings, volume 5123 of *Lecture Notes in Computer Science*, pages 52–65. Springer, 2008.

[38] Alan D. Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis E. Shasha. Making snapshot isolation serializable. *ACM Transactions on Database Systems*, 30(2):492–528, 2005.

[39] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Transactions on Computer Systems*, 25(2):5, 2007.

[40] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[41] Håkan Grahn. Transactional memory. *Journal of Parallel and Distributed Computing*, 70(10):993–1008, 2010. Transactional Memory.

[42] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Model checking transactional memories. *Distributed Computing*, 22(3):129–145, 2010.

[43] Rachid Guerraoui, Maurice Herlihy, Michal Kapalka, and Bastian Pochon. Robust contention management in software transactional memory. In *Proceedings of the OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL'05)*, 2005.

[44] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Polymorphic contention management. In Pierre Fraigniaud, editor, *Distributed Computing, 19th International Conference, DISC 2005, Cracow, Poland, September 26-29, 2005, Proceedings*, volume 3724 of *Lecture Notes in Computer Science*, pages 303–323. Springer, 2005.

[45] Rachid Guerraoui, Maurice Herlihy, and Bastian Pochon. Toward a theory of transactional contention managers. In Marcos Kawazoe Aguilera and James Aspnes, editors, *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, PODC 2005, Las Vegas, NV, USA, July 17-20, 2005*, pages 258–264. ACM, 2005.

[46] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In Siddhartha Chatterjee and Michael L. Scott, editors, *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, pages 175–184. ACM, 2008.

[47] Lance Hammond, Vicky Wong, Michael K. Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *31st International Symposium on Computer Architecture (ISCA 2004), 19-23 June 2004, Munich, Germany*, pages 102–113. IEEE Computer Society, 2004.

[48] Tim Harris, Adrián Cristal, Osman S. Unsal, Eduard Ayguadé, Fabrizio Gagliardi, Burton Smith, and Mateo Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, 2007.

[49] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Verifying sequential consistency on shared-memory multiprocessor systems. In Nicolas Halbwachs and Doron A. Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 301–315. Springer, 1999.

[50] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In Alan Jay Smith, editor, *Proceedings of the 20th Annual International Symposium on Computer Architecture, San Diego, CA, USA, May 1993*, pages 289–300. ACM, 1993.

[51] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[52] William N. Scherer III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In Marcos Kawazoe

Aguilera and James Aspnes, editors, *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Distributed Computing, PODC 2005, Las Vegas, NV, USA, July 17-20, 2005*, pages 240–248. ACM, 2005.

[53] Damien Imbs and Michel Raynal. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science*, 444:113–127, 2012.

[54] Udo Kelter. The complexity of strict serializability revisited. *Information Processing Letters*, 25(6):407–412, 1987.

[55] Elaine Lies Kerry Grens. Spike in deaths blamed on 2003 new york blackout. https://www.reuters.com/article/us-blackout-newyork-idUSTRE80Q07G20120127. Accessed: 2023-05-27.

[56] Jürgen König and Heike Wehrheim. Value-based or conflict-based? opacity definitions for stms. In Dang Van Hung and Deepak Kapur, editors, *Theoretical Aspects of Computing - ICTAC 2017 - 14th International Colloquium, Hanoi, Vietnam, October 23-27, 2017, Proceedings*, volume 10580 of *Lecture Notes in Computer Science*, pages 118–135. Springer, 2017.

[57] Jürgen König and Heike Wehrheim. Data independence for software transactional memory. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, volume 11460 of *Lecture Notes in Computer Science*, pages 263–279. Springer, 2019.

[58] Jürgen König and Heike Wehrheim. On the correctness problem for serializability. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings*, volume 12819 of *Lecture Notes in Computer Science*, pages 47–64. Springer, 2021.

[59] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony D. Nguyen. Hybrid transactional memory. In Josep Torrellas and Siddhartha Chatterjee, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006*, pages 209–220. ACM, 2006.

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

[61] Mohsen Lesani, Victor Luchangco, and Mark Moir. Putting opacity in its place. In *Workshop on the theory of transactional memory*, pages 137–151, 2012.

[62] Mohsen Lesani and Jens Palsberg. Proving non-opacity. In Yehuda Afek, editor, *Distributed Computing - 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings*, volume 8205 of *Lecture Notes in Computer Science*, pages 106–120. Springer, 2013.

[63] Mohsen Lesani and Jens Palsberg. Decomposing opacity. In Fabian Kuhn, editor, *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer Science*, pages 391–405. Springer, 2014.

[64] Nancy Leveson et al. Medical devices: The therac-25. *Appendix of: Safeware: System Safety and Computers*, 1995.

[65] Nancy G Leveson and Clark S Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[66] Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. Model checking linearizability via refinement. In Ana Cavalcanti and Dennis Dams, editors, *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, volume 5850 of *Lecture Notes in Computer Science*, pages 321–337. Springer, 2009.

[67] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, page 79–90, New York, NY, USA, 2010. Association for Computing Machinery.

[68] Roman Manevich, Tal Lev-Ami, Mooly Sagiv, Ganesan Ramalingam, and Josh Berdine. Heap decomposition for concurrent shape analysis. In María Alpuente and Germán Vidal, editors, *Static Analysis, 15th International Symposium, SAS 2008, Valencia, Spain, July 16-18, 2008. Proceedings*, volume 5079 of *Lecture Notes in Computer Science*, pages 363–377. Springer, 2008.

[69] Virendra Marathe and Michael Scott. A qualitative survey of modern software transactional memory systems. Technical Report TR 839, Dept. of Computer Science, Univ. of Rochester, 2004.

[70] Ragnar Normann and Lene T. Østby. A theoretical study of 'snapshot isolation'. In Luc Segoufin, editor, *Database Theory - ICDT 2010, 13th International Conference, Lausanne, Switzerland, March 23-25, 2010, Proceedings*, ACM International Conference Proceeding Series, pages 44–49. ACM, 2010.

[71] John W. O'Leary, Bratin Saha, and Mark R. Tuttle. Model checking transactional memory with spin. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*, pages 335–342. IEEE Computer Society, 2009.

[72] Christos H Papadimitriou. Some computational problems related to database concurrency control. In *Proceedings of the Conference on Theoretical Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1977*, 1977.

[73] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[74] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in STM. In Andréa W. Richa and Rachid Guerraoui, editors, *Proceedings of the 29th Annual ACM Symposium on Principles of Distributed Computing, PODC 2010, Zurich, Switzerland, July 25-28, 2010*, pages 16–25. ACM, 2010.

[75] Miroslav Popovic, Branislav Kordic, and Ilija Basicevic. Transaction scheduling for software transactional memory. In *2017 IEEE 2nd International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pages 191–195. IEEE, 2017.

[76] Kevin Poulsen. Software bug contributed to blackout. https://www.theregister.com/2004/02/12/software_bug_contributed_to_blackout/. Accessed: 2023-05-27.

[77] Kevin Poulsen. Tracking the blackout bug. https://www.theregister.com/2004/04/08/blackout_bug_report/. Accessed: 2023-05-27.

[78] Shaz Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. *IEEE Transactions on Parallel and Distributed Systems*, 14(8):730–741, 2003.

[79] Pedro Ramalhete, Andreia Correia, Pascal Felber, and Nachshon Cohen. Onefile: A wait-free persistent transactional memory. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 151–163. IEEE, 2019.

[80] Michel Raynal, Gérard Thia-Kime, and Mustaque Ahamad. From serializable to causal transactions for collaborative applications. In *23rd EUROMICRO Conference '97, New Frontiers of Information Technology, 1-4 September 1997, Budapest, Hungary*, page 314. IEEE Computer Society, 1997.

[81] Torvald Riegel, Christof Fetzer, and Pascal Felber. Snapshot Isolation for Software Transactional Memory. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*, pages 1–10. Association for Computing Machinery (ACM), 2006.

[82] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Ben Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In Josep Torrellas and Siddhartha Chatterjee, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006*, pages 187–197. ACM, 2006.

[83] Gerhard Schellhorn, Monika Wedel, Oleg Travkin, Jürgen König, and Heike Wehrheim. Fastlane is opaque - a case study in mechanized proofs of opacity. In Einar Broch Johnsen and Ina Schaefer, editors, *Software Engineering and Formal Methods - 16th International Conference, SEFM 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings*, volume 10886 of *Lecture Notes in Computer Science*, pages 105–120. Springer, 2018.

[84] Ralf Schenkel and Gerhard Weikum. Integrating snapshot isolation into transactional federations. In *International Conference on Cooperative Information Systems*, pages 90–101. Springer, 2000.

[85] William N Scherer III and Michael L Scott. Randomization in stm contention management (poster paper). In *Proceedings of the 24th ACM Symposium on Principles of Distributed Computing, Las Vegas, NV*, volume 10, 2005.

[86] Michael Scott. Sequential specification of transactional memory semantics. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, 2006.

[87] Michael L. Scott. *Shared-Memory Synchronization.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013.

[88] Ravi Sethi. Useless actions make a difference: Strict serializability of database updates. *Journal of the ACM*, 29(2):394–403, 1982.

[89] Ohad Shacham, Eran Yahav, Guy Golan-Gueta, Alex Aiken, Nathan Grasso Bronson, Mooly Sagiv, and Martin T. Vechev. Verifying atomicity via data independence. In Corina S. Pasareanu and Darko Marinov, editors, *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 26–36. ACM, 2014.

[90] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[91] Konrad Siek and Pawel T. Wojciechowski. Atomic RMI: A distributed transactional memory framework. *International Journal of Parallel Programming*, 44(3):598–619, 2016.

[92] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In Daniel A. Reed and Vivek Sarkar, editors, *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, pages 141–150. ACM, 2009.

[93] Michael F. Spear, Virendra J. Marathe, William N. Scherer III, and Michael L. Scott. Conflict detection and validation strategies for software transactional memory. In Shlomi Dolev, editor, *Distributed Computing, 20th International Symposium, DISC 2006, Stockholm, Sweden, September 18-20, 2006, Proceedings*, volume 4167 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 2006.

[94] Michael F. Spear, Maged M. Michael, and Christoph von Praun. Ringstm: scalable transactions with a single atomic instruction. In Friedhelm Meyer auf der Heide and Nir Shavit, editors, *SPAA 2008: Proceedings of the 20th Annual ACM Symposium on Parallelism in Algorithms and Architectures, Munich, Germany, June 14-16, 2008*, pages 275–284. ACM, 2008.

[95] Herb Sutter and James Larus. Software and the concurrency revolution: Leveraging the full power of multicore processors demands new tools and new thinking from the software industry. *Queue*, 3(7):54–62, sep 2005.

[96] Jeannette M. Wing and Chun Gong. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing*, 17(1-2):164–182, 1993.

[97] Mihalis Yannakakis. Serializability by locking. *Journal of the ACM*, 31(2):227–244, 1984.

A
Proofs for Section 3.2

In this appendix, we are going to prove lemmas 2, 3, 4 and 1 in this order. With these given we will prove Theorem 1.

Lemma 2 (One-to-One mapping of transactions)**.** *For an arbitrary p-history ph, it holds that*

$$tr(ph) = tr(f(ph)) \text{ (transaction sets identical) } .$$

Proof. Take an arbitrary p-history ph. Applying the reduction function f to it generates a complete transaction for each transaction in ph, it generates no other transactions. The claim holds. □

Lemma 3 (Preservation of *SR*-constraints)**.** *For an arbitrary p-history ph, it holds that*

$$\mathbf{W}^{tr_1}_{thr(tr_1)}(var, val) \in f(ph) \rightarrow val = tr_1 \wedge var \in WS^{vo}_{ph}(tr_1)$$
$$\mathbf{R}^{tr_1}_{thr(tr_1)}(var, val) \in f(ph) \rightarrow \exists tr_2 \in Tr : (tr_2, tr_1, var) \in ph.RF \wedge val = tr_2$$
$$\forall tr \in tr^-(ph) : tr_w \prec_{f(ph)} tr \wedge tr \prec_{f(ph)} tr_r$$

Proof. Take an arbitrary p-history ph.

FIRST EQUATION Given an arbitrary write event $\mathbf{W}^{tr_1}_{thr(tr_1)}(var, val) \in f(ph)$, by construction of f it must be the result of r applied to a write event of ph. There are no other write events generated by other means by f. Let $\mathbf{W}^{tr'}_{thr(tr')}[Var']$ be this event. The following sequence is the result of r on input of this event:

$$r_{ph}(\mathbf{W}^{tr'}_{thr(tr')}[Var']) = \underset{var' \in Var'}{\bullet} \mathbf{W}^{tr'}_{thr(tr')}(var', tr').$$

Note that $Var' = WS^{vo}_{ph}(tr')$. For $\mathbf{W}^{tr_1}_{thr(tr_1)}(var, val)$ to be a member of the above sequence generated by r the following equations must hold:

$$val = tr', \quad var \in WS^{vo}_{ph}(tr'), \quad \text{and } tr' = tr_1.$$

It follows that

$$\begin{aligned} \mathbf{W}^{tr_1}_{thr(tr_1)}(var, val) = \mathbf{W}^{tr_1}_{thr(tr_1)}(var, tr_1), \qquad & \mid (val = tr', tr' = tr_1) \\ \text{and } var \in WS^{vo}_{ph}(tr_1). \qquad & \mid (var \in WS^{vo}_{ph}(tr'), tr' = tr_1) \end{aligned}$$

This proves the claim.

SECOND EQUATION Given an arbitrary read event $\mathbf{R}^{tr_1}_{thr(tr_1)}(var, val) \in f(ph)$, it must be the result of r applied to a read event of ph. There are no other read events generated by other means by f. Let $\mathbf{R}^{tr'}_{thr(tr')}[Var']$ be this event. The following sequence is generated by r for this event:

$$r_{ph}(\mathbf{R}^{tr'}_{thr(tr')}[Var']) = \underset{var \in Var'}{\bullet} \mathbf{R}^{tr'}_{thr(tr')}(var, rf_{ph}(tr', var)).$$

For $\mathbf{R}^{tr_1}_{thr(tr_1)}(var, val)$ to be in that sequence it must hold that

$$val = rf_{ph}(tr', var), \quad \text{and } tr' = tr_1.$$

Note that the first equation implies $var \in Var'$, which is not mentioned separately as it is not needed further on. We can now substitute val in the event:

$$\mathbf{R}^{tr_1}_{thr(tr_1)}(var, val) = \mathbf{R}^{tr_1}_{thr(tr_1)}(var, rf_{ph}(tr_1, var)) \qquad \mid (val = rf_{ph}(tr', var), tr' = tr_1).$$

The result of $rf_{ph}(tr_1, var)$ is $tr_2 \in Tr,$ where $(tr_2, tr_1, var) \in ph.RF$. Thus,

$$\exists tr_2 \in Tr : (tr_2, tr_1, var) \in ph.RF \wedge val = tr_2$$

is true, and the claim holds.

Third equation By construction of f, the commit of tr_w is before the begin of every other transaction and the begin of tr_r is after the commit of every other transaction. The claim holds. □

Lemma 4 (Preventing additional OP-constraints)**.** *For an arbitrary p-history ph, it holds that*

$$\forall tr, tr' \in tr^{-}(ph) : \neg(tr' \prec_{f(ph)} tr \vee tr \prec_{f(ph)} tr')$$

Proof. As evident by the definition of f, each non-augmented transaction is concurrent to each other. Thus, no two non-augmented transactions can be real-time ordered with each other. □

Lemma 1. *Given an arbitrary p-history ph, it holds that*

1. $$f(ph) \textit{ is opaque under } OP$$
$$\leftrightarrow$$
$$ph \textit{ is serializable under } SR$$

2. *and f is computable in polynomial time.*

Proof. We will prove both directions of the first statement separately.

$\leftarrow$ Given an arbitrary serializable p-history ph and an arbitrary SR-witness ph_s of it, we construct a g-history h_s, which we will prove to be an OP-witness for $f(ph)$. Let h_s have two properties:

- it is equivalent to $f(ph)$
- and $h_s.RT = ph_s.RT$.

The second assignment is meaningful as $tr(h_s) = tr(ph_s)$. This is true as:

$$\begin{aligned} tr(ph_s) &= tr(ph) && \mid (SR\text{-witness}) \\ tr(ph) &= tr(f(ph)) && \mid (\text{Lemma 2} \\ tr(f(ph)) &= tr(h_s). && \mid (f(ph) \text{ equivalent to } h_s) \end{aligned}$$

Both properties completely define a sequence of g-events/a g-history as events, an internal thread order and a total real-time order have been assigned. The last two properties imply the event order, so it is not necessary to specify explicitly. Additionally, each transaction has the same read and write set in between all of these histories. Similarly to before, this holds for $f(ph)$ and ph because of the construction of f. For h_s and $f(ph)$, it is the case because h_s is an OP-witness of $f(ph)$. For ph and ph_s, it is the case because ph_s is an SR-witness of ph.

We show that h_s is an OP-witness of $f(ph)$.

1. h_s is serial:
   True, as its real-time order is total.

2. h_s is equivalent to a g-history in $compl(h)$:
   By construction of f, all transactions are finished in $f(ph)$. Thus, it is the completion of itself, meaning that $compl(f(ph)) = \{f(ph)\}$ holds. By its definition, h_s is equivalent to $f(ph)$, the claim holds.

3. Each transaction in h_s is legal:
   Note that each transaction is committed in $f(ph)$ and thus also in h_s. For the history to be legal then each read has to read the most recently written value on its variable. Let $\mathbf{R}^{tr_1}_{thr(tr_1)}(var, val)$ be a read in h_s. By construction

of h_s, a read exists in h_s iff it exists in $f(ph)$. As of Lemma 3 val is the transaction that tr_1 read from in ph

$\mathbf{R}^{tr_1}_{thr(tr_1)}(var, val) \in f(ph) \to \exists tr_2 \in Tr : (tr_2, tr_1, var) \in ph.RF \wedge val = tr_2$

As ph_s is an SR-witness of ph, their reads-from relation is identical. So we can substitute ph with ph_s :

$\mathbf{R}^{tr_1}_{thr(tr_1)}(var, val) \in f(ph) \to \exists tr_2 \in Tr : (tr_2, tr_1, var) \in ph_s.RF \wedge val = tr_2$

The following statements are implied by $(tr_2, tr_1, var) \in ph_s.RF$:

$$tr_2 \prec_{ph_s} tr_1,$$
$$\neg(\exists tr_x \in Tr : tr_2 \prec_{ph_s} tr_x \prec_{ph_s} tr_1 \wedge var \in WS^{vo}_{ph_s}(tr_x)).$$

The real-time orders of ph_s and h_s are identical, their transaction sets are identical and the read and write sets for each transaction are identical. Thus, we can substitute ph_s with h_s resulting in

$$tr_2 \prec_{h_s} tr_1,$$
$$\neg(\exists tr_x \in Tr : tr_2 \prec_{h_s} tr_x \prec_{h_s} tr_1 \wedge var \in WS^{vo}_{h_s}(tr_x)).$$

Thus, tr_2 is the most recent writer on var for tr_1. By Lemma 3, we know that tr_2 wrote tr_2 to var and tr_1 read tr_2 from var. Thus, it has read the most recently written value on its variable.

Overall each transaction then is legal in h_s making it legal altogether.

4. h_s preserves the real-time order of $f(ph)$:
   By definition,

$$\forall tr \in tr^-(ph_s) : tr_w \prec_{ph_s} tr \wedge tr \prec_{ph_s} tr_r.$$

and since $h_s.RT = ph_s.RT$

$$\forall tr \in tr^-(h_s) : tr_w \prec_{h_s} tr \wedge tr \prec_{h_s} tr_r.$$

By lemmas 3 (third equation) and 4, we know that the only real-time relations in $f(ph)$ are

$$\forall tr \in tr^{-}(f(ph)) : tr_w \prec_{f(ph)} tr \wedge tr \prec_{f(ph)} tr_r.$$

Thus

$$f(ph).RT \subseteq h_s.RT.$$

$\rightarrow$ Given an arbitrary p-history *ph*, we will show that if $f(ph)$ has at least one *OP*-witness, then *ph* has at least one *SR*-witness. The proof basically has the reverse structure as the proof of the other direction above. Assume an arbitrary *OP*-witness of $f(ph)$, named h_s. Let ph_s be a p-history having

- the same events as *ph*
- and its real-time order is $h_s.RT$.

By an analogue argument as in the proof of the other direction, this is a well-formed definition of a p-history. We show that ph_s is an *SR*-witness of *ph*.
Because we use the definition of SR^+ with the assumption that each thread executes a single transaction, the internal thread order requirement holds trivially. The augmented transactions are ordered correctly. This is because tr_w is real-time ordered before any other transaction and tr_r real-time ordered after any other transaction in $f(ph)$. By preservation of real-time order, the same holds in h_s and thus in ph_s.

We prove p-equivalence ($ph \equiv ph_s$). Trivially both histories contain the same events. It is left to prove that $ph.RF = ph_s.RF$. Let $(tr_1, tr_2, var) \in ph.RF$ be an arbitrary member of the reads-from relation of *ph*. Then there exist the p-events $\mathbf{W}^{tr_1}_{thr(tr_1)}[Var']$ and $\mathbf{R}^{tr_2}_{thr(tr_2)}[Var'']$ in *ph* s.t. $var \in Var'$ and $var \in Var''$. Also, it holds that the write is ordered before the read with no other write on *var* in between:

$$tr_1 \prec_{ph} tr_2, \quad \neg(\exists tr_x \in Tr : tr_1 \prec_{ph} tr_x \prec_{ph} tr_2 \wedge var \in WS^{vo}_{ph}(tr_x)).$$

By construction of f and r, the two g-events $ev_w = \mathbf{W}^{tr_1}_{thr(tr_1)}(var, tr_1)$ and $ev_r = \mathbf{R}^{tr_2}_{thr(tr_2)}(var, tr_1)$ exist in $f(ph)$. Also by construction of these functions, there exists no additional writes to *var* writing tr_1 as their value in $f(ph)$. As h_s is an *OP*-witness of $f(ph)$ both contain the same events. Thus, the two events exist in h_s and the write is unique in the sense discussed above. By this uniqueness and by the legality of h_s, ev_w must be the last write on *var* before ev_r in h_s. As h_s is serial this implies for transactions tr_1 and tr_2 the following:

$$tr_1 \prec_{h_s} tr_2, \quad \neg(\exists tr_x \in Tr : tr_1 \prec_{h_s} tr_x \prec_{h_s} tr_2 \land var \in WS^{vo}_{h_s}(tr_x)).$$

By construction of ph_s, it is given that $ph_s.RT = h_s.RT$. Also write and read sets are identical for each transaction as noted in the proof of the other direction. Thus, we can substitute h_s by ph_s:

$$tr_1 \prec_{ph_s} tr_2, \quad \neg(\exists tr_x \in Tr : tr_1 \prec_{ph_s} tr_x \prec_{ph_s} tr_2 \land var \in WS^{vo}_{ph_s}(tr_x)).$$

Furthermore in ph_s tr_1 writes to *var* and tr_2 reads from var. Combining these two facts shows that tr_2 reads *var* from tr_1 in ph_s or formally: $(tr_1, tr_2, var) \in ph_s.RF$. This proves the claim.

f IS COMPUTABLE IN POLYNOMIAL TIME Subfunction r computes replacements based on a single input event without further context, this is trivially computable in polynomial time w.r.t. to its input. The reduction function f then spawns one begin and one commit for each transaction and applies r twice for each transaction, this is trivially doable in polynomial w.r.t. to its input. □

Lemma 5. *Given two g-histories h and h_s, it is determinable in polynomial time whether h_s is an OP-witness of h or not.*

Proof. We discuss how each of the properties of *OP*-witnesses is determinable in polynomial time.

1. h_s is serial: For each transaction find all of its events, check whether its events are interrupted by another transaction.

2. h_s is equivalent to a g-history in $compl(h)$: Check for each transaction whether its events in h are a prefix of its events h_s and whether it has been complete according to the completion rules.

3. Each transaction in h_s is legal: Iterate over each read in h_s check whether the last write on its variable wrote the value the read read.

4. h_s preserves the real-time order of h: Build real-time orders of h and h_s by iterating over each commit and checking for begins afterwards. Then check whether $h.RT \subseteq h_s.RT$.

All of these points can obviously be computed in polynomial time. □

Theorem 1 (Complexity of the membership problem for OP)**.** *The membership problem for OP is NP-complete.*

Proof. By Lemma 1, f is a reduction function from the NP-complete membership problem for *SR* to the membership problem for OP. By Lemma 5, it is computable in polynomial time whether a given g-historyh_s is an OP-witness for another g-history h. From combining both lemmas follows the NP-completeness of the membership problem for OP. □

B

Proofs for Section 4.2

Proposition 2 (Generation of candidates by insertion function)**.** *Given a p-history ph and a p-event pev, it holds that*

$$\bigcup_{ph_c \in C_{ph}} ins(ph_c, pev) = C_{ph \cdot pev}.$$

Proof. We first show that $\bigcup_{ph_c \in C_{ph}} ins(ph_c, pev) \subseteq C_{ph \cdot pev}$. We prove this via case distinction over pev.

$\boldsymbol{pev = \mathbf{W}_t^{tr}[Var']}$ **:**
Let ph_c be an arbitrary candidate of ph_c. Let the result of $ins(ph_c, \mathbf{W}_t^{tr}[Var'])$ be $\{ph_c'\}$. The set only contains one event as for a write event ins generates only one candidate. By definition of ins, pev is inserted directly after the last event of tr in ph_c. For ph_c' to be a candidate it must be serial and preserve the real-time order of $ph \cdot pev$.

1. ph_c' is serial: It is trivially since the write is inserted directly after the read of its transaction in the (serial) candidate ph_c.

2. $ph \cdot pev$ $(ph \cdot pev).RT \subseteq ph_c'.RT$: The real-time order of $ph \cdot pev$ is identical to the real-time order of ph as the write is appended at the end. All p-events of ph_c are present in ph_c' with the same relative order. The only different element in ph_c' is pev. Thus, the real-time order of ph_c' is the real-time order of ph_c with more or equal to 0 additional rt-elements as a write is inserted somewhere in ph_c possibly generating new rt-elements. By this reasoning,

$$(ph \cdot pev).RT = ph.RT \text{ and } \; ph_c.RT \subseteq ph_c'.RT \quad \text{holds.}$$

Additionally, ph_c is a candidate of ph, and thus $ph.RT \subseteq ph_c.RT$ is true. This overall implies

$$(ph \cdot pev).RT \subseteq ph_c'.RT.$$

***pev* $= \mathbf{R}_t^{tr}[\,\boldsymbol{Var'}]$:**

Let ph_c be an arbitrary candidate of ph_c. Let the ph_c' be a p-history s.t. $ph_c' \in ins(ph_c, \mathbf{R}_t^{tr}[\,Var'])$. By definition of ins_r, pev is inserted somewhere after the last write of a non-augmented transaction.

1. ph_c' is serial: The p-history ph_c' is a supersequence of the serial p-history ph_c containing a read as the only additional event. By definition of ins, the read is not inserted between a read and a write of one transaction; thus, the result is serial.

2. ph_c' preserves the real-time order of $ph \cdot pev$: For $ph \cdot pev$ its real-time order consists of the real-time order of ph unified with the new rt-elements caused by pev.

$$(ph \cdot pev).RT = ph.RT \cup \{(tr', tr) \mid tr' \text{ is finished in } ph\}.$$

For ph_c' its real-time order is the real-time order of ph_c unified with the new rt-elements caused by pev. Note that the transactions finished in ph and ph_c are identical.

$$ph_c'.RT = ph_c.RT \cup \{(tr', tr) \mid tr' \text{ is finished in } ph\},$$

which in turn implies

$$ph_c.RT \subseteq ph_c'.RT.$$

Additionally, ph_c is a candidate of ph, and thus

$$ph.RT \subseteq ph_c.RT.$$

This overall implies

$$(ph \cdot pev).RT \subseteq ph'_c.RT.$$

We prove the other direction $C_{ph\cdot pev} \subseteq \bigcup_{ph_c \in C_{ph}} ins(ph_c, pev)$. Take an arbitrary candidate $ph'_c \in C_{ph\cdot pev}$ s.t. $ph'_c = pev'_0 \dots pev \dots pev'_n$. By Proposition 1, it holds there exists a candidate $ph''_c \in C_{ph}$ s.t. $ph''_c \sqsubseteq ph'_c$. Let $ph''_c = pev'_0 \dots pev'_n$. We show that

$$ph'_c \in ins(ph''_c, pev)$$

holds.

$\boldsymbol{pev} = \mathbf{R}_{\boldsymbol{t}}^{\boldsymbol{tr}}[\boldsymbol{Var'}]$:
Let pev_x be the last write in ph'_c, and thus also in ph''_c. Let tr_x be the transaction of that write event. It is the case that $pev_x <_{ph\cdot pev} pev$, and thus $tr_x \prec_{ph\cdot pev} tr$ holds. Then, as ph'_c preserves the real-time order of $ph \cdot pev$, it must hold that $pev_x <_{ph'_c} pev$. Let $en(ph'_c)$ be the subsequence of ph'_c after pev_x and $st(ph'_c)$ be the subsequence before and including pev_x. As ph'_c and ph''_c only differ with regard to pev, it holds that

$$st(ph'_c) = st(ph''_c),$$

and there exists a index n s.t. $en(ph''_c) = add(en(ph'_c), pev, n)$. It is trivial to see that the following holds:

$$st(ph'_c) \cdot add(en(ph'_c), pev, n) \in \{st(ph'_c) \cdot add(en(ph'_c), pev, n) \mid 0 \leq n \leq lsInd(en(ph_c))\}.$$

$\boldsymbol{pev} = \mathbf{W}_{\boldsymbol{t}}^{\boldsymbol{tr}}[\boldsymbol{Var'}]$:
Let pev_x be the read event of tr in ph'_c, and thus also in ph''_c.
Thus then, $ph''_c = pev'_0 \ldots pev_x \ldots pev'_n$ holds. The candidate ph'_c is serial.
Thus, $ph'_c = pev'_0 \ldots pev_x \; pev \ldots pev'_n$ holds.
Also, it holds that $ins(ph''_c, pev) = \{pev'_0 \ldots pev_x \; pev \ldots pev'_n\}$ thus the claim holds.

□

Lemma 11 (Conditions for fixed rf-elements in p-histories)**.** *Given a p-history ph, two arbitrary transactions tr, tr' and an arbitrary variable x, it holds for all $(tr, tr', x) \in ph.RF$ that*

$$(tr, tr', x) \in ph.RF_{fix} \leftrightarrow tr' \neq tr_r.$$

Proof. We show that both directions of the equivalency hold true.
Direction →:

$$(tr, tr', x) \in ph.RF_{fix} \rightarrow tr' \neq tr_r$$

We show the contraposition:

$$tr' = tr_r \rightarrow (tr, tr', x) \notin ph.RF_{fix}.$$

It has to hold that

$$\exists seq \in PEv^* : (tr, tr_r, x) \notin (ph \cdot seq).RF.$$

Assume the sequence contains exactly one empty read and one write on x of a transaction tr'' not contained in ph, then $(tr'', tr_r, x) \in (ph \cdot seq).RF$ holds which means $(tr, tr_r, x) \notin (ph \cdot seq).RF$. Such a sequence trivially always exists.

Direction ←:

$$tr' \neq tr_r \rightarrow (tr, tr', x) \in ph.RF_{fix}$$

For this to be true, the following must hold: $\forall seq \in PEv^* : (tr, tr', x) \in (ph \cdot seq).RF$. We show this via induction over $seq = pev_0 \dots pev_n$ $n \in \mathbb{N}$.

Induction start ($seq = \epsilon$)**:** Trivially true.

Induction step ($seq = pev_0 \dots pev_i$, where claim holds for $pev_0 \dots pev_{i-1}$)**:** We show this via case distinction.

Case distinction $pev_i = \mathbf{W}_t^{tr''}[Var']$**:**
Let pev_r be the read of tr' and pev_w be the write of tr. Trivially pev_i is not in between them as it is appended at the end and $tr' \neq tr_r$. Also, the order of all other elements is identical to $ph \cdot pev_0 \dots pev_{i-1}$ and $(tr, tr', x) \in (ph \cdot seq).RF$.

Case distinction $pev_i = \mathbf{R}_t^{tr''}[Var']$**:**
A read does not write, and thus it cannot be read by tr'. Also, the order of all other elements is identical to $ph \cdot pev_0 \dots pev_{i-1}$ and $(tr, tr', x) \in (ph \cdot seq).RF$.
□

Lemma 13 (Conditions for interruptible rf-elements)**.** *Given a candidate ph_c, an rf-element $(tr, tr', x) \in ph_c.RF$ with $tr' \neq tr_r$ is interruptible iff*

$$\begin{gathered} \exists tr'' \in Tr : unfin_{ph_c}(tr'') \wedge pev_{ph_c}^{tr,wr} <_{ph_c} pev_{ph_c}^{tr'',rd} <_{ph_c} pev_{ph_c}^{tr',rd} \\ \vee \\ \left(\begin{gathered} \neg(\exists tr'' \in Tr : fin_{ph_c}(tr'') \wedge pev_{ph_c}^{tr',rd} <_{ph_c} pev_{ph_c}^{tr'',wr}) \\ \wedge \\ \exists t \in T : pev_{ph_c}^{t,ls} <_{ph_c} pev_{ph_c}^{tr',rd} \end{gathered} \right). \end{gathered}$$

Proof. We need to show that for $tr \neq tr_r$ it holds that

$$\begin{gathered}
\exists seq \in PEv^*, \exists ph_{c,ins} \in ins(ph_c, seq) \exists tr'' \in Tr : (tr'', tr', x) \in ph_{c,ins}.RF \wedge tr'' \neq tr \\
\leftrightarrow \\
\left(\begin{gathered} \neg(\exists tr'' \in Tr : fin_{ph_c}(tr'') \wedge pev_{ph_c}^{tr',rd} <_{ph_c} pev_{ph_c}^{tr'',wr}) \\ \wedge \\ \exists t \in T : pev_{ph_c}^{t,ls} <_{ph_c} pev_{ph_c}^{tr',rd} \end{gathered} \right) \\
\vee \\
\exists tr'' \in Tr : unfin_{ph_c}(tr'') \wedge pev_{ph_c}^{tr,wr} <_{ph_c} pev_{ph_c}^{tr'',rd} <_{ph_c} pev_{ph_c}^{tr',rd}
\end{gathered} .$$

→:
We show the contraposition. Assume the right-hand side does not hold. We show the left-hand side does not hold as well. Note this is the negation of the right-hand side:

$$\begin{gathered}
\left(\begin{gathered} (\exists tr'' \in Tr : fin_{ph_c}(tr'') \wedge pev_{ph_c}^{tr',rd} <_{ph_c} pev_{ph_c}^{tr'',wr}) \\ \vee \\ \neg(\exists t \in T : pev_{ph_c}^{t,ls} <_{ph_c} pev_{ph_c}^{tr',rd}) \end{gathered} \right) \\
\wedge \\
\neg(\exists tr'' \in Tr : unfin_{ph_c}(tr'') \wedge pev_{ph_c}^{tr,wr} <_{ph_c} pev_{ph_c}^{tr'',rd} <_{ph_c} pev_{ph_c}^{tr',rd}).
\end{gathered}$$

We apply the distributive law which leads to

$$\begin{gathered}
\left(\begin{gathered} (\exists tr'' \in Tr : fin_{ph_c}(tr'') \wedge pev_{ph_c}^{tr',rd} <_{ph_c} pev_{ph_c}^{tr'',wr}) \\ \wedge \\ \neg(\exists tr'' \in Tr : unfin_{ph_c}(tr'') \wedge pev_{ph_c}^{tr,wr} <_{ph_c} pev_{ph_c}^{tr'',rd} <_{ph_c} pev_{ph_c}^{tr',rd}) \end{gathered} \right) \\
\vee \\
\left(\begin{gathered} \neg(\exists t \in T : pev_{ph_c}^{t,ls} <_{ph_c} pev_{ph_c}^{tr',rd}) \\ \wedge \\ \neg(\exists tr'' \in Tr : unfin_{ph_c}(tr'') \wedge pev_{ph_c}^{tr,wr} <_{ph_c} pev_{ph_c}^{tr'',rd} <_{ph_c} pev_{ph_c}^{tr',rd}) \end{gathered} \right) .
\end{gathered}$$

We do a case distinction over both parts of the above disjunction, and show each implies that the left-hand side is false.

Part 1:
There is no unfinished transaction in between tr and tr', so a sequence must contain a read that is inserted in between them. This is not possible as a new read must be inserted after the last write, which is ordered behind the read of tr', which stays unchanged over all extensions.

Part 2:
There is no unfinished transaction in between tr and tr', so a sequence must contain a read that is inserted in between them. This is not possible as a new read must be inserted after the last event of its thread, which for all threads is ordered behind the read of tr', which stays unchanged over all extensions.

$\leftarrow$:
We apply the distributive law, and show that both parts of the disjunction imply the left-hand side.

$\boldsymbol{\exists tr'' \in Tr : unfin_{ph_c}(tr'') \wedge pev_{ph_c}^{\mathbf{tr,wr}} <_{ph_c} pev_{ph_c}^{tr'',\mathbf{rd}} <_{ph_c} pev_{ph_c}^{\mathbf{tr',rd}}}$ **:**
Let $tr_{ri} = tr''$, let the sequence seq contain only a write of tr'' with a write set containing only x, there exists one valid insertion of the write, directly after the read of tr_{ri}, and thus trivially $(tr'', tr', x) \in ph_{c,ins}.RF$ holds and also by assumption $tr'' \neq tr$.

$\boldsymbol{\neg(\exists tr'' \in Tr : fin_{ph_c}(tr'') \wedge pev_{ph_c}^{tr',\mathbf{rd}} <_{ph_c} pev_{ph_c}^{\mathbf{tr'',wr}}) \wedge \exists t \in T : pev_{ph_c}^{t,\mathbf{ls}} <_{ph_c} pev_{ph_c}^{tr',\mathbf{rd}}}$:

Let the sequence seq contain an arbitrary read and a write on x of a transaction tr'' of a thread that has no event after the read of tr'. This thread exists because of the second part of the conjunction. Note that tr'' cannot be tr'. The read can be inserted in between tr and tr' as the last write in the candidate is not ordered after the read of tr' by the second part of the disjunction. The write then is in between tr and tr'; thus, $(tr'', tr', x) \in ph_{c,ins}.RF$ and $tr'' \neq tr$.

□

Lemma 12 (Conditions for fixed rf-elements in candidates)**.** *Given a serial p-history ph_c, two arbitrary transactions tr, tr' and an arbitrary variable x, it holds for all $rf = (tr, tr', x)$ s.t. $rf \in ph_c.RF$:*

$$(tr, tr', x) \in ph_c.RF_{fix}$$
$$\leftrightarrow$$
$$\neg int_{ph_c}(rf) \wedge tr' \neq tr_r.$$

Proof. We show both directions separately.
Direction $\rightarrow$:

$$(tr, tr', x) \in ph_c.RF_{fix}$$
$$\rightarrow$$
$$\neg int_{ph_c}(rf) \wedge tr' \neq tr_r$$

We show that the contraposition holds, which is the following:

$$(int_{ph_c}(rf) \vee tr' = tr_r) \rightarrow (tr, tr', x) \notin ph.RF_{fix}.$$

We show that $int_{ph_c}(rf) \rightarrow (tr, tr', x) \notin ph.RF_{fix}$, and then show that $tr' = tr_r \rightarrow (tr, tr', x) \notin ph.RF_{fix}$, which together imply the contraposition.

$\mathbf{int}_{\boldsymbol{ph_c}}(rf) \rightarrow (\boldsymbol{tr}, \boldsymbol{tr'}, \mathbf{x}) \notin \boldsymbol{ph.RF}_{\mathbf{fix}}$:
It has to hold that

$$\exists seq \in PEv^*, \exists ph_{c,ins} \in ins(ph_c, seq) : (tr, tr', x) \notin ph_{c,ins}.RF.$$

By Lemma 13, $int_{\boldsymbol{ph_c}}(rf)$ is equivalent to

$$\neg(\exists tr'' \in Tr : fin_{ph_c}(tr'') \wedge pev_{ph_c}^{tr',rd} <_{ph_c} pev_{ph_c}^{tr'',wr})$$
$$\vee$$
$$\exists tr'' \in Tr : unfin_{ph_c}(tr'') \wedge pev_{ph_c}^{tr,wr} <_{ph_c} pev_{ph_c}^{tr'',rd} <_{ph_c} pev_{ph_c}^{tr',rd}.$$

In the first case, of the disjunction, there exists no write (including a write event of tr itself) after the read event of tr. Let seq be a sequence s.t. it contains only a arbitrary read and write on x of a new transaction tr''. Then, the read of tr'' can be inserted by ins in between $pev_{ph_c}^{tr,wr}$ and $pev_{ph_c}^{tr',rd}$ as tr'' is concurrent to tr' and the last write of ph_c is ordered before the read event of tr'. The write is then inserted directly after the read and thus also in between $pev_{ph_c}^{tr,wr}$ and $pev_{ph_c}^{tr',rd}$. Then, $(tr'', tr', x) \in ph_{c,ins}.RF$ holds, which is mutually exclusive with $(tr, tr', x) \in ph_{c,ins}.RF$. Thus, $(tr, tr', x) \notin ph_c.RF_{fix}$ holds.

In the second case, of the disjunction, there exists a read event of an unfinished transaction tr'' between the write of tr and the read tr'. Let seq be a sequence s.t. it contains only a write event on x of tr''. Then $(tr'', tr', x) \in ph_{c,ins}.RF$ holds which is mutually exclusive with $(tr, tr', x) \in ph_{c,ins}.RF$; thus, $(tr, tr', x) \notin ph_c.RF_{fix}$.

$\boldsymbol{tr' = tr_{\mathrm{r}} \rightarrow (tr, tr', \mathrm{x}) \notin ph.RF_{\mathrm{fix}}}$:
It has to hold that

$$\exists seq \in PEv^*, \exists ph_{c,ins} \in ins(ph_c, seq) : (tr, tr_r, x) \notin ph_{c,ins}.RF.$$

Consider a sequence seq s.t. that it contains exactly one empty read and one write on x of a transaction tr'' not contained in ph_c, then there exists an insertion $ph_{c,ins} \in ins(ph_c, seq)$ where the read is appended at the end of ph_c and the write directly after, and thus $(tr'', tr_r, x) \in ph_{c,ins}.RF$ holds which means $(tr, tr_r, x) \notin ph_{c,ins}.RF$.

Direction $\leftarrow$:

$$\neg int_{ph_c}(rf) \wedge tr' \neq tr_r \\ \rightarrow \\ (tr, tr', x) \in ph_c.RF_{fix}$$

It has to hold that if $\neg int_{ph_c}(rf) \wedge tr' \neq tr_r$, then

$$\forall seq \in PEv^*, \forall ph_{c,ins} \in ins(ph_c, seq) : (tr, tr', x) \in ph_{c,ins}.RF.$$

We prove via induction that if $\neg int_{ph_c}(rf) \wedge tr' \neq tr_r$, then for any p-event sequence seq it holds that

$$\neg int_{ph_{c,ins}}(rf) \wedge tr' \neq tr_r \wedge, \forall ph_{c,ins} \in ins(ph_c, seq) : (tr, tr', x) \in ph_{c,ins}.RF.$$

We show this via induction over $seq = pev_0 \dots pev_n$, $n \in \mathbb{N}$.

Induction start ($seq = \epsilon$**):** Trivially true.

Induction step ($seq = pev_0 \dots pev_i$, where claim holds for $pev_0 \dots pev_{i-1}$**):** Let $ph_c \in ins(ph_c, pev_0 \dots pev_{i-1})$ and $ph_{c,ins} \in ins(ph_c, pev_i)$.

Case distinction $pev_i = \mathbf{W}_t^{tr''}[Var']$**:**
As $\neg(int_{ph_c}(rf))$ and also $tr' \neq tr_r$ holds, there is no unfinished transaction in between tr and tr'. Thus, the write is not inserted in between them and cannot interrupt the rf. Also, $tr' \neq tr_r$ holds, and $\neg int_{ph_{c,ins}}(rf)$ holds as the order of all events of ph_c is the same in $ph_{c,ins}$.

Case distinction $pev_i = \mathbf{R}_t^{tr''}[Var']$:

In this case, the read is inserted somewhere after the last write event of ph_c. As discussed in the previous case, a read cannot be read. Thus, $rf \in ph_{c,ins}.RF$ holds. Also, $tr' \neq tr_r$ holds, and $\neg(int_{ph_{c,ins}}(rf))$ holds as the order of all events of ph_c is the same in $ph_{c,ins}$ and the read cannot be inserted between the read of tr' and the write of tr. This is because tr'' is real-time ordered with the transaction writing after the read of tr', which exists by Lemma 13.
□

Lemma 14 (Removal function correctness)**.** *Given a p-history or candidate ph and a set of transactions Tr^-, it holds that*

$$(ph\backslash Tr^-).RF = ph.RF\backslash\{rf \in ph \mid \exists tr \in Tr^- : tr \in rf\}.$$

Proof. Let $ph = pev_0 \dots pev_n$ and $ph\backslash Tr^- = pev'_0 \dots pev'_n$ hold. We show both subset or equal relations.

$\boldsymbol{(ph\backslash Tr^-).RF \subseteq ph.RF\backslash\{rf \in ph \mid \exists tr \in Tr^- : tr \in rf\} :}$
Consider an arbitrary rf-element $rf = (tr, tr', x)$ of $(ph\backslash Tr^-).RF$. We show that $rf \in ph.RF$ and $rf \notin \{rf \in ph \mid \exists tr \in Tr^- : tr \in rf\}$.
Let pev'_i be the write event of tr and pev'_k be the read event of tr', it holds that $i < k$. Assume $tr \in Tr^-$ then $pev'_i = \epsilon$ which is a contradiction to rf existing in $(ph\backslash Tr^-).RF$. The same holds for the case $tr' \in Tr^-$.
There are thus events pev_i of $tr \notin Tr^-$ and pev_k of $tr' \notin Tr^-$ in ph with x in their respective write or read set as the removal function does not add elements to write or read sets. As there is no write on x in between pev'_i and pev'_k in $ph\backslash Tr^-$, there can be no write on x of a transaction $tr'' \notin Tr^-$ in between pev_i and pev_k in ph. Assume there is a write of x of a transaction in Tr^- in between. Let tr'' be the transaction of the write last before pev_k. Thus, $tr'' \in Tr^-$ and $(tr'', tr', x) \in ph.RF$ holds. Then by definition of the removal function, pev'_k does not contain x which is a contradiction. So, $rf \in ph.RF$ holds.

$\boldsymbol{ph.RF\backslash\{rf \in ph \mid \exists tr \in Tr^- : tr \in rf\} \subseteq (ph\backslash Tr^-).RF :}$
Let pev_i be the write event of tr and pev_k be the read event of tr'. It holds that $i < k$ and there is no write event on x in between. As $tr \notin Tr^-$ $pev'_i = pev_i$ and as $tr' \notin Tr^-$ as well, pev'_k is a read of tr' with a read set containing x. As the removal function does not add writes or add variables to write sets, there is no write on x in between pev'_i and pev'_k in $ph\backslash Tr^-$, the claim thus holds. □

Lemma 15 (Upper limit of hc-pairs)**.** *The amount of compressed hc-pairs for a given Var and T is finite.*

Proof. Assume a given Var and T. For these parameters we first prove an upper bound to the length of a compressed p-history and a compressed candidate, which are not dependent on the history compressed but only on Var and T. Then, we determine how many pairwise different p-histories of a length up to and equal to this upper bound exist, not taking into account any semantical constraints. Finally, we determine how many pairwise different interrupting write sets exist, and show how many pairwise different elements exists for the set containing interrupting write sets of a compression. We then combine these facts for the overall upper bound.

Upper bound for history length:
Assume an arbitrary hc-pair $hc = (ph, ph_c)$. We give an upper bound to how many transactions are not removed by the compression function. The compression function removes all transactions tr s.t. $tr \in fix(hc)$, which is the case if all of the following three conditions are given:

1. $fin_{ph}(tr)$ (this is equivalent to $fin_{ph_c}(tr)$),
2. $\forall rf \in ph.RF : tr \in rf \rightarrow fix_{ph}(rf)$
3. and $\forall rf \in ph_c.RF : tr \in rf \rightarrow fix_{ph_c}(rf) \vee int_{ph_c}(rf)$.

We upper bound the amount of transactions not fulfilling these conditions meaning they fulfil one of these conditions:

1. $\neg(fin_{ph}(tr))$ (this is equivalent to $\neg(fin_{ph_c}(tr))$),
2. $\neg(\forall rf \in ph.RF : tr \in rf \rightarrow fix_{ph}(rf))$
3. and $\neg(\forall rf \in ph_c.RF : tr \in rf \rightarrow fix_{ph_c}(rf) \vee int_{ph_c}(rf))$.

We upper bound the set of transactions fulfilling each condition separately.

$\{\boldsymbol{tr} \mid \neg(\boldsymbol{fin_{ph}}(\boldsymbol{tr}))\}$ **:**
This is equivalent to $\{tr \mid \mathit{unfin}_{ph}(tr)\}$. There are at most $|T|$ transactions in this set as one thread cannot have multiple unfinished transactions

$$|\{tr \mid \neg(\mathit{fin}_{ph}(tr))\}| \leq |T|.$$

$\{\boldsymbol{tr} \mid \neg(\forall \boldsymbol{rf} \in \boldsymbol{ph.RF} : \boldsymbol{tr} \in \boldsymbol{rf} \rightarrow \mathbf{fix}_{\boldsymbol{ph}}(\boldsymbol{rf}))\}$ **:**
We simplify the predicate for the set and use Lemma 11 on it.

$$\begin{aligned}
&\neg(\forall rf = (tr'', tr', x), rf \in ph.RF : tr \in rf \rightarrow fix_{ph}(rf))\\
\leftrightarrow&\exists rf = (tr'', tr', x), rf \in ph.RF : tr \in rf \wedge \neg(fix_{ph}(rf))\\
\leftrightarrow&\exists rf = (tr'', tr', x), rf \in ph.RF : tr \in rf \wedge tr_r = tr' \qquad |(\mathit{Lemma}\ 11)
\end{aligned}$$

The transaction tr_r can read from one transaction for each variable in Var. Thus, $|\mathit{Var}|$ is an upper bound to the transactions whose writes can be read by tr_r in a p-history (as else transactions with identical write sets overwrite each other). Thus, it holds that

$$|\{tr \mid \neg(\forall rf \in ph.RF : tr \in rf \rightarrow fix_{ph}(rf))\}| \leq |Var|.$$

$\{\boldsymbol{tr} \mid \neg(\forall \boldsymbol{rf} \in \boldsymbol{ph.RF} : \boldsymbol{tr} \in \boldsymbol{rf} \rightarrow \mathbf{fix}_{\boldsymbol{ph_c}}(\boldsymbol{rf}) \vee \mathbf{int}_{\boldsymbol{ph_c}}(\boldsymbol{rf}))\}$ **:**
We simplify the predicate for the set and use Lemma 12 on it.

$$\begin{aligned}
&\neg(\forall rf = (tr'', tr', x), rf \in ph.RF : tr \in rf \rightarrow fix_{ph_c}(rf) \vee int_{ph_c}(rf))\\
\leftrightarrow\ &\exists rf = (tr'', tr', x), rf \in ph.RF : tr \in rf \wedge \neg(fix_{ph_c}(rf) \vee int_{ph_c}(rf))\\
\leftrightarrow\ &\exists rf = (tr'', tr', x), rf \in ph.RF : tr \in rf \wedge \neg fix_{ph_c}(rf) \wedge \neg int_{ph_c}(rf))\\
\leftrightarrow\ &\exists rf = (tr'', tr', x), rf \in ph.RF : tr \in rf \wedge tr_r = tr' \qquad |(\mathit{Lemma}\ 11)
\end{aligned}$$

By the same argument as in the previous case, it holds that

$$|\{tr \mid \neg(\forall rf \in ph.RF : tr \in rf \rightarrow fix_{ph_c}(rf) \vee int_{ph_c}(rf)))\}| \leq |Var|.$$

Thus, overall the amount of events in the p-history or candidate of a compressed hc-pair representation of an arbitrary hc-pair (with the given T and Var) is upper bounded by

$$2 \cdot (|T| + |Var|).$$

Upper bound for p-histories of Length $2 \cdot (|T| + |Var|)$:
For each event there exist $2^{|Var|} \cdot |T|$ many possibilities as it is defined by a write set or read set and a thread. Thus, overall there are

$$2^{|Var| \cdot 2 \cdot (|T| + |Var|)} \cdot |T|^{2 \cdot (|T| + |Var|)}$$

many unique p-histories of of length $2 \cdot (|T| + 2^{|Var|})$ or lower. The lower length p-histories are included in this calculation as any transaction with an empty read and write set can be considered as non-existent.

Upper bound of pairwise different interrupting write sets for a compression:
Similar to the possibilities for read and write sets there are $2^{|Var|}$ pairwise different interrupting write sets. In a compression there can be at most $|T|$ unfinished transactions thus the upper bound of all pairwise different sets containing interrupting write sets of a compression is

$$2^{|Var| \cdot |T|}.$$

Combining the facts:
Overall the amount of compressed hc-pair representations is upper bounded by

$$2^{2|Var|\cdot(|T|+|Var|)} \cdot |T|^{2\cdot(|T|+|Var|)} \cdot 2^{2|Var|\cdot(|T|+|Var|)} \cdot |T|^{2\cdot(|T|+|Var|)} \cdot 2^{|Var|\cdot|T|},$$

and thus the amount is finite. $\square$

Lemma 16 (Compression represents an equivalence class)**.** *Given two arbitrary hc-pairs* (ph, ph_c) *and* (ph', ph'_c)*, it holds that*

$$cmp(ph, ph_c) = cmp(ph', ph'_c) \rightarrow (ph, ph_c) \equiv_{ext} (ph', ph'_c).$$

Proof. It is to prove that if $cmp(ph, ph_c) = cmp(ph', ph'_c)$ holds, then it holds for any arbitrary p-event sequence seq that

$$\exists ph_{c,ins} \in ins(ph_c, seq) : ph \cdot seq \equiv ph_{c,ins} \leftrightarrow \exists ph'_{c,ins} \in ins(ph'_c, seq) : ph' \cdot seq \equiv ph'_{c,ins}.$$

If $cmp(ph, ph_c) = cmp(ph', ph'_c) = \mathbf{DM}$, then in the extension of both hc-pairs by any sequence the respective p-history is not equivalent to its respective candidate. This is because they both contain mutually exclusive rf-elements between their p-history and candidate, which are all fixed in the p-history and fixed or interruptible in the candidate. As we argued in the respective section in the thesis, both extended hc-pairs are then not consistent and the claim holds true.

We will now prove the lemma for when this is not the case. We will show this by showing two statements and then proving why these imply the lemma.

1. Given an arbitrary hc-pair $hc = (ph, ph_c)$, it holds that

$$hc \text{ is consistent } \leftrightarrow cmp(hc) \text{ is consistent.} \tag{B.1}$$

2. For any two hc-pairs $hc = (ph, ph_c), hc' = (ph', ph'_c)$ where $cmp(hc) =$

$cmp(hc')$ and an arbitrary p-event pev, it holds that

$$\begin{aligned}&\forall ph_{c,ins} \in ins(ph_c, pev)\exists ph'_{c,ins} \in ins(ph'_c, pev) :\\ &cmp(ph \cdot pev, ph_{c,ins}) = cmp(ph' \cdot pev, ph'_{c,ins}).\end{aligned} \tag{B.2}$$

To declutter the overall proof, these statements are proven in separate propositions. Equation (B.1) in Proposition 5 and Equation (B.2) in Proposition 6 for the case where pev is a read and in Proposition 7 for the case pev is a write.

We show that Equation (B.2) implies the following For any two hc-pairs $hc = (ph, ph_c)$, $hc' = (ph', ph'_c)$ where $cmp(hc) = cmp(hc')$ and an arbitrary sequence of p-events seq, it holds that

$$\begin{aligned}&\forall ph_{c,ins} \in ins(ph_c, seq)\exists ph'_{c,ins} \in ins(ph'_c, seq) :\\ &cmp(ph \cdot seq, ph_{c,ins}) = cmp(ph' \cdot seq, ph'_{c,ins}).\end{aligned}$$

We show this via induction over the sequence seq.

Induction start seq = *pev*:
This directly follows from Equation (B.2).

Induction step seq → seq · *pev* :
There exists $ph_{c1,ins} \in ins(ph_c, seq)$ for which by induction statement then there also exists $ph'_{c1,ins} \in ins(ph'_c, seq)$ s.t. $cmp(ph \cdot seq, ph_{c1,ins}) = cmp(ph \cdot seq, ph'_{c1,ins})$. It follows by Equation (B.2) that

$$\begin{aligned}&\forall ph_{c,ins} \in ins(ph_{c1,ins}, pev)\exists ph'_{c,ins} \in ins(ph'_{c1,ins}, pev) :\\ &cmp(ph \cdot seq \cdot pev, ph_{c,ins}) = cmp(ph' \cdot seq \cdot pev, ph'_{c,ins}).\end{aligned}$$

This concludes the proof by induction.

From this result it trivially follows that for any two hc-pairs $hc = (ph, ph_c), hc' =$

(ph', ph_c') s.t. $cmp(ph, ph_c) = cmp(ph', ph_c')$ it holds that

$$\begin{aligned}&\exists ph_{c,ins} \in ins(ph_c, seq) : ph \cdot seq \equiv ph_{c,ins}\\ &\rightarrow \exists ph'_{c,ins} \in ins(ph_c', seq) : cmp(ph \cdot seq, ph_{c,ins}) = cmp(ph' \cdot seq, ph'_{c,ins}).\end{aligned}$$

We show that the left-hand side of this statement then also implies $ph' \cdot seq \equiv ph'_{c,ins}$. If $(ph \cdot seq, ph_{c,ins})$ is consistent, then from Equation (B.1) it follows that $cmp(ph \cdot seq, ph_{c,ins})$ is consistent as well. Thus, $cmp(ph' \cdot seq, ph'_{c,ins})$ is also consistent as the compressions are equal, which implies that $(ph' \cdot seq, ph'_{c,ins})$ is consistent, and thus $ph' \cdot seq \equiv ph'_{c,ins}$. From this the left to right direction of the overall claim follows:

$$\exists ph_{c,ins} \in ins(ph_c, seq) : ph \cdot seq \equiv ph_{c,ins} \rightarrow \exists ph'_{c,ins} \in ins(ph_c', seq) : ph' \cdot seq \equiv ph'_{c,ins}.$$

We can show the reverse direction of this equation analogue. Combining both shows the lemma to be true which concludes the proof. □

Proposition 5 (HC-Pair consistency equivalent to compression consistency)**.** *It holds for any arbitrary hc-pair hc that*

$$hc \textit{ is consistent } \leftrightarrow cmp(hc) \textit{ is consistent.}$$

Proof. We prove both directions separately.

Direction →:

If hc is consistent, then $ph.RF = ph_c.RF$ holds, meaning $(ph \backslash fix(hc)).RF = (ph_c \backslash fix(hc)).RF$ (follows from Lemma 14) from which it follows that $cmp(hc)$ is consistent.

Direction ←:

We show the contraposition, which is

$$hc \text{ is not consistent } \rightarrow cmp(hc) \text{ is not consistent.}$$

Let $rf = (tr, tr', x)$ and $rf' = (tr'', tr', x)$, $tr \neq tr''$ be the mutually exclusive rf-elements in hc. If rf is fixed in ph and rf' is fixed or interruptible in ph_c, then $cmp(hc) = \mathbf{DM}$. If both are not fixed or interruptible in either, then $tr, tr', tr'' \notin fix(hc)$ meaning that by Lemma 14 rf occurs in $ph \backslash fix(hc)$ and rf' occurs in $ph_c \backslash fix(hc)$, meaning $cmp(hc)$ is not consistent as these elements are mutually exclusive.

□

Proposition 6. *Given a p-event $pev = \mathbf{R}_t^{tr}[V]$, two arbitrary hc-pairs (ph, ph_c) and (ph', ph'_c) s.t. $cmp(ph, ph_c) = cmp(ph', ph'_c)$, it holds that*

$$\forall ph_{c,ins} \in ins(ph_c, pev) \exists ph'_{c,ins} \in ins(ph'_c, pev) : \\ cmp(ph_c \cdot pev, ph_{c,ins}) = cmp(ph'_c \cdot pev, ph'_{c,ins}).$$

Proof. Given the construction of ins, let n be the index s.t.

$$ph_{c,ins} = st(ph_c) \cdot add(en(ph_c), pev, n).$$

Let $hc = (ph, ph_c)$ and $hc' = (ph', ph'_c)$. Recall that

$$\begin{array}{rcl} cmp(ph, ph_c) & = & (ph \backslash fix(hc), ph_c \backslash fix(hc), IWS_{ph_c}) \\ = \quad cmp(ph', ph'_c) & = & (ph' \backslash fix(hc'), ph'_c \backslash fix(hc'), IWS_{ph'_c}). \end{array}$$

Thus, note that $en(ph_c) = en(ph'_c)$ as after the last write there are only unfinished transactions consisting of a read event which are neither in $fix(hc)$ or $fix(hc')$ and any transaction writing to them is also read by tr_r and thus not in $fix(hc)$. Thus, the removal function does not modify the events in $en(ph_c)$ and $en(ph'_c)$. This makes the following a valid result of the insertion function for ph'_c:

$$ph'_{c,ins} = st(ph'_c) \cdot add(en(ph_c), pev, n).$$

We show that for this specific assignment

$$hc_{+} = cmp(ph_{c} \cdot pev, ph_{c,ins}) = cmp(ph'_{c} \cdot pev, ph'_{c,ins}) = hc'_{+}$$

holds true.

Case distinction $\boldsymbol{cmp(ph \cdot pev, ph_{c,ins}) = \mathbf{DM}}$:
By definition of *cmp*, it holds that

$$\begin{aligned}\exists rf \in (ph \cdot pev).RF, \exists rf' \in ph_{c,ins}.RF : \quad & mutex(rf, rf') \wedge fix_{ph \cdot pev}(rf) \\ & \wedge(fix_{ph_{c,ins}}(rf') \vee int_{ph_{c,ins}}(rf'))\end{aligned}$$

As $cmp(ph, ph_{c}) \neq \mathbf{DM}$,

$$\begin{aligned}\neg(\exists rf \in (ph \backslash fix(hc)).RF, \exists rf' \in ph_{c}.RF : \quad & mutex(rf, rf') \wedge fix_{ph}(rf) \\ & \wedge(fix_{ph_{c}}(rf') \vee int_{ph_{c}}(rf'))\end{aligned}.$$

Let rf and rf' be mutually exclusive rf-elements in $ph \cdot pev$ and $ph_{c,ins}$, respectively. Let x be the variable in rf and rf'. Both must involve tr as the reader as the appending/inserting of pev created the mutually exclusive rf-elements and there were no such rf-elements in hc. So, there exist $tr', tr'' \in Tr$ with $tr' \neq tr''$ s.t. $rf = (tr', tr, x)$ and $rf' = (tr'', tr, x)$. Note that pev is ordered after the last write in both $ph \cdot pev$ and $ph_{c,ins}$. It reads x from tr' in $ph \cdot pev$ and x from tr'' in $ph_{c,ins}$. This means the write on x of tr' and tr'' is the last write on x in $ph \cdot pev$ and $ph_{c,ins}$, respectively. These writes were also present in ph and ph_{c}, which are identical except for pev to $ph \cdot pev$ and $ph_{c,ins}$, respectively. Thus, it holds that

$$(tr', tr_{r}, x) \in ph.RF \wedge (tr'', tr_{r}, x) \in ph_{c}.RF.$$

This implies that $tr', tr'' \notin fix(hc)$ and as $cmp(hc) = cmp(hc')$ also that $tr', tr'' \notin$

$fix(hc')$. Thus, it holds that

$$(tr', tr_r, x) \in ph'.RF \wedge (tr'', tr_r, x) \in ph'_c.RF.$$

As *pev* is a read, it holds that

$$(tr', tr, x) \in (ph' \cdot pev).RF \wedge (tr'', tr, x) \in ph'_{c,ins}.RF.$$

So finally, it follows that

$$cmp(hc'_+) = \mathbf{DM},$$

which proves the claim.

Case distinction $\boldsymbol{cmp(ph \cdot pev, ph_{c,ins}) \neq}$ DM :
The proof is divided into two parts. First, we show that

$$((ph \cdot pev)\backslash fix(hc_+), (ph_{c,ins}\backslash fix(hc_+))) = ((ph' \cdot pev)\backslash fix(hc'_+), (ph'_{c,ins}\backslash fix(hc'_+))). \tag{B.3}$$

Second, we show that

$$IWS_{ph_{c,ins}} = IWS_{ph'_{c,ins}}. \tag{B.4}$$

Equation (B.3):
Let $hc_+ = (ph \cdot pev, ph_{c,ins})$. Let $hc'_+ = (ph' \cdot pev, ph'_{c,ins})$. We first show that

1. $fix(hc) = fix(hc_+)$
2. and $fix(hc') = fix(hc'_+)$.

Finally, we show that this implies

1. $(ph \cdot pev)\backslash fix(hc_+) = (ph' \cdot pev)\backslash fix(hc'_+)$
2. and $ph_{c,ins}\backslash fix(hc_+) = ph'_{c,ins}\backslash fix(hc'_+)$.

Both of these statements together imply Equation (B.3).

$\mathbf{fix}(\mathbf{hc}) = \mathbf{fix}(\mathbf{hc}_+)$:
This is shown in Proposition 8.

$\mathbf{fix}(\mathbf{hc'}) = \mathbf{fix}(\mathbf{hc'_+})$:
This is shown in Proposition 8.

$(\boldsymbol{ph} \cdot \boldsymbol{pev})\backslash\mathbf{fix}(\mathbf{hc}) = (\boldsymbol{ph'} \cdot \boldsymbol{pev})\backslash\mathbf{fix}(\mathbf{hc'})$:
Any transaction read by *pev* is also read by tr_r in both *ph* and ph', and is thus not in $fix(hc)$ or $fix(hc')$. Thus, *pev* is not affected by a removal of the transactions in $fix(hc)$ or $fix(hc')$. It also holds that $fix(hc) = fix(hc_+)$ and $fix(hc') = fix(hc'_+)$ from which we can deduce the following:

$$\begin{array}{rlcl} & (ph \cdot pev)\backslash fix(hc_+) & = & (ph' \cdot pev)\backslash fix(hc'_+) \\ \leftrightarrow & (ph \cdot pev)\backslash fix(hc) & = & (ph' \cdot pev)\backslash fix(hc') \\ \leftrightarrow & ph\backslash fix(hc) \cdot pev & = & ph'\backslash fix(hc') \cdot pev. \end{array}$$

By assumption, $ph\backslash fix(hc) = ph'\backslash fix(hc')$ is true. This means the last line holds true and thus also

$$(ph \cdot pev)\backslash fix(hc) = (ph' \cdot pev)\backslash fix(hc').$$

$ph_{c,ins}\backslash fix(hc_+) = ph'_{c,ins}\backslash fix(hc'_+)$:
Note that

$$\begin{array}{r} ph_{c,ins} = st(ph_c) \cdot add(en(ph_c), pev, n) \\ \text{and } ph'_{c,ins} = st(ph'_c) \cdot add(en(ph_c), pev, n). \end{array}$$

Given the following facts:

1. $add(en(ph_c), pev, n))$ contains no transaction in $fix(hc_+)$ and only unfin-

ished transactions which are exclusively part in interruptible rf-elements,

2. $add(en(ph'_c), pev, n))$ contains no transaction in $fix(hc'_+)$ and only unfinished transactions which are exclusively part in interruptible rf-elements,
3. $fix(hc_+) = fix(hc)$
4. and $fix(hc'_+) = fix(hc')$,

it follows that

$$\begin{aligned} & ph_{c,ins} \backslash fix(hc_+) \\ = \;& (st(ph_c) \cdot add(en(ph_c), pev, n)) \backslash fix(hc_+) \\ = \;& (st(ph_c) \backslash fix(hc)) \cdot add(en(ph_c), pev, n) \\ = \;& (st(ph'_c) \backslash fix(hc')) \cdot add(en(ph_c), pev, n) \\ = \;& (st(ph'_c) \cdot add(en(ph_c), pev, n)) \backslash fix(hc'_+) \\ = \;& ph'_{c,ins} \backslash fix(hc'_+). \end{aligned}$$

This proves the claim.

Overall then Equation (B.3) holds true.

Equation (B.4):
Note that by assumption $IWS_{ph_c} = IWS_{ph'_c}$. As *pev* starts a new transaction *tr*, it is in the domain of $IWS_{ph_{c,ins}}$ and $IWS_{ph'_{c,ins}}$. As *pev* is a read, inserting it into ph_c does not change the interrupting write sets of other transactions. Then, $IWS_{ph_{c,ins}}(tr)$ is the set containing all variables read by transactions except tr_r after the occurrence of *pev* in $ph_{c,ins}$. Thus, it holds that

$$IWS_{ph_{c,ins}}(tr') = \begin{cases} IWS_{ph_{c,ins}}(tr) & tr' = tr \\ IWS_{ph_c}(tr') & \text{else} \end{cases}.$$

As *pev* is a read, it is inserted into $en(ph_c)$ which is - as we discussed - above equal to $en(ph'_c)$. Thus, all variables read by transactions except tr_r after the occurrence of pev in $ph_{c,ins}$ are the same for $ph'_{c,ins}$. Thus, $IWS_{ph_{c,ins}}(tr) = IWS_{ph'_{c,ins}}(tr)$. As $IWS_{ph_c} = IWS_{ph'_c}$ holds, this implies

$$IWS_{ph_{c,ins}} = IWS_{ph'_{c,ins}}.$$

□

Proposition 7 (HC-Pair consistency equivalent to compression consistency). *Given a p-event $pev = \mathbf{W}_t^{tr}[V]$, it holds that*

$$\begin{gathered}\forall ph_{c,ins} \in ins(ph_c, pev) \exists ph'_{c,ins} \in ins(ph'_c, pev) : \\ cmp(ph_c \cdot pev, ph_{c,ins}) = cmp(ph'_c \cdot pev, ph'_{c,ins}).\end{gathered}$$

Proof. Let it hold that $hc = (ph, ph_c)$ and $hc' = (ph', ph'_c)$ and $hc_+ = (ph \cdot pev, ph_{c,ins})$. Note that $ins(ph'_c, pev)$ contains exactly one element which we denote $ph'_{c,ins}$. Then, let $hc'_+ = (ph' \cdot pev, ph'_{c,ins})$ hold. We do a case distinction over whether $hc_+ = \mathbf{DM}$ or not.

Case distinction $\boldsymbol{hc_+} = \mathbf{DM}$:
By definition of cmp, it holds that

$$\begin{aligned}\exists rf \in (ph \cdot pev).RF, \exists rf' \in ph_{c,ins}.RF : \quad & mutex(rf, rf') \wedge fix_{ph \cdot pev}(rf) \\ & \wedge (fix_{ph_{c,ins}}(rf') \vee int_{ph_{c,ins}}(rf')).\end{aligned}$$

As $cmp(hc) \neq \mathbf{DM}$:

$$\begin{aligned}\neg(\exists rf \in (ph).RF, \exists rf' \in ph_c.RF : \quad & mutex(rf, rf') \wedge fix_{ph}(rf) \\ & \wedge (fix_{ph_c}(rf') \vee int_{ph_c}(rf'))\end{aligned}.$$

Analogue claims hold true for hc' and hc'_+. Let $rf = (tr_a, tr_b, x)$ and $rf' = (tr_c, tr_b, x)$ be mutually exclusive rf-elements in $ph \cdot pev$ and $ph_{c,ins}$, respectively, s.t. rf is fixed and rf' is either fixed or interruptible. This implies that $tr_b \neq tr_r$

as else both elements would neither be fixed nor interruptible. Note that

$$(ph \cdot pev).RF = ph.RF \cup \{(tr, tr_r, z) \mid z \in WS^{vo}_{ph \cdot pev}(tr)\}.$$

Since $tr_b \neq tr_r$ holds, it follows that $rf \notin \{(tr, tr_r, z) \mid z \in WS^{vo}_{ph \cdot pev}(tr)\}$, which implies

$$rf \in ph.RF.$$

Also, obviously $tr_a \neq tr$ as $pev \notin ph$, which in turn means that $tr_c = tr$ holds. As $pev^{tr_b, rd} \in ph_c$ but $pev \notin ph_c$, there must exists $tr_d \neq tr$ s.t.

$$(tr_d, tr_b, x) \in ph_c.$$

Additionally, as inserting pev interrupts this rf-element, it holds that

$$pev^{tr_d, wr} <_{ph_c} pev^{tr, rd} <_{ph_c} pev^{tr_b, rd}.$$

It follows that

$$x \in IWS_{ph_c}(tr).$$

As it holds that $cmp(hc) = cmp(hc')$, it follows that

$$x \in IWS_{ph'_c}(tr).$$

This means there exist tr'_a, tr'_b s.t.

$$(tr'_a, tr'_b, x) \in ph'_c.RF \text{ s.t. } tr'_a \neq tr, tr'_b \neq tr_r, \text{ and } int_{ph_c}(tr, (tr'_a, tr'_b, x)).$$

It holds that $(tr'_a, tr'_b, x) \in ph'.RF$. This is because if a transaction $tr'_d \neq tr'_a$ exists s.t. $(tr'_d, tr'_b, x) \in ph'.RF$, then (tr'_d, tr'_b, x) and (tr'_a, tr'_b, x) are mutually exclusive and the first is fixed in ph' and the second is interruptible in ph'_c which would make $cmp(hc') = \mathbf{DM}$ which it is not. Thus, it holds that $(tr'_a, tr'_b, x) \in ph' \cdot pev$ and $(tr, tr'_b, x) \in ph'_{c,ins}.RF$ by construction of ins. It holds that (tr'_a, tr'_b, x) is fixed in $ph' \cdot pev$ and (tr, tr'_b, x) is fixed or interruptible in $ph'_{c,ins}.RF$. It follows that

$$cmp(hc'_+) = \mathbf{DM}.$$

Case distinction $\boldsymbol{hc_+} \neq$ DM :
The proof is divided into two parts. First, we show that

$$((ph \cdot pev) \backslash fix(hc_+), (ph_{c,ins} \backslash fix(hc_+))) = ((ph' \cdot pev) \backslash fix(hc'_+), (ph'_{c,ins} \backslash fix(hc'_+))). \tag{B.5}$$

Second, we show that

$$IWS_{ph_{c,ins}} = IWS_{ph'_{c,ins}}. \tag{B.6}$$

Equation (B.5)
We first define the set of transactions that after appending/inserting *pev* to/into *ph*,ph_c,ph' and ph'_c are not being read by tr_r in the resulting p-histories/candidates. We say that *pev* covers these transactions. We first define this for *ph*.

$$\begin{aligned} cov_{ph} = \{tr' \mid \;& pev^{tr',wr}_{ph\cdot pev} <_{ph\cdot pev} pev^{tr,wr}_{ph\cdot pev}, \exists x \in Var : (tr', tr_r, x) \in ph.RF, \\ & \forall x \in Var : (tr', tr_r, x) \in ph.RF \to x \in V\} \end{aligned}$$

For ph_c cov_{ph_c} is defined analogue with $ph_{c,ins}$ as the extended p-history. Also, the set is defined analogue for ph' and ph'_c. We show the following:

1. $cov_{ph} = cov_{ph'}$,
2. $cov_{ph_c} = cov_{ph'_c}$,
3. $fix(hc_+) = fix(hc) \cup (cov_{ph} \cap cov_{ph_c})$
4. and $fix(hc'_+) = fix(hc') \cup (cov_{ph'} \cap cov_{ph'_c})$.

We then show that this implies Equation (B.5).

$\boldsymbol{cov_{ph} = cov_{ph'}}$**:**

We show that $cov_{ph} = cov_{ph\backslash \mathit{fix}(hc)}$. As we use no specific properties that distinguish ph from ph' in our proof it also then holds that $cov_{ph'} = cov_{ph'\backslash \mathit{fix}(hc')}$. Then as $cmp(hc) = cmp(hc')$ holds it follows

$$cov_{ph} = cov_{ph\backslash \mathit{fix}(hc)} = cov_{ph'\backslash \mathit{fix}(hc')} = cov_{ph'}.$$

We show $cov_{ph} = cov_{ph\backslash \mathit{fix}(hc)}$ by showing all three parts of the builder predicate for cov_{ph} are equivalent to their counterparts in $cov_{ph\backslash \mathit{fix}(hc)}$.

Predicate 2:

$$\begin{gathered}\exists \mathrm{x} \in \boldsymbol{Var} : (\boldsymbol{tr'}, \boldsymbol{tr}_{\mathrm{r}}, \mathrm{x}) \in \boldsymbol{ph.RF} \\ \leftrightarrow \\ \exists \mathrm{x} \in \boldsymbol{Var} : (\boldsymbol{tr'}, \boldsymbol{tr}_{\mathrm{r}}, \mathrm{x}) \in (\boldsymbol{ph}\backslash \mathrm{fix(hc)}).\boldsymbol{RF} :\end{gathered}$$

We show the direction from left to right: $\exists x \in Var : (tr', tr_r, x) \in ph.RF$ implies $tr' \notin fix(hc)$. By Lemma 14, this means

$$(tr', tr_r, x) \in (ph\backslash fix(hc)).RF.$$

We show the direction from right to left: Note that $fix(hc)$ by construction does not contain last writers on variables as they are read by tr_r which results in a non-fixed rf-element. So, if tr' is the last writer on x in $ph\backslash fix(hc)$, it is also in ph. Thus, $(tr', tr_r, x) \in ph.RF$ holds.

Predicate 1:

$$\begin{gathered}\boldsymbol{pev}^{\boldsymbol{tr'},\mathbf{wr}}_{\boldsymbol{ph}\cdot\boldsymbol{pev}} <_{\boldsymbol{ph}\cdot\boldsymbol{pev}} \boldsymbol{pev}^{\boldsymbol{tr},\mathbf{wr}}_{\boldsymbol{ph}\cdot\boldsymbol{pev}} \\ \leftrightarrow \\ \boldsymbol{pev}^{\boldsymbol{tr'},\mathbf{wr}}_{\boldsymbol{ph}\backslash \mathbf{fix(hc)}\cdot\boldsymbol{pev}} <_{\boldsymbol{ph}\backslash \mathbf{fix(hc)}\cdot\boldsymbol{pev}} \boldsymbol{pev}^{\boldsymbol{tr},\mathbf{wr}}_{\boldsymbol{ph}\backslash \mathbf{fix(hc)}\cdot\boldsymbol{pev}} :\end{gathered}$$

We only consider the case $tr' \notin fix(hc)$ as else predicate 2 would be false and tr' would not be included in both sets on that count, and thus the case is irrelevant here for showing equality of the sets. This means the write event of tr' exists in both $ph \cdot pev$ and $ph \backslash fix(hc) \cdot pev$. Note that $pev = pev^{tr,wr}_{ph \cdot pev} = pev^{tr,wr}_{ph \cdot pev \backslash fix(hc)}$. Trivially, then per definitionpev is ordered after the write event of tr' in both.

Predicate 3:

$$\begin{gathered}\forall \mathrm{x} \in \boldsymbol{Var} : (\boldsymbol{tr'}, \boldsymbol{tr}_{\mathrm{r}}, \mathrm{x}) \in \boldsymbol{ph.RF} \rightarrow \mathrm{x} \in \mathbf{V} \\ \leftrightarrow \\ \forall \mathrm{x} \in \boldsymbol{Var} : (\boldsymbol{tr'}, \boldsymbol{tr}_{\mathrm{r}}, \mathrm{x}) \in (\boldsymbol{ph} \backslash \mathbf{fix(hc)}).\boldsymbol{RF} \rightarrow \mathrm{x} \in \mathbf{V} :\end{gathered}$$

As we have discussed with Predicate 2, the rf-elements involving tr' and tr_r are the same in ph and $ph \backslash fix(hc)$. The write set V is independent of ph and $ph \backslash fix(hc)$, and thus both statements imply each other.

$\boldsymbol{cov_{ph_c} = cov_{ph'_c}}$**:**
In the previous case for predicate 2 and 3 no property of ph or ph' that distinguishes them from candidates was used. We prove predicate 1. Let $ph_{cf,ins}$ be the only element in $ins(ph_c \backslash fix(hc), pev)$.

Predicate 1:

$$\boldsymbol{pev^{tr',\mathrm{wr}}_{ph_{\mathrm{c,ins}}} <_{ph_{\mathrm{c,ins}}} pev^{tr,\mathrm{wr}}_{ph_{\mathrm{c,ins}}} \leftrightarrow pev^{tr',\mathrm{wr}}_{ph_{\mathrm{cf,ins}}} <_{ph_{\mathrm{cf,ins}}} pev^{tr,\mathrm{wr}}_{ph_{\mathrm{cf,ins}}}} :$$

We only consider the case $tr' \notin fix(hc)$ as else predicate 2 would be false and tr' would not be included in both sets on that count. Thus, the latter case is irrelevant here for showing equality of the sets. This means the write event of tr' exists in ph_c, $ph_c \backslash fix(hc)$, $ph_{c,ins}$ and $ph_{cf,ins}$. Note that the existence of a write of $tr' \notin fix(hc)$ and a read of an unfinished transaction (tr) and

that the order of both p-events is identical in both ph_c and $ph_c \backslash fix(hc)$. We do a case distinction over whether $pev_{ph_c}^{tr',wr} <_{ph_c} pev_{ph_c}^{tr,rd}$ holds or not. If it holds, in both cases *pev* is inserted after the read of tr'. Assume it does not hold, in both cases *pev* is inserted before the read of tr'.
The claim holds.

$\mathbf{fix(hc_+) = fix(hc) \cup (\boldsymbol{cov_{ph}} \cap \boldsymbol{cov_{ph_c}})}$ **:**
We use the following reformulation of $fix(hc)$:

$$fix(hc) = f(ph) \cap f(ph_c), \text{ where}$$

$$f(ph) = \{tr \in Tr \mid fin_{ph}(tr), \forall rf \in ph.RF : tr \in rf \to fix_{ph}(rf)\}$$
and
$$f(ph_c) = \{tr \in Tr \mid fin_{ph}(tr), \forall rf \in ph_c.RF : tr \in rf \to fix_{ph_c}(rf) \vee int_{ph_c}(rf)\}.$$

We show two things:

1. $f(ph \cdot pev) = f(ph) \cup cov_{ph}$
2. and $f(ph_{c,ins}) \backslash \{tr\} = f(ph_c) \cup cov_{ph_c}$.

Note that $f(ph \cdot pev)$ never contains *tr* as it is read by tr_r in $ph \cdot pev$. Then, $f(ph) \cap f(ph_c)$ cannot contain *tr*. Thus, both statements above combined show the claim.

$\mathbf{f(\boldsymbol{ph \cdot pev}) = f(\boldsymbol{ph}) \cup \boldsymbol{cov_{ph}}}$ **and** $\mathbf{f(\boldsymbol{ph' \cdot pev}) = f(\boldsymbol{ph'}) \cup \boldsymbol{cov_{ph'}}}$ **:**
This is shown in Proposition 9.

$\mathbf{f(\boldsymbol{ph_{c,ins}}) \backslash \{\boldsymbol{tr}\} = f(\boldsymbol{ph_c}) \cup \boldsymbol{cov_{ph_c}}}$ **and** $\mathbf{f(\boldsymbol{ph'_{c,ins}}) = f(\boldsymbol{ph'_c}) \cup \boldsymbol{cov_{ph'_c}}}$ **:**
This is shown in Proposition 10.

Now given the previous results we first prove that

$$(ph' \cdot pev) \backslash fix(hc'_+) = (ph \cdot pev) \backslash fix(hc_+),$$

and then that

$$(ph'_{c,ins})\backslash fix(hc'_{+}) = (ph'_{c,ins})\backslash fix(hc_{+}).$$

The first statement is a straightforward transformation using the previous results and that tr is not in $fix(hc)$ and $fix(hc')$, and that pev is a write of a transaction not in $fix(hc)$ and $fix(hc')$, and thus not modified by cmp for $fix(hc)$ and $fix(hc')$.

$$\begin{aligned}
& (ph' \cdot pev)\backslash fix(hc'_{+}) \\
= \;& (ph' \cdot pev)\backslash(fix(hc') \cup (cov_{ph'} \cap cov_{ph'_c}) \\
= \;& (ph' \cdot pev)\backslash(fix(hc') \cup (cov_{ph} \cap cov_{ph_c}) \\
= \;& ((ph'\backslash fix(hc')) \cdot pev)\backslash(cov_{ph} \cap cov_{ph_c}) \\
= \;& ((ph\backslash fix(hc)) \cdot pev)\backslash(cov_{ph} \cap cov_{ph_c}) \\
= \;& (ph \cdot pev)\backslash fix(hc_{+}).
\end{aligned}$$

For the second proof we need to show two things. We need to show that when given $ph_c = pev_0 \ldots pev_n$ and $ph_{\mathbf{c,ins}} = pev_0 \ldots pev \ldots pev_n$ it holds (note hc and hc_{+} are not doomed by assumption) that iff

$$\begin{gathered}
ph_c\backslash fix(hc) = pev'_0 \ldots pev'_n \\
\text{then} \\
ph_{c,ins}\backslash fix(hc) = pev'_0 \ldots pev \ldots pev'_n.
\end{gathered}$$

Also, we need to prove that $ph\backslash(Tr_1 \cup Tr_2) = (ph\backslash Tr_1)\backslash Tr_2$. The proofs for that can be found in Proposition 11 and Proposition 12. We partition $ph_{c,ins}$ and $ph'_{c,ins}$ into two parts one up until but not including pev and one containing the remaining subsequence after pev. These are denoted $p1(ph_{c,ins})$ and $p2(ph_{c,ins})$, analogue for ph'_c. Note that it holds that $p1(ph_{c,ins}) \cdot p2(ph_{c,ins}) =$

ph_c, analogue for ph'_c.

$$\begin{aligned}
& (ph'_{c,ins})\backslash fix(hc'_+) \\
= \; & (p1(ph'_{c,ins}) \cdot pev \cdot p2(ph'_{c,ins}))\backslash fix(hc'_+) \\
= \; & (p1(ph'_{c,ins}) \cdot pev \cdot p2(ph'_{c,ins}))\backslash(fix(hc'))\backslash(cov_{ph'} \cap cov_{ph'_c}) \\
= \; & (p1(ph_{c,ins}) \cdot pev \cdot p2(ph_{c,ins}))\backslash(fix(hc))\backslash(cov_{ph'} \cap cov_{ph'_c}) \text{(Propositions 11 and 12)} \\
= \; & (ph_{c,ins})\backslash fix(hc_+)
\end{aligned}$$

Both transformations prove Equation (B.5).

Equation (B.6)
Note that the domain of $IWS_{ph_{c,ins}}$ is the domain of IWS_{ph_c} without tr as tr is finished in $ph_{c,ins}$. Additionally, the finished status of all other transactions stays identical between ph_c and $ph_{\mathbf{c,ins}}$. The same holds for $IWS_{ph'_{c,ins}}$. Note that any rf-element not involving tr_r is identical in $ph_{c,ins}$ and ph_c as else tr is being read by a transaction that is not tr_r, and thus hc would be **DM**. It is not by assumption. Thus, for all transactions $tr' \neq tr$, $IWS_{ph_{c,ins}}(tr') = IWS_{ph_c}(tr')$. The same applies to $ph'_{\mathbf{c,ins}}$ and ph'_c. As $IWS_{ph_c} = IWS_{ph'_c}$, Equation (B.6) follows. □

Proposition 8. *Given an hc-pair $hc = (ph, ph_c)$ and its extension by an arbitrary read event $pev = \mathbf{R}_t^{tr}[V]$ to $hc_+ = (ph \cdot pev, ph_{c,ins})$ s.t. $ph_{c,ins} \in ins(ph_c, pev)$, it holds that $fix(hc) = fix(hc_+)$.*

Proof. We partition $fix(hc)$ and $fix(hc_+)$ into two subsets based on its builder predicates.

$$\begin{aligned}
& fix(hc) = \\
& \{tr \in Tr \mid fin_{ph}(tr), \forall rf \in ph.RF : tr \in rf \rightarrow fix_{ph}(rf), \} \\
& \qquad\qquad\qquad\qquad\qquad \cap \\
& \{tr \in Tr \mid fin_{ph}(tr), \forall rf \in ph_c.RF : tr \in rf \rightarrow fix_{ph_c}(rf) \vee int_{ph_c}(rf)\}.
\end{aligned}$$

For simplicity, we call the above sets $fix(ph)$ and $fix(ph_c)$. The partition of $fix(hc_+)$ is done and named analogue. Note that the set of finished transactions is identical between ph and ph_c. This also holds for hc and hc_+ as pev is a read.

We first show $fix(ph) = fix(ph \cdot pev)$, and then $fix(ph_c) = fix(ph_{c,ins})$.

$\mathbf{fix}(\boldsymbol{ph}) = \mathbf{fix}(\boldsymbol{ph \cdot pev})$:
Appending a read event at the end of ph adds only fixed rf-elements to the reads-from relation and does not remove existing rf-elements. Thus, all finished transactions who previously were only element of fixed rf-elements still are. Because of that, all transactions in $fix(ph)$ are in $fix(ph \cdot pev)$. Also, all transactions which were element of an rf-element with tr_r in it, still are. The new transaction tr is not finished, and thus it cannot be element of $fix(ph \cdot pev)$. This implies that a transaction not in $fix(ph)$ is also not in $fix(ph \cdot pev)$, from which $fix(ph) = fix(ph \cdot pev)$ follows.

$\mathbf{fix}(\boldsymbol{ph_c}) = \mathbf{fix}(\boldsymbol{ph_{c,ins}})$:
Note that tr cannot be element of $fix(ph_c)$ or $fix(ph_{c,ins})$. It cannot be in the former since it does not exist in ph_c and in the latter because it is unfinished in $ph_{c,ins}$. Also, note that a transaction that is not tr is either finished in both ph_c and $ph_{c,ins}$ or in none of them. This is because the p-events of such a transaction are identical in in ph_c and $ph_{c,ins}$. Thus, we only need to show that

$$\{tr \in Tr \mid fin_{ph_c}(tr), \forall rf \in ph_c.RF : tr \in rf \rightarrow fix_{ph_c}(rf) \vee int_{ph_c}(rf)\}$$
$$=$$
$$\{tr \in Tr \mid fin_{ph_c}(tr), \forall rf \in ph_{c,ins}.RF : tr \in rf \rightarrow fix_{ph_{c,ins}}(rf) \vee int_{ph_{c,ins}}(rf)\}.$$

Take an arbitrary transaction tr_y occurring in ph_c. Note that any transaction in both sets that is read by tr_r contains at least one rf-element that is not fixed and not interruptible, which means it cannot be in either set. Thus, we assume that

tr_y is not read by tr_r in ph_c and $ph_{c,ins}$. It holds that

$$\forall x \in \mathit{Var}\ ((tr_y, tr_r, x) \notin ph_c.RF \leftrightarrow (tr_y, tr_r, x) \notin ph_{c,ins}.RF)$$
and
$$\forall x \in \mathit{Var}\ (tr_y, tr_r, x) \notin ph_c \rightarrow \{rf \in ph_c.RF \mid tr_y \in rf\} = \{rf \in ph_{c,ins}.RF \mid tr_y \in rf\}.$$

The former holds true as all p-events in ph_c are in $ph_{c,ins}$, and their order remains identical. The latter holds true as all existing rf-elements are unchanged by the same reason as above. Any new rf-element must involve tr, which is inserted after the last write. Thus, it cannot read tr_y any more if it is not read by tr_r.
Let $rf = (tr', tr'', x)$ be an arbitrary rf-element in $\{rf \in ph_c.RF \mid tr_y \in rf\}$. Note that it is also in $\{rf \in ph_{c,ins}.RF \mid tr_y \in rf\}$ by the above reasoning. We show two things:

1. $fix_{ph_c}(rf) \leftrightarrow fix_{ph_{c,ins}}(rf)$
2. and $int_{ph_c}(rf) \leftrightarrow int_{ph_{c,ins}}(rf)$.

$\mathbf{fix}_{\boldsymbol{ph_c}}(rf) \leftrightarrow \mathbf{fix}_{\boldsymbol{ph_{c,ins}}}(rf)$
If $fix_{ph_c}(rf)$ holds, then by Lemma 12 it holds that

$$\neg(int_{ph_c}(rf)) \wedge tr'' \neq tr_r.$$

The same holds analogue for $ph_{c,ins}$. The second part of the equivalency is true for both $fix_{ph_c}(rf)$ and $fix_{ph'_c}(rf)$. We thus need to show that $\neg(int_{ph_c}(rf)) \leftrightarrow \neg(int_{ph_{c,ins}}(rf))$. By using Lemma 13, we expand the equivalency to

$$(\exists tr_a \in Tr : fin_{ph_c}(tr_a) \wedge pev_{ph_c}^{tr'',rd} <_{ph_c} pev_{ph_c}^{tr_a,wr})$$
$$\wedge$$
$$\neg(\exists tr_i \in Tr : unfin_{ph_c}(tr_i) \wedge pev_{ph_c}^{tr',wr} <_{ph_c} pev_{ph_c}^{tr_i,rd} <_{ph_c} pev_{ph_c}^{tr'',rd})$$
$$\leftrightarrow$$
$$(\exists tr_a \in Tr : fin_{ph_{c,ins}}(tr_a) \wedge pev_{ph_{c,ins}}^{tr'',rd} <_{ph_{c,ins}} pev_{ph_{c,ins}}^{tr_w,wr})$$
$$\wedge$$
$$\neg(\exists tr_i \in Tr : unfin_{ph_{c,ins}}(tr_i) \wedge pev_{ph_{c,ins}}^{tr',wr} <_{ph_{c,ins}} pev_{ph_{c,ins}}^{tr_i,rd} <_{ph_{c,ins}} pev_{ph_{c,ins}}^{tr'',rd}).$$

We transform the right-hand part of the equivalency. If such a tr_i exists, then it cannot be tr as tr is not finished. Also note that if the first part of the conjunction is true, then a write exists after the read of tr''. By definition of the insertion function, pev is located after that write in $ph_{c,ins}$, and thus cannot be in between $pev_{ph_{c,ins}}^{tr',wr}$ and $pev_{ph_{c,ins}}^{tr'',rd}$. Thus, the right-hand part is equivalent to

$$\begin{gathered}(\exists tr_a \in Tr\backslash\{tr\} : fin_{ph_{c,ins}}(tr_a) \wedge pev_{ph_{c,ins}}^{tr'',rd} <_{ph_{c,ins}} pev_{ph_{c,ins}}^{tr_w,wr})\\ \wedge \\ \neg(\exists tr_i \in Tr\backslash\{tr\} : unfin_{ph_{c,ins}}(tr_i) \wedge pev_{ph_{c,ins}}^{tr',wr} <_{ph_{c,ins}} pev_{ph_{c,ins}}^{tr_i,rd} <_{ph_{c,ins}} pev_{ph_{c,ins}}^{tr'',rd}).\end{gathered}$$

This means that all p-events referenced are occurring in ph_c. It is trivially true by Definition 20 that $pev' \in ph_c \rightarrow pev' \in ph_{c,ins}$ and

$$\forall pev', pev'' \in ph_c : pev' <_{ph_{c,ins}} pev'' \leftrightarrow pev' <_{ph_c} pev''.$$

Inserting this into the left-hand side of the equivalency makes both sides of the implication trivially equivalent.

$\mathbf{int}_{\boldsymbol{ph_c}}(rf) \leftrightarrow \mathbf{int}_{\boldsymbol{ph_{c,ins}}}(rf)$:
We need to show that

$$\begin{gathered}\forall tr_a \in Tr : unfin_{ph_c}(tr_a) \vee \neg(pev_{ph_c}^{tr'',rd} <_{ph_c} pev_{ph_c}^{tr_w,wr})\\ \vee \\ \exists tr_i \in Tr : unfin_{ph_c}(tr_i) \wedge pev_{ph_c}^{tr',wr} <_{ph_c} pev_{ph_c}^{tr_i,rd} <_{ph_c} pev_{ph_c}^{tr'',rd}\\ \leftrightarrow \\ \forall tr_a \in Tr : unfin_{ph_{c,ins}}(tr_a) \vee \neg(pev_{ph_{c,ins}}^{tr'',rd} <_{ph_{c,ins}} pev_{ph_{c,ins}}^{tr_w,wr})\\ \vee \\ \exists tr_i \in Tr : unfin_{ph_{c,ins}}(tr_i) \wedge pev_{ph_{c,ins}}^{tr',wr} <_{ph_c} pev_{ph_{c,ins}}^{tr_i,rd} <_{ph_{c,ins}} pev_{ph_{c,ins}}^{tr'',rd}.\end{gathered}$$

The transaction tr is unfinished, and thus for the case $tr_a = tr$ the statement after the all quantifier is true on both sides of the equivalency. Thus, replacing it with $\forall tr_a \in Tr\backslash\{tr\}$ is equivalent to the original quantifier for this statement.

Next, we take a look at

$$\exists tr_i \in Tr : unfin_{ph_{c,ins}}(tr_i) \wedge pev^{tr',wr}_{ph_{c,ins}} <_{ph_c} pev^{tr_i,rd}_{ph_{c,ins}} <_{ph_{c,ins}} pev^{tr'',rd}_{ph_{c,ins}}.$$

Assume $tr_i = tr$, then there cannot be another write p-event after $pev^{tr_i,rd}_{ph_{c,ins}}$ (which equals pev) by definition of the insertion function. As $pev^{tr_i,rd}_{ph_{c,ins}} <_{ph_{c,ins}} pev^{tr'',rd}_{ph_{c,ins}}$ holds, this implies that

$$\forall tr_a \in Tr : unfin_{ph_{c,ins}}(tr_a) \vee \neg(pev^{tr'',rd}_{ph_{c,ins}} <_{ph_{c,ins}} pev^{tr_w,wr}_{ph_{c,ins}})$$

is true as there is no write event after $pev^{tr_w,wr}_{ph_{c,ins}}$. Thus, the overall equivalency is equivalent to

$$\begin{gathered}
\forall tr_a \in Tr \backslash \{tr\} : unfin_{ph_c}(tr_a) \vee \neg(pev^{tr'',rd}_{ph_c} <_{ph_c} pev^{tr_w,wr}_{ph_c}) \\
\vee \\
\exists tr_i \in Tr \backslash \{tr\} : unfin_{ph_c}(tr_i) \wedge pev^{tr',wr}_{ph_c} <_{ph_c} pev^{tr_i,rd}_{ph_c} <_{ph_c} pev^{tr'',rd}_{ph_c} \\
\leftrightarrow \\
\forall tr_a \in Tr \backslash \{tr\} : unfin_{ph_{c,ins}}(tr_a) \vee \neg(pev^{tr'',rd}_{ph_{c,ins}} <_{ph_{c,ins}} pev^{tr_w,wr}_{ph_{c,ins}}) \\
\vee \\
\exists tr_i \in Tr \backslash \{tr\} : unfin_{ph_{c,ins}}(tr_i) \wedge pev^{tr',wr}_{ph_{c,ins}} <_{ph_c} pev^{tr_i,rd}_{ph_{c,ins}} <_{ph_{c,ins}} pev^{tr'',rd}_{ph_{c,ins}}.
\end{gathered}$$

This means that all p-events referenced are occurring in ph_c. It is trivially true by Definition 20 that all $pev' \in ph_c \to pev' \in ph_{c,ins}$ and

$$\forall pev', pev'' \in ph_c : pev' <_{ph_{c,ins}} pev'' \leftrightarrow pev' <_{ph_c} pev''.$$

Inserting this into the left-hand side of the equivalency makes both sides of the implication trivially equivalent. □

Proposition 9. *Given an hc-pair* $hc = (ph, ph_c)$ *and its extension by an arbitrary write event* $pev = \mathbf{W}^{tr}_t[V]$ *to* $hc_+ = (ph \cdot pev, ph_{c,ins})$ *s.t.* $ph_{c,ins} \in ins(ph_c, pev)$*, it holds that*

$$f(ph \cdot pev) = f(ph) \cup cov_{ph}.$$

Proof. Note that

$$f(ph \cdot pev) \backslash f(ph) = \{tr' \in Tr \mid \quad \mathit{fin}_{ph}(tr'), \; \exists rf \in ph.RF : tr' \in rf \wedge tr_r \in rf, \; \forall rf \in (ph \cdot pev).RF : tr' \in rf \rightarrow tr_r \notin rf\}.$$

We show that $f(ph \cdot pev) \backslash f(ph) = cov_{ph}$ by showing that both are a subset or equal of each other.

$\mathbf{f}(\boldsymbol{ph \cdot pev})\backslash\mathbf{f}(\boldsymbol{ph}) \subseteq \boldsymbol{cov_{ph}}$ **:** Let tr' be an arbitrary transaction in $f(ph \cdot pev) \backslash f(ph)$. Let $x \in Var$ be a variable s.t. $(tr', tr_r, x) \in ph.RF$. It holds by construction that $(tr', tr_r, x) \notin (ph \cdot pev).RF$. This implies that $pev_{ph}^{tr',wr} <_{ph} pev_{ph}^{tr,wr}$ and $\exists x \in Var : (tr', tr_r, x) \in ph.RF$. Also, if any arbitrary $(tr', tr_r, x) \in ph.RF$ is not in $(ph \cdot pev).RF$, then $\forall x \in Var : (tr', tr_r, x) \in ph.RF \rightarrow x \in V$ must hold. As these three conditions are true, $tr' \in cov_{ph}$, and thus the subset equal relation holds true.

$\boldsymbol{cov_{ph}} \subseteq \mathbf{f}(\boldsymbol{ph \cdot pev})\backslash\mathbf{f}(\boldsymbol{ph})$ **:**
Let tr' be an arbitrary transaction in cov_{ph}. It has a write in $ph \cdot pev$ by definition of cov_{ph} and ph contains this write so $\mathit{fin}_{ph}(tr')$ holds. Also, by definition of cov_{ph}, $\exists rf \in ph.RF : tr' \in rf \wedge tr_r \in rf$ holds. Finally, as pev is ordered after $pev_{ph \cdot pev}^{tr',wr}$ and writes to each variable tr_r reads from tr', it holds that $\forall rf \in (ph \cdot pev).RF : tr' \in rf \rightarrow tr_r \notin rf$. This proves the claim. □

Proposition 10. *Given an hc-pair* $hc = (ph, ph_c)$ *and its extension by an arbitrary write event* $pev = \mathbf{W}_t^{tr}[V]$ *to* $hc_+ = (ph \cdot pev, ph_{c,ins})$ *s.t.* $ph_{c,ins} \in ins(ph_c, pev)$, *it holds that*

$$f(ph_{c,ins}) = f(ph_c) \cup cov_{ph_c}.$$

Proof. Note that

$$f(ph_{c,ins})\backslash f(ph_c)\backslash\{tr\} = \{tr' \in Tr, tr' \neq tr \mid \begin{array}{l} \mathit{fin}_{ph_c}(tr'), \\ \exists rf \in ph_c.RF : tr' \in rf \land tr_r \in rf, \\ \forall rf \in (ph_{c,ins}).RF : tr' \in rf \rightarrow tr_r \notin rf\} \end{array}$$

and

$$cov_{ph_c} = \{tr' \mid \begin{array}{l} pev^{tr',wr}_{ph_{c,ins}} <_{ph_{c,ins}} pev^{tr,wr}_{ph_{c,ins}}, \exists x \in \mathit{Var} : (tr', tr_r, x) \in ph_c.RF, \\ \forall x \in \mathit{Var} : (tr', tr_r, x) \in ph_c.RF \rightarrow x \in V\}. \end{array}$$

$\mathbf{f}(\boldsymbol{ph}_{\mathbf{c,ins}})\backslash\mathbf{f}(\boldsymbol{ph}_{\mathbf{c}})\backslash\{\boldsymbol{tr}\} \subseteq \boldsymbol{cov}_{\boldsymbol{ph}_{\mathbf{c}}}$ **:**
Consider an arbitrary transaction $tr' \in f(ph_{c,ins})\backslash f(ph_c)\backslash\{tr\}$. We show it fulfils all builder predicates of cov_{ph_c}. Let X be the set of variables that tr_r reads from tr' in ph_c. As in $ph_{c,ins}$, tr_r reads no variable from tr' and the only change with regard to ph_c is the insertion of pev. That means that all variables of X are written to by pev and now read by tr_r. This implies $pev^{tr',wr}_{ph_{c,ins}} <_{ph_{c,ins}} pev^{tr,wr}_{ph_{c,ins}}$ and $\forall x \in \mathit{Var} : (tr', tr_r, x) \in ph_c.RF \rightarrow x \in V$. The statement $\exists rf \in ph_c.RF : tr' \in rf \land tr_r \in rf$ is trivially equivalent to $\exists x \in \mathit{Var} : (tr', tr_r, x) \in ph_c.RF$.

$\boldsymbol{cov}_{\boldsymbol{ph}_{\mathbf{c}}} \subseteq \mathbf{f}(\boldsymbol{ph}_{\mathbf{c,ins}})\backslash\mathbf{f}(\boldsymbol{ph}_{\mathbf{c}})\backslash\{\boldsymbol{tr}\}$ **:**
Consider an arbitrary transaction $tr' \in cov_{ph_c}$. We show it fulfils all builder predicates of $f(ph_{c,ins})\backslash f(ph_c)\backslash\{tr\}$. As by definition of cov_{ph_c}, there is a write event of tr' existing in ph_c. Thus, $tr' \neq tr$ and $\mathit{fin}_{ph_c}(tr')$ holds. Again by definition of cov_{ph_c}, $\exists x \in \mathit{Var} : (tr', tr_r, x) \in ph_c.RF$ holds. This is equivalent to $\exists rf \in ph_c.RF : tr' \in rf \land tr_r \in rf$. Lastly, by definition of cov_{ph_c} pev is a write ordered after the write of tr' and writes to all variables that tr_r reads from tr' in ph_c. Thus, in $ph_{c,ins}$ tr_r reads no variable from tr' which means $\forall rf \in (ph_{c,ins}).RF : tr' \in rf \rightarrow tr_r \notin rf$. The claim holds. □

Proposition 11. *Given an arbitrary p-history ph and a set of transactions X* *with two subsets X_1, X_2 s.t. $X_1 \cup X_2 = X$, it holds that*

$$ph\backslash X = (ph\backslash X_1)\backslash X_2.$$

Proof. Let $ph = pev_0 \dots pev_n$, let $ph\backslash X = pev_0^* \dots pev_n^*$ and let $(ph\backslash X_1) = pev_0' \dots pev_n'$ let $(ph\backslash X_1)\backslash X_2 = pev_0'' \dots pev_n''$. We show that for $0 \leq x \leq n$ $pev_x'' = pev_x^*$ holds. This is done via case distinction.

Case distinction $pev_x^* = \epsilon$:
By Definition 27, it holds that $tr_{ph}(pev_x) \in X$, which means $tr_{ph}(pev_x) \in X_1 \vee tr_{ph}(pev_x) \in X_2$. Thus, either $pev_x' = \epsilon$ or $pev_x'' = \epsilon$. In the latter case, $pev_x^* = pev_x''$ follows directly. In the first case, an ϵ "event" is still non-existent after applying the removal function again with any set, and thus $pev_x^* = pev_x''$ also holds.

Case distinction $pev_x^* = \mathbf{R}_t^{tr}[V^*]$:
Then, by Definition 27 it holds that $ph_x = \mathbf{R}_t^{tr}[V]$ and $V^* = V\backslash\{y \mid \exists tr' \in X(tr', tr, y) \in ph.RF\}$, this is equivalent to

$$V^* = (V\backslash\{y \mid \exists tr' \in X_1(tr', tr, y) \in ph.RF\})\backslash\{y \mid \exists tr' \in X_2(tr', tr, y) \in ph.RF\}.$$

For $pev_x' = \mathbf{R}_t^{tr}[V']$ by Definition 27 it holds that $pev_x = \mathbf{R}_t^{tr}[V]$ and $V' = V\backslash\{y \mid \exists tr' \in X_1(tr', tr, y) \in ph.RF\}$, then $pev_x'' = \mathbf{R}_t^{tr}[V'']$, and by the same definition is $V'' = V'\backslash\{y \mid \exists tr' \in X_2(tr', tr, y) \in ph.RF\}$. Thus, overall it holds that

$$\begin{aligned}
pev_x'' &= \mathbf{R}_t^{tr}[V'\backslash\{y \mid \exists tr' \in X_2(tr', tr, y) \in ph.RF\}] \\
&= \mathbf{R}_t^{tr}[(V\backslash\{y \mid \exists tr' \in X_1(tr', tr, y) \in ph.RF\})\backslash\{y \mid \exists tr' \in X_2(tr', tr, y) \in ph.RF\}] \\
&= \mathbf{R}_t^{tr}[(V\backslash\{y \mid \exists tr' \in X(tr', tr, y) \in ph.RF\})] \\
&= pev_x^*.
\end{aligned}$$

Case distinction $pev_x^* = pev_x$:

In this case, pev_x was neither a read nor was its transaction in X, if this is the case, then $pev''_x = pev'_x = pev_x$ as the transaction of pev_x is not in X_1 or X_2. Thus,

$$pev'' = pev_x = pev^*_x.$$

The claim holds.

$\square$

Proposition 12. *Given an hc-pair $hc = (ph, ph_c)$ and its extension by an arbitrary write event $pev = \mathbf{W}^{tr}_t[V]$ to $hc_+ = (ph \cdot pev, ph_{c,ins})$ s.t. $ph_{c,ins} \in ins(ph_c, pev)$ and not $doomed(hc_+)$, it holds that*

$$ph_c \backslash fix(hc) = pev'_0 \dots pev'_n \leftrightarrow ph_{c,ins} \backslash fix(hc) = pev'_0 \dots pev \dots pev'_n.$$

Proof. Let $ph_c = pev_0 \dots pev_n$, $ph_{c,ins} = pev_0 \dots pev \dots pev_n$ and $ph_c \backslash fix(hc) = pev'_0 \dots pev'_n$ hold. Note that pev is a write, and $tr \notin fix(hc)$ as tr is not finished in ph_c; thus, it is not removed by the removal function. Thus, $ph_{c,ins} \backslash fix(hc) = pev''_0 \dots pev \dots pev''_n$ holds. We show that $pev'_x = pev''_x$ for $0 \leq x \leq n$. This is done via case distinction.

Case distinction $tr_{ph_c}(pev_x) \in$ fix(hc):
This is equivalent to $tr_{ph_{c,ins}}(pev_x) \in fix(hc)$ by definition of the insertion function (Definition 20). Thus, $pev''_x = \epsilon = pev'_x$ holds.

Case distinction $pev_x = \mathbf{W}^{tr'}_{t'}[V], tr' \notin$ fix(hc):
The p-event pev_x does not belong to a transaction in $fix(hc)$, and it is not a read. Thus, by definition of the removal function, it holds that

$$pev''_x = pev_x = pev'_x.$$

Case distinction $\boldsymbol{pev_{\mathrm{x}}} = \mathbf{R}_{t'}^{tr'}[\boldsymbol{V}], \boldsymbol{tr'} \notin \mathbf{fix(hc)}$:
Then, by Definition 27 it holds that $pev'_x = \mathbf{R}_{t'}^{tr'}[V']$ and $V' = V \backslash \{x \mid \exists tr'' \in fix(hc) : (tr'', tr', x) \in ph_c.RF\}$. Also, by Definition 27, it holds that $pev''_x = \mathbf{R}_{t'}^{tr'}[V'']$ and $V'' = V \backslash \{x \mid \exists tr'' \in fix(hc) : (tr'', tr', x) \in ph_{c,ins}.RF\}$. Note that $doomed(hc_+)$ is not true, and thus tr is not read by any other transaction in $ph_{c,ins}$ than tr_r. Thus, given $tr' \neq tr_r$ it holds for any rf-element (tr'', tr', x) that $(tr'', tr', x) \in ph_c.RF$ iff $(tr'', tr', x) \in ph_{c,ins}.RF$. Also, $tr_r \notin fix(hc)$ by definition, and thus

$$\{x \mid \exists tr'' \in fix(hc) : (tr'', tr', x) \in ph_c.RF\} = \{x \mid \exists tr'' \in fix(hc) : (tr'', tr', x) \in ph_{c,ins}.RF\}$$

holds from which $pev'_x = pev''_x$ follows.

The claim holds. □

Lemma 17. *Two p-histories ph, ph' are SSR^--extension equivalent iff the following two conditions hold:*

1. $\forall (ph, ph_c) \in HC_{ph}, \exists (ph', ph'_c) \in HC_{ph'} : (ph, ph_c) \equiv_{ext} (ph', ph'_c)$,
2. *and* $\forall (ph', ph'_c) \in HC_{ph'}, \exists (ph, ph_c) \in HC_{ph} : (ph', ph'_c) \equiv_{ext} (ph, ph_c)$.

Proof. We first show that the first condition implies that if ph has a extension that is SSR^- serializable, then ph' does as well. Assume for an arbitrary p-event sequence seq and arbitrary p-histories ph and ph' that

$$\forall (ph, ph_c) \in HC_{ph}, \exists (ph', ph'_c) \in HC_{ph'} : (ph, ph_c) \equiv_{ext} (ph', ph'_c)$$

holds. It is trivial to see by the definition of candidates that a p-history is serializable under SSR^- iff one of its candidates is equivalent to it. Thus, if $ph \cdot seq$ is serializable under SSR^{-}, there exists an consistent hc-pair $(ph \cdot seq, ph_{c,ins})$. By using Proposition 2 and induction, it is given that

$$\bigcup_{ph_c \in C_{ph}} ins(ph_c, seq) = C_{ph \cdot seq}.$$

Thus, there must exist an hc-pair (ph, ph_c) s.t. $ph_{c,ins} \in ins(ph_c, seq)$. Then, by assumption there exists (ph', ph'_c) s.t. $(ph, ph_c) \equiv_{ext} (ph', ph'_c)$. As $ph_{c,ins} \in ins(ph_c, pev)$ and $ph_{c,ins}$ is equivalent to $ph \cdot seq$ by definition of extension equivalence for hc-pairs, there must exist $ph'_{c,ins} \in ins(ph'_c, pev)$ and $ph'_{c,ins}$ is equivalent to $ph' \cdot seq$. This means $ph' \cdot seq$ is serializable under SSR^{-}.

The other direction is proven analogue as we did not use specific properties of ph or ph'.

□

Lemma 18. *Two p-histories ph, ph' are SSR^{-}-extension equivalent if $ssr(ph) = ssr(ph')$.*

Proof. If $ssr(ph) = ssr(ph')$, then it holds that for each hc-pair (ph, ph_c) s.t. $ph_c \in C_{ph}$ there exists a hc-pair (ph', ph'_c) s.t. $cmp(ph, ph_c) = cmp(ph', ph'_c)$ which by Lemma 16 implies that $(ph, ph_c) \equiv_{ext} (ph', ph'_c)$. Thus,

$$\forall (ph, ph_c) \in HC_{ph}, \exists (ph', ph'_c) \in HC_{ph'} : (ph, ph_c) \equiv_{ext} (ph', ph'_c)$$

holds true, which is the first condition of Lemma 17).

By an analogue proof the second condition of Lemma 17,

$$\forall (ph', ph'_c) \in HC_{ph'}, \exists (ph, ph_c) \in HC_{ph} : (ph', ph'_c) \equiv_{ext} (ph, ph_c)$$

holds true as well.

As shown, both conditions of Lemma 17 hold true, and thus ph and ph' are SSR^{-}-extension equivalent.

□

Theorem 3. *Let I be an implementation automaton. Then I only produces strictly serializable p-histories iff $L(E(I)) = \emptyset$.*

Proof. Let $I = (Q, \delta, q_0, F)$ be an implementation automaton and its SSR^--automaton $E(I)$ be $(Q_E, \delta_E, q_{0,E}, F_E)$. We show that $E(I)$ is a deterministic finite automaton in Proposition 13.

I is strictly serializable $\rightarrow$ L(E(I)) $= \emptyset$:
We show the contraposition. Let the accepted word be $ph = pev_0 \dots pev_n$. Then, there exists a run of $E(I)$

$$(q_0, ssr(\epsilon)) \dots (q_{n+1}, ssr(ph)),\ \text{s.t.}(q_{n+1}, ssr(ph)) \in F_E.$$

It must hold that $ssr(ph) \in SSR^-_{\emptyset\ T,Var}$, and thus there exists no consistent compressed hc-pair in $ssr(ph)$. As shown in Proposition 5, if a compressed hc-pair is not consistent, then all of its uncompressed versions are also not consistent. This means that there exists no consistent hc-pair for ph, meaning it is not strictly serializable.
It must also hold that $q_{n+1} \in F$ and for all indices $i < n+1$ that

$$((q_i, ssr(pev_0 \dots pev_{i-1})), pev_i, (q_{i+1}, ssr(pev_0 \dots pev_i))) \in \delta_E.$$

Thus, for the word ph in I the run $q_0 \dots q_{n+1}$ exist s.t. for all indices $i < n+1$ it holds that $(q_i, pev_i, q_{i+1}) \in \delta$. As $q_{n+1} \in F$ this is an accepting run meaning I is not strictly serializable.

L(E(I)) $= \emptyset \rightarrow$ I is strictly serializable :
We show the contraposition. Let $ph = pev_0 \dots pev_n$ be a p-history that is not serializable and accepted by I. Then in I a run $q_0 \dots q_{n+1}$ exist s.t. for all indices $i < n+1$ it holds that $(q_i, pev_i, q_{i+1}) \in \delta$ and $q_{n+1} \in F$ holds. Consider

the following sequence of states of Q_E:

$$(q_0, ssr(\epsilon)) \dots (q_{n+1}, ssr(ph))$$

s.t. for all indices $i < n + 1$

$$((q_i, ssr(pev_0 \dots pev_{i-1})), pev_i, (q_{i+1}, ssr(pev_0 \dots pev_i))) \in \delta_E$$

holds. This sequence exists by the definition of an SSR^--automaton. As ph is not serializable, for all of its candidates ph_c, the hc-pair (ph, ph_c) is not consistent, and as shown in Equation (B.1) its compression $cmp(ph, ph_c)$ is then also not consistent. Thus, there exists no consistent compressed hc-pair in $ssr(ph)$, and thus $ssr(ph) \in SSR^-_{\emptyset\ T,Var}$. As $q_{n+1} \in F$, this means $(q_{n+1}, ssr(ph)) \in F_E$, and thus $E(I)$ accepts ph. It follows that its language is not empty. □

Proposition 13 (SSR^- construction is DFA)**.** *Let* $I = (Q, \delta, q_0, F)$ *be an implementation automaton and its* SSR^--automaton of $E(I)$ *be* $(Q_E, \delta_E, q_{0,E}, F_E)$. *Then,* $E(I)$ *is a deterministic finite automaton.*

Proof. We assume I to be an DFA. We show that $E(I)$ is a finite automaton and then that it is a deterministic automaton.

$\mathbf{Q_E}$ is finite:
Note that Q_E equals $Q \times SSR^-_{T,Var}$. As I is a DFA Q, is finite. As proven in Lemma 15, there is only a finite amount of compressed hc-pairs for a given T and Var. Thus, $SSR^-_{T,Var}$ is finite as well as each element in it is a set of hc-pairs. Combining both facts means Q_E is a finite set.

$\boldsymbol{\delta_E}$ is deterministic:
Consider a given state/ssr data pair $(q, ssr) \in Q_E$, $q \in Q$ and $ssr \in SSR^-_{T,Var}$ and an arbitrary p-event pev. We show there is only exactly one pair of $q' \in Q$

and SSR^{-}-data ssr' that fulfils the condition for a transition, which is

$$(q, pev, q') \in \delta \text{ and } \exists ph \in \mathcal{PH} : ssr(ph) = ssr \wedge ssr(ph \cdot pev) = ssr'.$$

We do this by a proof via contradiction. Assume a different second pair (q'', ssr'') fulfils the condition for the transition. So, at least $q'' \neq q'$ or $ssr'' \neq ssr'$ must hold. Trivially, as Q is a DFA q'' equals q'. So, $ssr'' \neq ssr'$ must hold. Thus, there must exist a p-history $ph' \neq ph$ s.t. $ssr(ph') = ssr \wedge ssr(ph' \cdot pev) = ssr''$. This implies $ssr(ph) = ssr(ph')$ is true. As we show in the proof of Lemma 19 in this appendix,

$$\forall ph, ph' \in \mathcal{PH} : ssr(ph) = ssr \wedge ssr(ph') = ssr \rightarrow ssr(ph \cdot pev) = ssr(ph' \cdot pev)$$

holds. This implies that $ssr(ph' \cdot pev) = ssr(ph \cdot pev) = ssr$, which is a contradiction to the assumption. Thus, there exists only one pair (q', ssr') that fulfils the condition for a transition. $\square$

Lemma 19. *Given two arbitrary p-histories ph and ph', it holds that*

$$\forall seq \in PEv^{*} : ssr(ph) = ssr(ph') \rightarrow ssr(ph \cdot seq) = ssr(ph' \cdot seq).$$

Proof. Let pev be an arbitrary p-event. By definition, it holds that

$$ssr(ph \cdot pev) = \{cmp(ph \cdot pev, ph_{c+}) \mid ph_{c+} \in C_{ph \cdot pev}\}.$$

Using Proposition 2, it follows that

$$ssr(ph \cdot pev) = \{cmp(ph \cdot pev, ph_{c+}) \mid ph_{c+} \in \bigcup_{ph_c \in C_{ph}} ins(ph_c, pev)\}.$$

The same holds for $ph' \cdot pev$.

$$ssr(ph' \cdot pev) = \{cmp(ph' \cdot pev, ph'_{c+}) \mid ph'_{c+} \in \bigcup_{ph'_c \in C_{ph'}} ins(ph'_c, pev)\}.$$

$\mathbf{ssr(\boldsymbol{ph \cdot pev}) \subseteq ssr(\boldsymbol{ph' \cdot pev}) :}$
Let (ph, ph_{c+}) be an arbitrary element in $ssr(ph \cdot pev)$. Consider a candidate $ph_c \in C_{ph}$ s.t. $ph_{c+} \in ins(ph_c, pev)$, which exists by Proposition 2. Given that $ssr(ph) = ssr(ph')$, there exists $ph'_c \in C_{ph'}$ s.t. $cmp(ph, ph_c) = cmp(ph', ph'_c)$. As we have shown in the proof of Lemma 16 in this appendix, it then holds that

$$\exists ph'_{c,ins} \in ins(ph'_c, pev) : cmp(ph \cdot pev, ph_{c,ins}) = cmp(ph' \cdot pev, ph'_{c,ins}).$$

Let $ph'_{c,ins}$ be one of these candidate extensions. As

$$ph'_{c,ins} \in \bigcup_{ph'_c \in C_{ph'}} ins(ph'_c, pev)$$

holds, it holds that $cmp(ph' \cdot pev, ph'_{c,ins}) \in ssr(ph' \cdot pev)$.

$\mathbf{ssr(\boldsymbol{ph' \cdot pev}) \subseteq ssr(\boldsymbol{ph \cdot pev}) :}$
This proof is analogue to the previous case.

Combining both cases shows that $ssr(ph \cdot pev) = ssr(ph' \cdot pev)$ holds. By induction, it follows that

$$\forall seq \in PEv^* : ssr(ph) = ssr(ph') \rightarrow ssr(ph \cdot seq) = ssr(ph' \cdot seq).$$

□

C
Proofs for Section 4.3

Proposition 3 (Supersequence property)**.** *Given a g-history h and an arbitrary extension h' of it, it holds that*

$$\forall h'_c \in C_{h'}, \exists h_c \in C_h : h_c \sqsubseteq h'_c.$$

Proof. We prove the claim for the extension of an arbitrary g-history h by a single g-event ev to $h' = h \cdot ev$. This proves the claim for arbitrary extensions by induction. Let h'_c be an arbitrary candidate of h'. Let h_c be h'_c with the g-event ev removed. We prove it is a candidate of h by showing all candidate properties.

$\boldsymbol{h_c}$ is serial:
Removing an event from the serial history h'_c preserves the property.

$\boldsymbol{h_c}$ is equivalent to $\boldsymbol{h}$:
The history h' contains the same events in the same order as h except for the addition of ev. Then, for h'_c the same holds by property of it being a candidate of h'. It holds that h_c is identical to h'_c except for the removal of ev. Thus, h_c

has the same events and internal thread order as h.

h_c preserves the real-time order of h:
For this we first define the set of rt-elements generated in h' and h'_c by adding ev to h. The set of events rt-elements generated by ev in h' is

$$RT_{h'(ev)} = \{(tr, tr') \in h'.RT \mid ev \in \{\mathbf{B}^{tr'}_{thr(tr')}, \mathbf{A}^{tr}_{thr(tr)}, \mathbf{Resp}^{tr}_{thr(tr)}(\mathbf{C})\}\}.$$

The definition for h'_c is analogue. Note that any rt-element (tr, tr') that is in $RT_{h'_c}(ev)$ but not in $RT_{h'}(ev)$ is not present in $h'.RT$. This is the case because if it was present in $h'.RT$, it would also be in $RT_{h'}(ev)$ as tr and tr' have the same events in h' and h'_c. This is expressed by the following formula:

$$\forall (tr, tr') \in RT_{h'_c}(ev) : (tr, tr') \notin RT_{h'}(ev) \rightarrow (tr, tr') \notin h'.RT. \tag{C.1}$$

Given the real-time order preservation property of candidates, it is given that

$$h'.RT \subseteq h'_c.RT.$$

If we subtract the rt-elements caused by ev on each side and use the previous facts, we get the following equation:

$$h'.RT \backslash RT_{h'}(ev) \subseteq h'_c.RT \backslash RT_{h'_c}(ev).$$

We argue why the subset or equal relation still holds. Equation C.1 implies that every rt-element that is subtracted from the right side and not from the left side, is not present in $h'.RT$. Thus, the only rt-elements that are subtracted from the real-time order of $h'_c.RT$ are either also subtracted from $h'.RT$ or are not an element of it. It follows that the subset or equal relation is preserved by the subtraction. The real-time order of h' without any rt-elements caused by ev is identical the real-time order of h. Also, the real-time order of h'_c minus all rt-elements caused by ev is the real-time order of h_c. Thus, it holds that

$$h.RT \subseteq h_c.RT.$$

So, h_c preserves the real-time order of h.

This proves the claim. $\square$

Proposition 4 (Generation of candidates by insertion function)**.** *Given a g-history h and a g-event ev, it holds that*

$$\bigcup_{h_c \in C_h} ins(h_c, ev) = C_{h \cdot ev}.$$

Proof. We show the subset equal relation in both directions.

$\bigcup_{h_c \in C_h} ins(h_c, ev) \subseteq \mathbf{C}_{h \cdot ev}$:
Let $h_c = ev_0 \ldots ev_n$ be an arbitrary candidate of h, so both contain the same g-events. Let h'_c be an arbitrary element of $ins(h_c, ev)$. We show that h'_c is a candidate of $h \cdot ev$ by case distinction over ev.

ev is not a begin g-event:

Let ev be of transaction tr. Then, there is at least one event of tr in h_c, so $h_c = ev_0 \ldots ev^{tr,ls}_{h_c} ev \ldots ev_n$ holds. It holds that

$$h'_c \in \{ev_0 \ldots ev^{tr,ls}_{h_c} ev \ldots ev_n\}.$$

h'_c is serial:
The g-history h'_c is trivially serial.

h'_c is equivalent to $h \cdot ev$:
The g-history h'_c trivially contains the same elements as $h \cdot ev$. We show that all events are ordered identical for each thread in h'_c and $h \cdot ev$. For all

g-events that are not ev this is the case as their order is identical in h and h_c, and it neither changes between h and $h \cdot ev$ or between h_c and h'_c. The g-event ev is ordered directly behind the last event of its transaction in h'_c, which is also the case in $h \cdot ev$. Thus, the claim holds.

h'_c preserves the real-time order of h:
We show h'_c preserves the real-time order of $h \cdot ev$. Appending a non-begin g-event does not change the real-time order of a g-history. Thus, it holds that

$$h.RT = (h \cdot ev).RT.$$

Also, by h_c being a candidate of h it holds that

$$h.RT \subseteq h_c.RT.$$

Adding a non-begin g-event to h_c does not subtract rt-elements from the real-time order. Thus, it holds that

$$(h \cdot ev).RT \subseteq h'_c.RT.$$

ev is a begin g-event:
Let ev be of transaction tr. It holds that

$$h'_c \in \{st(h_c) \cdot add(en(h_c), ev, n) \mid n \in TrI_{en(h_c)}\}.$$

h'_c is serial: The g-history h'_c is serial as ev is inserted in between two transactions. This is because $TrI_{en(h_c)}$ by definition only contains indices of the last g-event of each transaction.

h'_c is equivalent to $h \cdot ev$:
The g-history h'_c contains the same elements as $h \cdot ev$. We show that all events are ordered identical for each thread. For all g-events that are not ev this is the case as their order is identical in h and h_c, and it does not change between h and $h \cdot ev$, and between h_c and h'_c. The g-event ev is the first event of its transaction in h'_c, which is also the case in $h \cdot ev$.

h'_c preserves the real-time order of h:
The real-time order of $h \cdot ev$ is the one of h plus a number of rt-elements (tr', tr) between the transaction tr of the begin and each aborted and committed transaction tr' of h. The set of these rt-elements is denoted by $RT_{h \cdot ev}(ev)$. There are no other rt-elements added and none are subtracted. Thus, it holds that

$$(h \cdot ev).RT = h.RT \cup RT_{h \cdot ev}(ev).$$

For the candidate adding a begin g-event for tr somewhere after the last commit or abort adds an rt-element (tr', tr) for each committed or aborted transaction tr' in h_c to h'_c. As these transactions are identical to h, we can denote this set also by $RT_{h \cdot ev}(ev)$ There are no other rt-elements added and none are subtracted. Thus, it holds that

$$h'_c = h_c.RT \cup RT_{h \cdot ev}(ev).$$

As h_c preserves the real-time order of h,

$$h.RT \subseteq h_c.RT,$$

holds. It then holds that

$$h.RT \cup RT_{h \cdot ev}(ev) \subseteq h_c.RT \cup RT_{h \cdot ev}(ev),$$

and thus also

$$(h \cdot ev).RT \subseteq h'_c.RT.$$

The claim holds.

$\mathbf{C}_{\boldsymbol{h \cdot ev}} \subseteq \bigcup_{\boldsymbol{h_c \in C_h}} \boldsymbol{ins(h_c, ev)}$ **:**

Let h'_c be an arbitrary element in $C_{h \cdot ev}$. By the supersequence property (Proposition 3), a candidate h_c of h exists s.t.

$$h_c \sqsubseteq h'_c.$$

We show that $h'_c \in ins(h_c, ev)$ via case distinction.

***ev* is not a begin g-event:**

Let ev be of transaction tr. This transaction must at least have one g-event in h_c. Then, let $h_c = ev_0 \dots ev_{h_c}^{tr,ls} \dots ev_n$ hold. Note that $ins_{warc}(h_c, ev) = \{ev_0 \dots ev_{h_c}^{tr,ls} ev \dots ev_n\}$. As h'_c is serial and a super sequence of h_c, it must hold that $h'_c = ev_0 \dots ev_{h_c}^{tr,ls} ev \dots ev_n$. From this the claim follows:

$$h'_c \in ins(h_c, ev).$$

***ev* is a begin g-event:**

Let ev be of transaction tr. Let the last committed or aborted transaction in h_c be tr_x. Note that

$$ins_b(h_c, ev) = \{st(h_c) \cdot add(en(h_c), ev, n) \mid n \in TrI_{en(h_c)}\}.$$

It holds that $tr_x \prec_{h \cdot ev} tr$ and that h'_c preserves the real-time order of $h \cdot ev$ so $ev_{h'_c}^{tr_x,ls} <_{h'_c} ev$ holds. Also, h_c and h'_c have the same events and order of events except for ev. This implies that ev is not part of $st(h'_c)$, but it is part of $en(h'_c)$. The transaction tr_x contains the same g-events in both candidates, and it is the last committed or aborted transaction in both. It follows that $st(h_c) = st(h'_c)$. In addition, h'_c is serial. Thus, there exists some index n s.t.

$n \in TrI_{en(h_c)}$ and it holds that

$$en(h'_c) = add(en(h_c), ev, n).$$

This implies that

$$h'_c \in \{st(h_c) \cdot add(en(h_c), ev, n) \mid n \in TrI_{en(h_c)}\},$$

which proves the claim. $\square$

Lemma 20. *Given a g-history h, it holds that*

$$\exists h_s \in \mathcal{H} : h_s \in OW(h) \to \exists h_{c,comp} \in \mathcal{H} : h_{c,comp} \in SC(C_h) \wedge h_{c,comp} \textit{ is legal.}$$

Proof. Let h_c be a history s.t.

1. $\forall ev \in Ev : ev \in h_c \leftrightarrow ev \in h$,
2. and $\forall ev, ev' \in h_c : ev <_{h_c} ev' \leftrightarrow ev <_{h_s} ev'$,

We show that h_c is a candidate for h and one of its completions is legal. We start by showing it is a candidate.

h_c is serial: It is trivially serial.

h_c is equivalent to h :

It contains the same elements as h by definition. As h_s is an OP^--witness for h, it is equivalent to h, which implies

$$\forall ev, ev' \in h : tr_h(ev) = tr_h(ev') \to (ev <_h ev' \to ev <_{h_s} ev').$$

By construction, all events of h_c are ordered as in h_s, and all of these event are existing in h; thus, we can insert h_c for h_s:

$$\forall ev, ev' \in h : tr_h(ev) = tr_h(ev') \to (ev <_h ev' \to ev <_{h_c} ev').$$

Thus, all g-events of the same transaction are ordered the same in h and h_c.

h_c preserves the real-time order of h :
As h_s is an OP^--witness of h and any completion of h also preserves the real-time order of h, it holds that

$$\forall (tr, tr') \in h.RT : (tr, tr') \in h_s.RT.$$

Now h_c contains the same events as h; thus, all completed transactions in h are also completed in h_c and all of their events also exists in h_c by construction. In addition, if two of these completed transactions are real-time ordered in h_s, they are also in h_c by the second construction condition of h_c. Thus, in the previous equation, we can replace h_s by h_c

$$\forall (tr, tr') \in h.RT : (tr, tr') \in h_c.RT,$$

which proves the claim.

Next we prove that there exists a legal serial completion for h_c. It holds that h_c and h_s share all events occurring in h and these events are ordered identically in both. It is trivial to see that h_s then is one of the serial completions of h_c. It follows that h_c has a legal completion as h_s is legal. $\square$

Lemma 21. *Given a g-history h, it holds that*

$$\exists h_{c,comp} \in \mathcal{H} : h_{c,comp} \in SC(C_h) \wedge h_{c,comp} \textit{ is legal } \rightarrow \exists h_s \in \mathcal{H} : h_s \in OW(h).$$

Proof. Let h_c be a witness s.t. $h_{c,comp}$ is a legal serial completion of it. We show that $h_s = h_{c,comp}$ is an OP^--witness of h by first showing there exists a completion of h that is equivalent to it and then show it preserves the real-time order of h. It is already legal by definition.

There exists a completion of $\boldsymbol{h}$ equivalent to $\boldsymbol{h_{c,comp}}$:

Let Ev_c be all g-events that occur in $h_{c,comp}$ but not in h_c. Let h_{comp} be h with all events of Ev_c added directly after the last event of their respective transactions. This is trivially a well-formed definition of a completion of h. We show that h_{comp} is equivalent to $h_s/h_{c,comp}$. The g-events of both are trivially identical. Given two g-events $ev, ev' \in h_{comp}$ from the same transaction i.e. $tr_{h_{comp}}(ev) = tr_{h_{comp}}(ev')$, we show that

$$ev <_{h_{comp}} ev' \leftrightarrow ev <_{h_s} ev'.$$

We show this via case distinction.

$\boldsymbol{ev \notin Ev_c, ev' \in Ev_c}$ **:**
In this case, ev must be ordered before ev' in h_s and h_{comp} as both are a completion, so the added events must be added after the last event of their transaction of the g-histories they were generated from. The claim holds for this case.

$\boldsymbol{ev \notin Ev_c, ev' \notin Ev_c}$ **:**
Note that h_c (and thus also h_s) and h_{comp} have an identical internal thread order to h for all events occurring h. So, the order of ev and ev' in h_s and h_{comp} is identical to their order in h, and thus the same.

$\boldsymbol{ev \in Ev_c}$ **:**
This case cannot occur, any added g-event occurs after the last g-event of its transaction in a completion.

$\boldsymbol{h_{c,comp}}$ **preserves the real-time order of $\boldsymbol{h}$:**
It holds that h_c preserves the real-time order of h, by being a candidate of it:

$$h.RT \subseteq h_c.RT.$$

In the serial completion abort and commit responses are added, no g-events are reordered or removed. Thus, its real-time order is a superset of the candidate, and it follows that

$$h.RT \subseteq h_c.RT \subseteq h_{c,comp}.RT,$$

which proves the claim. □

Lemma 23 (Legal)**.** *Given a g-history h and a serial completion h_s of one of its candidates, it holds that*

$$h_s \textit{ is legal } \leftrightarrow h_s.RF_c = h.RF_{val}.$$

Proof. We show both directions separately for this proof.

→:

We first show that if h_s is legal holds, then $h_s.RF_c \subseteq h.RF_{val}$ holds, and then that $h_s.RF_{val} \subseteq h.RF_c$ is true.

$\boldsymbol{h_s.RF_c \subseteq h.RF_{\text{val}}}$ **:**

Consider an arbitrary rf-element $(tr', tr, var) \in h_s.RF_c$. We show that $(tr', tr, var) \in h.RF_{val}$ by showing that there exists a value $val \in Val$ s.t. all conditions of Definition 38 hold.

$\boldsymbol{(var, val) \in WS_h(tr'), (var, val) \in RS_h(tr)}$ **:**

As h_s is legal, it holds that for a value val $(var, val) \in WS_{h_s}(tr')$ and $(var, val) \in RS_{h_s}(tr)$. This implies the same for h proving the claim.

$\mathbf{Inv}^{\boldsymbol{tr'}}_{\boldsymbol{thr(tr')}}(\mathbf{C}) \boldsymbol{<_h} \mathbf{R}^{\boldsymbol{tr}}_{\boldsymbol{thr(tr)}}(\boldsymbol{var, val})$:

Assume the opposite which is $\mathbf{R}^{tr}_{thr(tr)}(var, val) <_h \mathbf{Inv}^{tr'}_{thr(tr')}(\mathbf{C})$. By the RR-assumption, it holds that it only reads val if there exists a committed transaction tr'' writing var to val s.t. $\mathbf{Inv}^{tr''}_{thr(tr'')}(\mathbf{C}) <_h \mathbf{R}^{tr}_{thr(tr)}(var, val)$. So,

tr'' is either concurrent to tr or $tr'' \prec_h tr$ holds. As $tr' \prec_{h_s} tr$ holds and by the real-time order preservation property, tr' is either concurrent to tr or $tr' \prec_h tr$ holds. As both transactions write the same value to var, they belong to the same thread by the TIV-assumption. Let this thread be t. Assume tr'' is concurrent to tr. The most recent transaction of t writing to var real-time ordered before tr and all transactions of t concurrent to tr'' write pairwise different values by the TIV-assumption. As tr' is one of these transactions and writes val to var, tr'' cannot be one of these transactions. So, it must be real-time ordered before the most recent committed transaction of t writing to var and real-time ordered before tr. This is a contradiction to the RR-assumption as there is one committed transaction writing to var real-time ordered between tr'' and tr. So, the claim holds.

$\boldsymbol{\neg(\exists tr'' \in Tr : var \in WS_h^{vo}(tr'') \wedge co_h(tr'') \wedge tr' \prec_h tr'' \prec_h tr)}$ **:**
Assume it the opposite holds then by real-time preservation it also holds that

$$(\exists tr'' \in Tr : var \in WS_{h_s}^{vo}(tr'') \wedge co_{h_s}(tr'') \wedge tr' \prec_{h_s} tr'' \prec_{h_s} tr)$$

which is in contradiction to $(tr', tr, var) \in h_s.RF_c$. So the claim holds.

$\boldsymbol{h.RF_{\mathrm{val}} \subseteq h_s.RF_c}$ **:**
Consider an arbitrary rf-element $(tr', tr, var) \in h.RF_{val}$. We show that $(tr', tr, var) \in h_s.RF_c$ by showing that all conditions of Definition 37 hold.

$\boldsymbol{var \in WS_h^{vo}(tr'), var \in RS_h^{vo}(tr)}$ **:**
By Definition 38, it holds that

$$(var, val) \in WS_h(tr') \text{ and } (var, val) \in RS_h(tr).$$

From this it follows that

$$(var, val) \in WS_{h_s}(tr') \text{ and } (var, val) \in RS_{h_s}(tr).$$

$\boldsymbol{tr' \prec_{h_s} tr}$ **:**
It is given that h_s is legal. As we argued in the previous case, in h there is exactly one transaction writing val to var s.t. it is either the most recent committed transaction real-time ordered before tr or it is a commit pending or committed transaction concurrent to tr. Thus, if the negation of the claim holds, which is $tr \prec_{h_s} tr'$ (as h_s is serial), then h_s cannot be legal.

$\boldsymbol{co_{h_s}(tr')}$ **:**
Since $(tr', tr, var) \in h.RF_{val}$ holds, tr' is visible and either committed or commit pending in h and thus committed in h_s.

$\boldsymbol{\neg(\exists tr'' \in Tr : var \in WS^{vo}_{h_s}(tr'') \wedge co_{h_s}(tr'') \wedge tr \prec_{h_s} tr'' \prec_{h_s} tr')}$ **:**
Assume such a tr'' would exist. As we argued in the previous case, in h there is exactly one transaction writing val to var s.t. it is either the most recent committed transaction real-time ordered before tr or it is a commit pending or committed transaction concurrent to tr. Thus, tr'' would write a value that is not val to var, and thus h_s would not be legal. It is legal by assumption, and thus such a tr'' cannot exist.

$\boldsymbol{\leftarrow}$**:**
Consider an arbitrary rf-element $(tr', tr, var) \in h.RF_{val}$. It follows that $(tr', tr, var) \in h_s.RF_c$. It holds that there exists a value val s.t. $(var, val) \in WS_{h_s}(tr')$, $(var, val) \in RS_{h_s}(tr)$ and tr' is the most recent committed transaction writing to var before transaction tr in h_s. Thus, the read of var of tr

obeys the sequential specification of read write registers. As we have shown this for an arbitrary rf-element, the history is legal.

□

Lemma 24 (Minimal completion legal iff legal completion exists)**.** *Given a g-history h and a candidate h_c, a serial completion for h_c that is an OP^- witness for h exists iff $mCl(h_c)$ is legal.*

Proof. We prove both directions.

←:
It is trivial to see that the minimal completion is a serial completion and if it is legal by Lemma 21 it is also an OP^--witness for h.

→:
Let h_{co} be an arbitrary legal serial completion of h_c. We show that if h_{co} is legal, $mCl(h_c)$ is also legal. It is trivially true that $h_{co}.RT = mCl(h_c).RT$ as these g-histories only differ in which commit pending transactions of h were aborted and which were committed. We show that $h_{co}.RF_c = mCl(h_c).RF_c$ by contradiction. Any read in a g-history reads from some transaction. Thus, it is sufficient assume the following and show that this leads to a contradiction.

$$\exists tr, tr', tr'' \in Tr : (tr', tr, x) \in h_{co}.RF_c, (tr'', tr, x) \in mCl(h_c).RF_c \text{ and } tr'' \neq tr'.$$

Let tr, tr', tr'' be transactions s.t. the formula after the quantifiers is true. Assuming this, tr' must be visible in h as $h.RF_{val} = h_{co}.RF_c$. By definition of a minimal completion, that means it is committed in $mCl(h_c)$. Also, tr' is committed in h_{co} by definition of $h_{co}.RF_c$. Thus, if tr does not read x from tr' in $mCl(h_c)$, the following must hold:

$$tr'' \prec_{mCl(h_c)} tr,$$

and

$$\neg(tr'' \prec_{mCl(h_c)} tr' \prec_{mCl(h_c)} tr).$$

Similarly, in h_{co} the following must hold:

$$tr' \prec_{h_{co}} tr,$$

and

$$\neg(tr' \prec_{h_{co}} tr'' \prec_{h_{co}} tr).$$

These facts are in contradiction to the fact that both completions have the same real-time order. This proves the claim.

□

Lemma 25. *Given an input g-history h, an OP^--witness for it exists iff there exists a candidate h_c s.t. $mCl(h_c)$ is legal.*

Proof. We show both directions separately.

→:
Given an OP^--witness h_s, by Lemma 22 it is a serial completion of a candidate h'_c. Then, from Lemma 24 it follows that $mCl(h'_c)$ is legal.
←:
This follows directly from Lemma 24.

□

Lemma 26 (Conditions for fixed rf-elements in g-histories)**.** *Given a g-history h, it holds that $h.RF^{fix}_{val} = h.RF_{val}$.*

Proof. We show for an rf-element $(tr, tr', var) \in h.RF_{val}$ and for an arbitrary g-event ev , that

$$(tr, tr', var) \in (h \cdot ev).RF_{val},$$

holds. By induction, this implies the claim for arbitrary event sequences.

We show $(tr, tr', var) \in (h \cdot ev).RF_{val}$ along the conditions of Definition 38.

- $(var, val) \in WS_{h \cdot ev}(tr)$: It is true that $(var, val) \in WS_h(tr)$ and no event is removed when appending ev, so it is true for $h \cdot ev$

- $(var, val) \in RS_{h \cdot ev}(tr')$: True, argumentation is analogue to the one above.
- $\neg(\exists tr'' \in Tr : var \in WS^{vo}_{h \cdot ev}(tr'') \wedge co_{h \cdot ev}(tr'') \wedge tr \prec_{h \cdot ev} tr'' \prec_{h \cdot ev} tr')$: As before, the formula is true when only considering h instead of $h \cdot ev$. Appending ev means it is ordered after the last events of both tr and tr'. Thus, its transaction cannot be real-time ordered in between tr and tr'.

Thus, $(tr, tr', x) \in (h \cdot ev).RF_{val}$ holds if $(tr, tr', x) \in h.RF_{val}$. Applying induction, it holds that

$$\forall seq \in Ev^* : (tr, tr', x) \in (h \cdot seq).RF_{val},$$

which then implies

$$h.RF^{fix}_{val} = h.RF_{val}.$$

□

Lemma 27 (Conditions for fixed rf-elements in candidates)**.** *Given a candidate h_c of g-history h and an arbitrary rf-element $rf \in h_c.RF$ with $rf = (tr, tr', var)$, rf is in $h_c.RF^{fix}_c$ iff*

1. *rf is not abortable,*
2. *$\neg(\exists tr'' \in Tr : unfin_{h_c}(tr'') \wedge \neg(comP_{h_c}(tr'')) \wedge tr \prec_{h_c} tr'' \wedge \neg(tr' \prec_{h_c} tr''))$,*
3. *$\neg(\exists tr'' \in Tr : comP_{h_c}(tr'') \wedge x \in WS_{h_c}(tr) \wedge tr \prec_{h_c} tr'' \wedge \neg(tr' \prec_{h_c} tr''))$*
4. *and $(\exists tr'' : fin_{h_c}(tr'') \wedge ev^{rd,rf}_{h_c} <_{h_c} ev^{tr'',ls}_{h_c}) \vee (\forall t : ev^{rd,rf}_{h_c} <_{h_c} ev^{ls,t}_{h_c})$.*

Proof. We show both directions separately.

$\rightarrow$:
We show that any rf-element $(tr, tr', var) \in h_c.RF^{fix}_c$ fulfils the above conditions. We do this by contradiction and show that if one of the conditions does not hold there is a contradiction.

rf **is not abortable:**
Assume *rf* was abortable. Then, consider a sequence *seq* just consisting of an abort event of tr'. If h_c is extended by *seq*, then *rf* is not present in the extension, which contradicts the assumption.

$\boldsymbol{\neg(\exists tr'' \in Tr : unfin_{h_c}(tr'') \wedge \neg(comP_{h_c}(tr'')) \wedge tr \prec_{h_c} tr'' \wedge \neg(tr' \prec_{h_c} tr''))}$ **:**
Assume such an unfinished and not commit pending transaction exists. If *seq* consists only of a write on x (if it does not already exist), a commit invoke and a commit of that transaction, then in an extension of h_c by *seq* this transaction is then read by tr'' meaning *rf* is removed.

$\boldsymbol{\neg(\exists tr'' \in Tr : comP_{h_c}(tr'') \wedge x \in WS_{h_c}(tr) \wedge tr \prec_{h_c} tr'' \wedge \neg(tr' \prec_{h_c} tr''))}$ **:**
Note that tr'' cannot be visible in h, else *rf* does not exist in h_c. If *seq* consists of only a commit event of tr'', then in an extension of h_c by *seq rf* is removed.

$\boldsymbol{(\exists tr'' : fin_{h_c}(tr'') \wedge ev_{h_c}^{rd,rf} <_{h_c} ev_{h_c}^{tr'',ls}) \vee (\forall t : ev_{h_c}^{rd,rf} <_{h_c} ev_{h_c}^{ls,t})}$ **:**
Assume both parts of the disjunction are not true. Then, there exists a transaction which has no event after tr'. Consider the sequence *seq* consisting of a begin, write on x, commit invoke and commit of that transaction. The begin can be inserted in between *tr* and tr' as the last commit or abort of a transaction is before tr' and the last event of its thread was before tr'. In an extension of h_c by *seq*, this transaction is then read by tr'' meaning *rf* is removed.

$\leftarrow$:

We show for an rf-element $(tr, tr', x) \in h_c.RF_c$ fulfilling the above conditions that $(tr, tr', x) \in h_c.RF_c^{fix}$. We first show after the insertion of an arbitrary event ev into h_c all conditions still hold in the resulting candidate.

***rf* is not abortable:**
Inserting an event does not remove events from h_c; thus, tr stays committed.

$\boldsymbol{\neg(\exists tr'' \in Tr : unfin_{h_c}(tr'') \land \neg(comP_{h_c}(tr'')) \land tr \prec_{h_c} tr'' \land \neg(tr' \prec_{h_c} tr''))}$ **:**
Given that this condition holds in h_c, no unfinished and not commit pending transaction in between tr and tr' exists in h_c. Thus, the only option for this to hold after inserting an event is for that event to be a begin. As the fourth condition also holds for h_c, either all threads have an event after tr' or there is a commit or abort event after tr'. In the first case, any begin is inserted after the last event of its thread and thus after tr'. In the second case, any begin is inserted after the last commit or abort of the candidate and thus after tr'. Thus, after inserting any event the second condition still holds.

$\boldsymbol{\neg(\exists tr'' \in Tr : comP_{h_c}(tr'') \land x \in WS_{h_c}(tr) \land tr \prec_{h_c} tr'' \land \neg(tr' \prec_{h_c} tr''))}$ **:**
As we have shown in the previous case, no unfinished transaction that is not commit pending has its last event in between tr and tr'' in any insertion. Thus, no commit invoke can be inserted in between tr and tr'. So, any commit pending transaction writing to x would have been present in h_c, which it is not as the conditions hold for h_c.

$\boldsymbol{(\exists tr'' : fin_{h_c}(tr'') \land ev_{h_c}^{rd,rf} <_{h_c} ev_{h_c}^{tr'',ls}) \lor (\forall t : ev_{h_c}^{rd,rf} <_{h_c} ev_{h_c}^{ls,t})}$ **:**
The insertion of an event does not remove events or their relative order and

new events of a thread are are inserted after any last event of that thread. Thus, the condition holds after inserting any event.

Next, we show that for any event ev, $\forall h_{c,ins} \in ins(h_c, ev) : rf \in h_{c,ins}.RF_c$ holds.

***ev* is a begin, write, abort or commit invoke:**
In this case, $h_{c,ins}$ has the same committed transactions as h_c. For the minimal completions the committed transactions may differ if there are transactions visible in $h \cdot ev$ that are not in h. Appending a begin, write, abort or commit invoke at the end of h does not change the value reads-from relation of it. So, the committed transactions in the minimal completions of h_c and $h_{c,ins}$ are identical too, and their order is identical as inserting an event does not reorder other events. Thus, $rf \in h_{c,ins}.RF_c$ holds.

***ev* is a read:**
Appending a read at the end of h changes the value reads-from relation of it thus transactions not visible in h can be visible in $h \cdot ev$. No transaction that is writing on x and is commit pending exists in h_c by the conditions; thus, such a transaction cannot become visible in $h \cdot ev$. So tr and tr' still have the same events relative order and no transaction writing to x that is committed in the minimal completion exists. Thus, $rf \in h_{c,ins}.RF_c$ holds.

***ev* is a commit:**
The event ev cannot be a commit of a transaction that is writing to x and is in between tr and tr' in h_c as no such transaction exists by the conditions. So, tr and tr' still have the same events and relative order. In addition, no transaction writing to x that is committed in the minimal completion exists in between them. So $rf \in h_{c,ins}.RF_c$ holds. □

Lemma 28 (Conditions for unreadable transactions in a g-history)**.** *For a g-*

history h, a finished transaction $tr \in h$ is unreadable iff

1. $ab_h(tr)$
2. *or* $\forall var \in WS_h^{vo}(tr), \exists tr' \in Tr : co_h(tr') \wedge var \in WS_h^{vo}(tr') \wedge tr \prec_h tr'$.

Proof. We first show that $tr \in ur(h_c)$ implies both conditions. We do so by contraposition. First, $\neg(ab_h(tr))$ is equivalent to $co_h(tr)$ as tr is finished. Second, assume it holds that

$$\exists var \in WS_h^{vo}(tr), \neg(\exists tr' \in Tr : co_h(tr') \wedge var \in WS_h^{vo}(tr') \wedge tr \prec_h tr').$$

Let var be any arbitrary variable in $WS_h^{vo}(tr)$, then it is easy to see appending a read of a variable var leads to it reading from tr following Definition 38. This implies $tr \notin ur(h)$.

We show that both conditions imply $tr \in ur(h)$ by contraposition. Assume there exists a sequence of g-events seq with a read of a transaction tr' s.t.

$$\exists rf = (tr, tr', var') \in (h \cdot seq).RF_{val} : (tr, tr', var') \notin h.RF_{val}.$$

W.l.o.g. we assume $ev_{h \cdot seq}^{rd,rf}$ is the first element of seq. By our assumption, tr must be commit pending or committed to be read. As it is finished by the prerequisites of this lemma, it must be committed. This means that the first condition does not hold as $co_h(tr) \rightarrow \neg(ab_h(tr))$ is true. By Definition 38 (value reads-from relation), it must also hold that

- $(var', val) \in WS_h(tr)$,
- $(var', val) \in RS_h(tr')$,
- $\neg(\exists tr'' \in Tr : var' \in WS_h^{vo}(tr'') \wedge co_h(tr'') \wedge tr \prec_h tr'' \prec_h tr')$.

The last statement is contradictory to the second condition of this lemma:

$$\forall var \in WS_h^{vo}(tr), \exists tr' \in Tr : co_h(tr') \wedge var \in WS_h^{vo}(tr') \wedge tr \prec_h tr'.$$

Thus, the second condition is false as well, which proves the contraposition overall.

□

Lemma 29 (Conditions for unreadable transactions in a candidate)**.** *Given a finished transaction tr occurring in candidate h_c of g-history h, then tr is unreadable iff*

1. *$ab_{h_c}(tr)$ (Aborted)*
2. *or $co_{h_c}(tr)$ and*
   - *$\forall var \in WS^{vo}_{h_c}(tr), \exists tr' \in Tr :$ $co_{h_c}(tr') \land var \in WS^{vo}_{h_c}(tr') \land tr \prec_{h_c} tr'$ (Overwritten before end)*
   - *and for all unfinished and not commit pending or abort pending transactions tr' in h_c*

     (a) $\neg(tr \prec_{h_c} tr')$ (Ordered after unfin. tr.)

     (b) or $\forall var \in WS^{vo}_{h_c}(tr), \exists tr'' \in Tr :$ $co_{h_c}(tr'') \land var \in WS^{vo}_{h_c}(tr'') \land tr \prec_{h_c} tr'' \prec_{h_c} tr'$ (Overwr. bef. unfin. tr.).

Proof. To make this proof easier to understand we abbreviate two of the subconditions of the lemma's conditions:

$$obe(h_c, tr) = \forall var \in WS^{vo}_{h_c}(tr), \exists tr' \in Tr : co_{h_c}(tr') \land var \in WS^{vo}_{h_c}(tr') \land tr \prec_{h_c} tr'$$

and

$$obt(h_c, tr, tr') = \forall var \in WS^{vo}_{h_c}(tr), \exists tr'' \in Tr : co_{h_c}(tr'') \land var \in WS^{vo}_{h_c}(tr'') \land tr \prec_{h_c} tr'' \prec_{h_c} tr'.$$

Additionally, we denote the set of all unfinished and not commit or abort pending transactions in h_c as Tr_{unc,h_c}. We show both directions separately.

$\rightarrow$:
This direction is

$$tr \in ur(h_c)$$
$$\rightarrow$$
$$ab_{h_c}(tr) \vee (co_{h_c}(tr) \wedge obe(h_c, tr) \wedge (\forall tr' \in Tr_{unc,h_c} : \neg(tr \prec_{h_c} tr') \vee obt(h_c, tr, tr'))).$$

We show this by contraposition. Note that by the lemma's assumption tr must be finished so if it is not aborted it is committed and the other way around. Also, h_c is serial, so either a transaction is ordered before another or after another. There are no concurrent transactions So, the contraposition is

$$co_{h_c}(tr) \wedge (ab_{tr}(h_c) \vee \neg(obe(h_c, tr)) \vee (\exists tr' \in Tr_{unc,h_c} : tr \prec_{h_c} tr' \wedge \neg obt(h_c, tr, tr')))$$
$$\rightarrow h_c \notin ur(tr).$$

It holds that tr cannot be committed and aborted at the same time so we can simplify to

$$(co_{h_c}(tr)) \wedge (\neg(obe(h_c, tr)) \vee (\exists tr' \in Tr_{unc,h_c} : tr \prec_{h_c} tr' \wedge \neg obt(h_c, tr, tr'))) \rightarrow tr \notin ur(h_c).$$

To prove this we show 2 things:

1. $(co_{h_c}(tr) \wedge \neg obe(h_c, tr)) \rightarrow tr \notin ur(h_c)$
2. and $(co_{h_c}(tr) \wedge (\exists tr' \in Tr_{unc,h_c} : tr \prec_{h_c} tr' \wedge \neg obt(h_c, tr, tr'))) \rightarrow tr \notin ur(h_c)$.

We show both separately.

$(co_{h_c}(tr) \wedge \neg obe(h_c, tr)) \rightarrow tr \notin ur(h_c)$:

If the write set of tr is empty, it is trivially unreadable. Thus, in the further we assume it to be non-empty. Let var be an arbitrary member of this write set and let val be the value written to var by tr. We do a case distinction over whether there exists at least one thread that has no unfinished transaction in h_c. If yes , let this thread be t' and tr' be a transaction identifier that is unused in h_c. Let $Tr_{vis,var}$ be the set of abortable transactions writing to var. Assume a sequence $seq = \bullet_{tr_x \in Tr_{vis,var}}(\mathbf{A}^{tr_x}_{thr(tr_x)})\mathbf{B}^{tr'}_{t'}\mathbf{R}^{tr'}_{t'}(var, val)$. Trivially it holds that $ins(h_c, \bullet_{tr_x \in Tr_{vis,var}}(\mathbf{A}^{tr_x}_{thr(tr_x)})) \cdot \mathbf{B}^{tr'}_{t'}\mathbf{R}^{tr'}_{t'}(var, val) \in ins(h_c, seq)$. We denote this g-history as $h_{c,ins}$. As tr is the last committed transaction writing on var before the end of the minimal completion of h_c and all commit pending transactions (in h_c) are aborted after inserting the sequence, it holds that

$$(tr, tr', var) \in h_{c,ins}.RF_c.$$

This means $tr \notin ur(h_c)$. If there exists no thread that has no unfinished transaction in h_c, then let tr' be an arbitrary unfinished transaction and t' be its thread. Let tr'' be an unused transaction identifier. Then, set

$$seq = \bullet_{tr_x \in Tr_{vis,var}}(\mathbf{A}^{tr_x}_{thr(tr_x)})\mathbf{A}^{tr'}_{t'}\mathbf{B}^{tr''}_{t'}\mathbf{R}^{tr''}_{t'}(var, val)$$

and the rest of the proof is analogue to the one above except the abort is inserted at the end of the respective transaction and the begin and read are appended at the end of the candidate. Aborting tr' ensures it does not overwrite tr in the extension.

$\boldsymbol{(co_{h_c}(tr) \wedge (\exists tr' \in Tr_{unc,h_c} : tr \prec_{h_c} tr' \wedge \neg obt(h_c, tr, tr'))) \rightarrow tr \notin ur(h_c)}$ **:**
Let tr' be transaction s.t. $tr \prec_{h_c} tr'$ and $\neg obt(h_c, tr, tr')$ holds. The negation of $obt(h_c, tr, tr')$ is

$$\begin{aligned}&\exists var \in WS^{vo}_{h_c}(tr), \forall tr'' \in Tr :\\ &\neg co_{h_c}(tr'') \vee var \notin WS^{vo}_{h_c}(tr'') \vee \neg(tr \prec_{h_c} tr'' \prec_{h_c} tr').\end{aligned}$$

Let var be a variable fulfilling the remaining formula after the exists quantifier. Let $seq = \bullet_{tr_x \in Tr_{vis,var}}(\mathbf{A}^{tr_x}_{thr(tr_x)})\mathbf{R}^{tr'}_{thr(tr')}(var, val)$ be the event sequence, where val is the value written by tr on var. Let $h_{c,ins}$ be the only member of the set $ins(h_c, seq)$. Note that the read is inserted directly after the last event of its transaction. Thus, all transactions have the same order and write sets

in both histories.
We show $(tr, tr', var) \in h_{c,ins}.RF_c$. Trivially the first four conditions of Definition 37 hold, tr is committed in $h_{mCl(h_{c,ins})}$, tr' is ordered after tr and tr' and tr have var in their write set and read set, respectively. It is left to show that

$$\neg(\exists tr'' \in Tr : var \in WS^{vo}_{mCl(h_{c,ins})}(tr'') \wedge co_{mCl(h_{c,ins})}(tr'') \wedge \\ tr \prec_{mCl(h_{c,ins})} tr'' \prec_{mCl(h_{c,ins})} tr').$$

As all commit pending transactions in h_c writing to var are aborted in $h_{c,ins}$ and tr is the committed last writer on var before tr' in h_c, it holds that in $mCl(h_{c,ins})$ it is the last committed writer on var before tr'. Thus, $(tr, tr', var) \in h_{c,ins}.RF_c$ holds.

$\leftarrow$:
We now show the other direction.

$$ab_{h_c}(tr) \vee (co_{h_c}(tr) \wedge obe(h_c, tr) \wedge (\forall tr' \in Tr_{unc,h_c} : tr' \prec_{h_c} tr \vee obt(h_c, tr, tr'))) \rightarrow tr \in ur(h_c).$$

It is obvious that if tr is aborted, it is unreadable, so we show

$$co_{h_c}(tr) \wedge obe(h_c, tr) \wedge (\forall tr' \in Tr_{unc,h_c} : tr' \prec_{h_c} tr \vee obt(h_c, tr, tr')) \rightarrow tr \in ur(h_c).$$

We assume tr is being read in an extension and show that this is in contradiction to the left-hand side of the formula. Let seq be an extension s.t.

$$\exists h'_c \in ins(h_c, seq), \exists \mathbf{R}^{tr'}_{thr(tr')}(var, val) \in seq : (tr, tr', var) \in h'_c.RF_c.$$

Let h'_c,tr'' and var be instances of the quantified variables h'_c, tr' and var, respectively, s.t. the interpretation of the formula is true. We first do a case distinction over tr''.

tr'' is an unfinished transaction in h_c:

This is in contradiction to

$$(\forall tr' \in Tr_{unc,h_c} : tr' \prec_{h_c} tr \vee obt(h_c, tr, tr')).$$

We show why by inserting tr'' as an instance of the quantified tr'.

$$(tr'' \prec_{h_c} tr \vee obt(h_c, tr, tr'')).$$

If it holds that

$$tr'' \prec_{h_c} tr,$$

then this also holds in h'_c and Condition 1 ($tr \prec_{mCl(h_c)} tr''$) of the conflicts reads-from relation definition (Definition 37) is false. If on the other hand $obt(h_c, tr, tr'')$ holds, then

$$\exists tr''' \in Tr : co_{h_c}(tr''') \wedge var \in WS^{vo}_{h_c}(tr''') \wedge tr \prec_{h_c} tr''' \prec_{h_c} tr'',$$

must be true, which it would also then be in h'_c. Then, this is in contradiction to Condition 3 of the conflict reads-from relation definition (Definition 37). Thus, in both cases tr' reading from tr leads to a contradiction.

tr'' does not exist in h_c

This is in contradiction to $obe(h_c, tr)$, which expanded and with var inserted for the all quantified variable is

$$\exists tr' \in Tr : co_{h_c}(tr') \wedge var \in WS^{vo}_{h_c}(tr') \wedge tr \prec_{h_c} tr'.$$

All of this is also true in h'_c as finished transactions are not modified by ins and existing events are not reordered. Let tr' be an arbitrary transaction instance which leads to a true interpretation of the formula. As this transaction is committed in h_c, it holds that

$$tr \prec_{h'_c} tr' \prec_{h'_c} tr'' \wedge var \in WS^{vo}_{h_c}(tr').$$

This and

$$(tr, tr'', var) \in ins(h_c, seq).RF_c,$$

cannot both be true because the first formula is in contradiction to Condition 3 of the conflicts reads-from relation definition (Definition 37).

□

Lemma 30. *Given an unreadable transaction tr in a candidate h_c in an hc-pair (h, h_c) and an arbitrary extension of it by a sequence seq, $(h \cdot seq, h'_c)$ the following property holds:*

$$\{rf \in h_c.RF_c \mid tr \in rf\} \neq \{rf \in h'_c.RF_c \mid tr \in rf\}$$
$$\rightarrow \forall seq' \in Ev^*, \forall h''_c \in ins(h'_c, seq') : (h \cdot seq \cdot seq', h''_c.RF_c) \textit{ is not consistent.}$$

Proof. If $\{rf \in h_c.RF_c \mid tr \in rf\} \neq \{rf \in h'_c.RF_c \mid tr \in rf\}$ holds, then either the first set contains an rf-element the second set does not or the other way around. Let rf be that rf-element, we do a case distinction of which of both cases holds.

$\boldsymbol{rf \in \{rf \in h_c.RF_c \mid tr \in rf\} \wedge rf \notin \{rf \in h'_c.RF_c \mid tr \in rf\}}$ **:**
We consider two subcases of this case, either $rf = (tr, tr', x)$ or $rf = (tr', tr, x)$ where in both cases $tr' \in Tr$ holds.

$\boldsymbol{rf = (tr, tr', \mathrm{x})}$ **:**
If $rf \notin \{rf \in h'_c.RF_c \mid tr \in rf\}$ is true, then there must exists $tr'' \in Tr$, $tr'' \neq tr'$ s.t. $(tr'', tr', x) \in \{rf \in h'_c.RF_c \mid tr \in rf\}$. As (h, h_c) is consistent, $(tr, tr', x) \in h.RF_{val}$ holds, and as it is fixed by Lemma 26 in h, $(tr, tr', x) \in (h \cdot seq).RF_{val}$ holds. Then, $(h \cdot seq, h'_c)$ is not consistent. The relative order of events of h'_c stays identical in extensions of it and tr is unreadable in h'_c as it is unreadable in h_c. So, there is no extension of it where tr' reads x from tr. Thus, any extension of $(h \cdot seq, h'_c)$ is also not consistent.

$\boldsymbol{rf = (tr', tr, \mathrm{x})}$ **:**

If $rf \notin \{rf \in h_c'.RF_c \mid tr \in rf\}$ holds, then there must exists $tr'' \in Tr$, $tr'' \neq tr'$ s.t. $(tr'', tr, x) \in \{rf \in h_c'.RF_c \mid tr \in rf\}$. As (h, h_c) is consistent, $(tr', tr, x) \in h.RF_{val}$ holds, and as it is fixed by Lemma 26 in h, (tr', tr, x) is also in $(h \cdot seq).RF_{val}$. Then, $(h \cdot seq, h_c')$ is not consistent. Any extension of $h \cdot seq$ contains (tr', tr, x) in its value reads-from relation as the rf-element is fixed in h. As the relative order of events in extensions of h_c' stays identical and visible transactions in a g-history are also visible in all of its extensions, there is no extension of h_c' where tr reads x from tr'. Thus, any extension of $(h \cdot seq, h_c')$ is not consistent.

$\boldsymbol{rf \in \{rf \in h_c'.RF_c \mid tr \in rf\} \wedge rf \notin \{rf \in h_c.RF_c \mid tr \in rf\}}$ **:**
Note that tr is unreadable in h_c, so $\{rf \in h_c'.RF_c \mid tr \in rf\}$ cannot contain an rf-element (tr, tr', x) with $tr' \in Tr$ and $x \in Var$ which $\{rf \in h_c.RF_c \mid tr \in rf\}$ does not contain. So the only option is that it contains an rf-element (tr', tr, x) with $tr' \in Tr$ that $\{rf \in h_c.RF_c \mid tr \in rf\}$ does not contain. As h_c' is an extension of h_c, it holds that $h_c.RF_c$ must contain (tr'', tr, x) with $tr'' \in Tr$ and $tr'' \neq tr$. This is a subcase of the previous case, and the claim is proven above for this case.

Thus, the overall claim holds.

□

Lemma 32 (Removal function correctness)**.** *Given a consistent hc-pair (h, h_c) and a set of transactions Tr^-, it holds for $(h \backslash Tr^-, h_c \backslash Tr^-)$ that*

1. $(h \backslash Tr^-).RF_{val} = h.RF_{val} \backslash \{rf \in h.RF_{val} \mid \exists tr \in Tr^- : tr \in rf\}$
2. *and* $(h_c \backslash Tr^-).RF_c = h_c.RF_c \backslash \{rf \in h_c.RF_c \mid \exists tr \in Tr^- : tr \in rf\}$.

Proof. We show the first condition. We show both subset or equal relations.

$\boldsymbol{(h \backslash Tr^{-}).RF_{\mathrm{val}} \subseteq h.RF_{val} \backslash \{rf \in h.RF_{val} \mid \exists tr \in Tr^{-} : tr \in rf\}}$ **:**
Consider an arbitrary rf-element $rf = (tr, tr', x)$ of $(h \backslash Tr^{-}).RF_{val}$. We show that $rf \in h.RF_{val}$ and $rf \notin \{rf \in h.RF_{val} \mid \exists tr \in Tr^{-} : tr \in rf\}$.

$\boldsymbol{rf \in h.RF_{val}}$ **:**
First, tr and tr' cannot be in Tr^{-} as else rf would not exist in $(h \backslash Tr^{-}).RF_{val}$. Also, the removal function does not add g-events to h and the relative order between events that are not removed is identical to h. As $rf \in (h \backslash Tr^{-}).RF_{val}$ holds, there must exist $val \in Val$ s.t. $(var, val) \in WS_{h \backslash Tr^{-}}(tr)$ and $(var, val) \in RS_{h \backslash Tr^{-}}(tr')$ hold. We fix this value as val. We prove the claim along Definition 38, using val as the value for which the other conditions hold.

$\boldsymbol{(var, val) \in WS_h(tr), (var, val) \in RS_h(tr')}$ **:**
The write event on var of tr and the read event on var of tr' exist in $(h \backslash Tr^{-})$ as $rf \in (h \backslash Tr^{-}).RF_{val}$. Both also exist in h as the removal function does not add or modify (except delete) existing events of h when applied to it. So, the claim holds.

$\mathbf{Inv}^{\boldsymbol{tr}}_{\boldsymbol{thr(tr)}}(\mathbf{C}) <_{\boldsymbol{h}} \mathbf{R}^{\boldsymbol{tr'}}_{\boldsymbol{thr(tr')}}(\boldsymbol{var}, \boldsymbol{val})$:
As $rf \in (h \backslash Tr^{-}).RF_{val}$ holds, $\mathbf{Inv}^{tr}_{thr(tr)}(\mathbf{C}) <_{h \backslash Tr^{-}} \mathbf{R}^{tr'}_{thr(tr')}(var, val)$ holds. Both events are not removed by the removal function as $tr, tr' \notin Tr^{-}$ and the read event of tr' on var exists after applying the removal function as else $rf \notin (h \backslash Tr^{-}).RF_{val}$ would hold. Thus, the claim holds.

$\neg(\exists \boldsymbol{tr''} \in \boldsymbol{Tr} : \boldsymbol{var} \in \boldsymbol{WS_h^{vo}(tr'')} \land \boldsymbol{co_h(tr'')} \land \boldsymbol{tr} \prec_h \boldsymbol{tr''} \prec_h \boldsymbol{tr'}) :$
It is given that

$$\neg(\exists tr'' \in Tr : var \in WS^{vo}_{his\setminus Tr^-}(tr'') \land co_{his\setminus Tr^-}(tr'')$$
$$\land tr \prec_{his\setminus Tr^-} tr'' \prec_{his\setminus Tr^-} tr').$$

Assume such a transaction exists in h, then $(tr, tr', var) \notin h.RF_{val}$ follows. Let tr'' be the transaction s.t. $(tr'', tr', var) \in h.RF_{val}$. If $tr'' \notin Tr^-$, then (tr'', tr', var) would still fulfil all conditions of Definition 38 in $h\setminus Tr^-$. This is because only read events of it can possibly be removed, the read event of tr' still exists as tr' and tr'' are not in Tr^- and the relative order of events remains unchanged and no events are added. Then, $(tr, tr', var) \notin (h\setminus Tr^-).RF_{val}$ follows which is a contradiction. If $tr'' \in Tr^-$ holds, then the read of tr' would not exist in $(h\setminus Tr^-)$. Then, $(tr, tr', var) \notin (h\setminus Tr^-).RF_{val}$ follows which is a contradiction.

$\boldsymbol{rf \notin \{rf \in h.RF_{val} \mid \exists tr \in Tr^- : tr \in rf\}} :$
Assume the opposite, then either $tr \in Tr^-$ or $tr' \in Tr^-$. In both cases the removal function would remove the read event of the reading transaction from h. So, if $rf \in (h\setminus Tr^-).RF_{val}$, then it cannot be in $\{rf \in h.RF_{val} \mid \exists tr \in Tr^- : tr \in rf\}$.

We show the subset or equal relation in the other direction.

$\boldsymbol{h.RF_{val}\setminus\{rf \in h.RF_{val} \mid \exists tr \in Tr^- : tr \in rf\} \subseteq (h\setminus Tr^-).RF_{val}} :$

Consider an arbitrary rf-element
$rf = (tr, tr', var)$ of $h.RF_{val}\setminus\{rf \in h \mid \exists tr \in Tr^- : tr \in rf\}$. Let val be the value s.t. rf meets the conditions of Definition 38. We show $rf \in (h\setminus Tr^-).RF_{val}$ by showing the conditions of Definition 38 are fulfilled for val. First note that by definition it holds that $tr \notin Tr^-$ and $tr' \notin Tr^-$. Thus, no begins, writes or

commit/abort invoke or responses are removed from both transactions by the removal function. Also, the read on var of tr' is not removed by the removal function as neither the first nor the second condition of the removal function hold for it. Its own transaction is not in Tr^- and its writer in the value reads-from relation is tr which is also not in Tr^-. The removal function does not add g-events to h or changes the relative order of the g-events it does not remove. Now, we show the conditions of Definition 38 are fulfilled for val.

$\boldsymbol{(var, val) \in WS_{h\backslash Tr^-}(tr), (var, val) \in RS_{h\backslash Tr^-}(tr')}$ **:**
As discussed above, the write event of tr for var remains unchanged and writes val. In addition the read event of tr for var remains unchanged and reads val. Thus, this condition holds.

$\mathbf{Inv}^{\boldsymbol{tr}}_{\boldsymbol{thr(tr)}}(\mathbf{C}) <_{\boldsymbol{h\backslash Tr^-}} \mathbf{R}^{\boldsymbol{tr'}}_{\boldsymbol{thr(tr')}}(\boldsymbol{var}, \boldsymbol{val})$:
As discussed above, both events exist in $h\backslash Tr^-$ and have the same relative order. Thus, the claim holds.

$$\boldsymbol{\neg(\exists tr'' \in Tr : var \in WS^{vo}_{h\backslash Tr^-}(tr'') \wedge co_{h\backslash Tr^-}(tr'')}$$
$$\boldsymbol{\wedge tr \prec_{h\backslash Tr^-} tr'' \prec_{h\backslash Tr^-} tr') :}$$

Assume a tr'' fulfilling the above conditions for h exists. Then there must exist a transaction $tr''' \neq tr$ s.t. $(tr''', tr', var) \in h.RF_{val}$.
That means $(tr, tr', var) \notin h.RF_{val}$, thus it also cannot be in

$$h.RF_{val}\backslash\{rf \in h \mid \exists tr \in Tr^- : tr \in rf\},$$

which is a contradiction to the assumption. Thus, such a transaction tr'' as assumed above does not exist in h. This means it also cannot exist in $h\backslash Tr^-$ as the removal function does not reorder or add events. Thus, the condition holds.

We show the second condition. We show both subset or equal relations.

$\boldsymbol{(h_c \backslash Tr^-).RF_c \subseteq h_c.RF_c \backslash \{rf \in h.RF_c \mid \exists tr \in Tr^- : tr \in rf\} :}$
Let h_s be the minimal completion of $h_c \backslash Tr^-$ in the context of $(h \backslash Tr^-, h_c \backslash Tr^-)$. Consider an arbitrary rf-element $rf = (tr, tr', x)$ of $h_s.RF_c$. We show that $rf \in h_c.RF_c$ and $rf \notin \{rf \in h_c.RF_c \mid \exists tr \in Tr^- : tr \in rf\}$.

$\boldsymbol{rf \in h_c.RF_c :}$
First, tr and tr' cannot be in Tr^- as else they would not exist in $h_c \backslash Tr^-$. Also, the removal function does not add g-events to h_c and the relative order between events that are not removed is identical to h_c. We show $rf \in h_c.RF_c$ along the conditions of Definition 37.

$\boldsymbol{var \in WS^{vo}_{mCl(h_c)}(tr), var \in RS^{vo}_{mCl(h_c)}(tr') :}$
As $rf \in h_s.RF_c$ holds, it holds that $var \in WS_{h_c \backslash Tr^-}(tr)$ and $var \in RS_{h_c \backslash Tr^-}(tr')$. As the removal function does not add or modify (except delete) events, the same holds for h_c and $mCl(h_c)$ and the claim follows.

$\boldsymbol{co_{mCl(h_c)}(tr) :}$
The transaction tr is visible in $h \backslash Tr^-$ as else $rf \notin (h_s).RF_c$. If a transaction is visible in $h \backslash Tr^-$, it is also visible in h. This is because as we have shown above $h.RF_{val}$ is a superset of $(h \backslash Tr^-).RF_{val}$. Then as the removal function does not reorder or modify (except delete) existing events or adds events, it follows that $co_{mCl(h_c)}(tr)$.

$\boldsymbol{tr \prec_{mCl(h_c)} tr' :}$
As $rf \in h_s.RF_c$, it follows that $tr \prec_{h_s} tr'$ from which it follows that $\neg(tr' \prec_{h_c \backslash Tr^-} tr)$. Then, as the removal function does not reorder or modify

(except delete) existing events or adds events, it follows that $\neg(tr' \prec_{h_c} tr)$ and further $\neg(tr' \prec_{mCl(h_c)} tr)$. Given that this and $co_{mCl(h_c)}(tr)$ holds, and $mCl(h_c)$ is serial, it follows that $tr \prec_{mCl(h_c)} tr'$.

$\boldsymbol{\neg(\exists tr'' \in Tr : var \in WS^{vo}_{mCl(h_c)}(tr'') \wedge co_{mCl(h_c)}(tr'')}$
$\boldsymbol{\wedge tr \prec_{mCl(h_c)} tr'' \prec_{mCl(h_c)} tr') :}$
Assume a transaction fulfilling the above conditions exists for $mCl(h_c)$, then there must exist tr'' s.t. $(tr'', tr, var) \in h_c$ and $var \in WS^{vo}_{mCl(h_c)}(tr'') \wedge co_{mCl(h_c)}(tr'') \wedge tr \prec_{mCl(h_c)} tr'' \prec_{mCl(h_c)} tr'$. Assume tr'' is committed in h_c, then if $tr'' \in Tr^-$ holds, the read of tr' does not exist in $h_c \backslash Tr^-$, and if not, then it holds that $(\exists tr'' \in Tr : var \in WS^{vo}_{h_s}(tr'') \wedge co_{h_s}(tr'') \wedge tr \prec_{h_s} tr'' \prec_{h_s} tr')$ which is in contradiction to $rf \in h_s.RF_c$. Assume tr'' is commit pending in h_c. Then, it must be visible in h. In addition, since the hc-pair is consistent, it must hold that $(tr'', tr', var) \in h.RF_{val}$. Assume $tr'' \in Tr^-$ then the read of tr' on var does not exists in $h_c \backslash Tr^-$ which is a contradiction to $rf \in h_s.RF_c$. Thus, $tr'' \notin Tr^-$ and as $tr' \notin Tr^-$ and since $(h \backslash Tr^-).RF_{val} = h.RF_{val} \backslash \{rf \in h.RF_{val} \mid \exists tr \in Tr^- : tr \in rf\}$ holds, it must be that $(tr'', tr', var) \in (h \backslash Tr^-)$. Thus, tr'' would be visible in h_c, thus committed in h_s, writing to var and be real-time ordered in between tr and tr'. This is a contradiction to $rf \in h_s.RF_c$.

$\boldsymbol{rf \notin \{rf \in h.RF_c \mid \exists tr \in Tr^- : tr \in rf\} :}$
Assume the opposite then either $tr \in Tr^-$ or $tr' \in Tr^-$. In both cases the removal function would remove the read event of the reading transaction from h. So if $rf \in (h_c \backslash Tr^-).RF_c$, then it cannot be in $\{rf \in h.RF_c \mid \exists tr \in Tr^- : tr \in rf\}$.

We show the subset or equal relation in the other direction.

$\boldsymbol{h.RF_c \backslash \{rf \in h.RF_c \mid \exists tr \in Tr^- : tr \in rf\} \subseteq (h_c \backslash Tr^-).RF_c :}$

Consider an arbitrary rf-element $rf = (tr, tr', var)$ of $h_c.RF_c \backslash \{rf \in h_c.RF_c \mid \exists tr \in Tr^- : tr \in rf\}$. We show $rf \in (h_c \backslash Tr^-).RF_c$ by showing the conditions of Definition 37 are fulfilled. First, note it holds that $tr \notin Tr^-$ and $tr' \notin Tr^-$ as rf else would not exist in $h.RF_c \backslash \{rf \in h.RF_c \mid \exists tr \in Tr^- : tr \in rf\}$. Thus, no begins, writes or commit/abort invoke or responses from both transactions are removed by the removal function. Also, the read on var of tr' is not removed by the removal function as neither the first nor the second condition of the removal function hold for it. Its own transaction is not in Tr^-, and its writer in the conflict reads-from relation is tr which is also not in Tr^-. The removal function does not add g-events to h_c or change the relative order of the g-events it does not remove. Now, we show the conditions of Definition 37 hold for rf.

$\boldsymbol{var \in WS^{vo}_{mCl(h_c \backslash Tr^-)}(tr), var \in RS^{vo}_{mCl(h_c \backslash Tr^-)}(tr') :}$
As tr is not in Tr^-, the write event of tr on var is not removed by the removal function. As neither tr nor tr' are in Tr^-, the read event of tr' on var is not removed by the removal function. Both events are not modified. Thus, this holds.

$\boldsymbol{co_{mCl(h_c \backslash Tr^-)}(tr) :}$
Assume tr is committed in h_c then it is committed in $h_c \backslash Tr^-$ and its minimal completion as well. If tr is commit pending, it is visible in h. This is because $rf \in h.RF_{val}$ holds, since $rf \in h_c.RF_c$ and (h, h_c) is consistent. Then, as we have shown above, it also exists in $(h \backslash Tr^-).RF_{val}$, as tr and tr' are not in Tr^-. Thus, tr is visible in $(h \backslash Tr^-)$ and thus committed in $mCl(h_c \backslash Tr^-)$.

$\boldsymbol{tr \prec_{mCl(h_c \backslash Tr^-)} tr' :}$
As proven above, tr is committed in $mCl(h_c \backslash Tr^-)$. It also holds that $tr \prec_{mCl(h_c)} tr'$ and as $tr, tr' \notin Tr^-$ it holds that $tr \prec_{mCl(h_c \backslash Tr^-)} tr'$.

$\boldsymbol{\neg(\exists tr'' \in Tr : var \in WS^{vo}_{mCl(h_c \backslash Tr^-)}(tr'') \wedge co_{mCl(h_c \backslash Tr^-)}(tr'') \wedge}$

$\boldsymbol{tr \prec_{mCl(h_c \setminus Tr^-)} tr'' \prec_{mCl(h_c \setminus Tr^-)} tr')}$ **:**
Assume such a transaction tr'' exists in $h_c \setminus Tr^-$. It is trivially true that $tr \neq tr' \neq tr''$ and that tr'' exists in h_c and is not in Tr^-. Because $(tr, tr', var) \in h_c$ holds for this transaction tr'', it holds that

$$var \notin WS^{vo}_{mCl(h_c)}(tr'') \vee ab_{mCl(h_c)}(tr'') \vee \neg(tr \prec_{mCl(h_c)} tr'' \prec_{mCl(h_c)} tr').$$

If $var \notin WS^{vo}_{mCl(h_c)}(tr'')$ holds, then trivially $var \notin WS^{vo}_{mCl(h_c \setminus Tr^-)}(tr'')$ holds. If $ab_{mCl(h_c)}(tr'')$ holds and tr'' is aborted, then it is also aborted in $h_c \setminus Tr^-$. Now, assume $ab_{mCl(h_c)}(tr'')$ holds and tr'' is commit pending and not visible in h. Then, it is also commit pending in $h_c \setminus Tr^-$. Now, we argue why it is also not visible in $h \setminus Tr^-$, assume it would be then there would be an rf-element $(tr_1, tr_2, var) \in h.RF_{val}$ s.t. $(tr'', tr_2, var) \in (h \setminus Tr^-).RF_{val}$. If $tr_1 \in Tr^-$, then the read on var of tr_2 does not exist, so $tr_1 \notin Tr^-$ must hold. Also, tr_2 trivially is not in Tr^-, else it would not exist in $h \setminus Tr^-$. The removal function does not modify (except delete) events or reorder them. So, if tr_1 met the conditions of Definition 38 in h, it still does in $h \setminus Tr^-$. So, tr'' cannot be visible in $h \setminus Tr^-$, and it is thus still aborted in $mCl(h_c \setminus Tr^-)$. If $\neg(tr \prec_{mCl(h_c)} tr'' \prec_{mCl(h_c)} tr')$ holds, it also holds in $mCl(h_c \setminus Tr^-)$. This is because the removal function does not change the order of events and all three transactions are not in Tr^-. This means the removal function if it removed events from these transactions, only removed read events. Thus, it follows that tr'' cannot exist in $h_c \setminus Tr^-$. $\square$

Lemma 33 (Upper limit of hc-pairs)**.** *The amount of compressed hc-pairs is finite for a given T Var and Val.*

Proof. We will show for a given T and Var, that the set of all compressed hc-pairs

$$\{(h \setminus ur(hc), h_c \setminus ur(hc), IWS_{h_c}, MC_{h_c}) \mid (h, h_c) \text{ is a consistent hc-pair}\} \cup \{\mathbf{DM}\}$$

is finite. The second set of the union is obviously finite. The first set of the union is equal to the following cartesian product of the following sets:

1. $\{h \setminus ur(hc) \mid (h, h_c) \text{ is a consistent hc-pair}\}$,

2. $\{h_c \backslash ur(hc) \mid (h, h_c)$ is a consistent hc-pair$\}$,
3. $\{IWS_{h_c} \mid (h, h_c)$ is a consistent hc-pair$\}$
4. and $\{MC_{h_c} \mid (h, h_c)$ is a consistent hc-pair$\}$.

We show that the size of each of these sets has a finite upper bound only depending on *Var*, *Val* and T. As prerequisites for this result, we will show that for any hc-pair hc there exists a finite upper bound to the number of transactions that are not in $ur(hc)$ and this bound only depends on *Var* and T. Then, we show the number of pairwise different transactions is finite for a given *Var*, *Val* and T.

The number of transactions not in $ur(hc)$ is upper bounded:
We first show that there exists an upper bound to the number of transactions not in $ur(h)$ which is independent of hc and then that there exists an upper bound to the number of transactions not in $ur(h_c)$ which is independent of hc.

The number of transactions not in $ur(his)$ has a finite upper bound which is independent of hc:
According to Lemma 28 the set of these transactions contains all transactions tr s.t.

$$\neg(ab_h(tr)) \wedge \exists var \in WS_h^{vo}(tr), \forall tr' \in Tr : \neg(co_h(tr')) \vee var \notin WS_h^{vo}(tr') \vee \neg(tr \prec_h tr').$$

Assume there are more than $(|T| \cdot (2^{|Var|} + 1))$ transactions in this set. Then, there is at least one thread that has more than $2^{|Var|} + 1$ transactions meaning $2^{|Var|}$ committed transactions exist in it. Two of these committed transactions must have an identical write set and thus cannot be in the set. This is a contradiction, and thus there can be at most $(|T| \cdot (2^{|Var|} + 1))$ transactions in this set.

The number of transactions not in $ur(his_c)$ has a finite upper bound which is independent ofhc:

We denote the set of all unfinished and not commit or abort pending transactions in h_c as Tr_{unc,h_c}. According to Lemma 29 the set of these transactions can be defined as the union of two sets of transactions s.t. the first set for each transaction tr

$$\begin{gathered}\neg(ab_h(tr)) \\ \wedge \\ (\exists var \in WS^{vo}_{h_c}(tr), \forall tr' \in Tr : \neg(co_{h_c}(tr')) \vee var \notin WS^{vo}_{h_c}(tr') \wedge \neg(tr \prec_{h_c} tr')),\end{gathered}$$

holds and for the second set for each transaction tr

$$\begin{aligned}&\exists tr' \in Tr_{unc,h_c} : \neg(ab_h(tr)) \wedge tr \prec_{h_c} tr' \wedge \exists var \in WS^{vo}_{h_c}(tr), \forall tr'' \in Tr : \\ &\neg(co_{h_c}(tr'')) \vee var \notin WS^{vo}_{h_c}(tr'') \vee \neg(tr \prec_{h_c} tr'' \prec_{h_c} tr')\end{aligned}$$

holds. We show that there exists a finite upper bound to the size of both sets which is independent of hc.

We start with the second set. We further divide this set into $|Tr_{unc,h_c}|$ subsets for each transaction $tr' \in Tr_{unc,h_c}$. This set equals

$$\left\{ \begin{array}{ll} tr \mid & \neg(ab_h(tr)) \wedge tr \prec_{h_c} tr' \wedge \exists var \in WS^{vo}_{h_c}(tr), \forall tr'' \in Tr : \\ & \neg(co_{h_c}(tr'')) \vee var \notin WS^{vo}_{h_c}(tr'') \vee \neg(tr \prec_{h_c} tr'' \prec_{h_c} tr') \end{array} \right\}.$$

We denote this set $Tr_{rd,tr'}$ and call it the set of transactions readable by tr'. Assume this set contains more than $2^{|Var|}$ transactions, then two transactions must share a write set as there are only $2^{|Var|}$ possible write sets. Let these transactions be tr_1 and tr_2. Then for transaction tr_1 the formula

$$\begin{aligned}&\neg(ab_h(tr_1)) \wedge tr_1 \prec_{h_c} tr' \wedge \exists var \in WS^{vo}_{h_c}(tr_1), \forall tr'' \in Tr : \wedge \\ &\neg(co_{h_c}(tr'')) \vee var \notin WS^{vo}_{h_c}(tr'') \vee \neg(tr_1 \prec_{h_c} tr'' \prec_{h_c} tr'),\end{aligned}$$

does not hold true as can be easily seen by considering tr_2 as an interpretation for tr''. Thus, the upper bound for the set is $|Tr_{unc,h_c}| \cdot 2^{|Var|}$ which is upper bounded by $|T| \cdot 2^{|Var|}$.

For the second set

$$\left\{ \begin{array}{ll} tr \mid & \neg(ab_h(tr)) \wedge \exists var \in WS_{h_c}^{vo}(tr), \forall tr' \in Tr : \\ & \neg(co_{h_c}(tr')) \vee var \notin WS_{h_c}^{vo}(tr') \wedge \neg(tr \prec_{h_c} tr'), \end{array} \right\}$$

we can reuse part of the above result. Assume an unfinished transaction tr' at the end of h, it is easy to see that $Tr_{rd,tr'}$ is equal to the above set. As discussed above, this set is bounded by $2^{|Var|}$.

Thus, overall the amount of transactions that are not unreadable in h_c is bounded by $(|T| + 1) \cdot 2^{|Var|}$ which is independent of hc.

Using these results, we can deduct the intersection of both sets with the union of a set containing at most 1 transaction has a finite upper bound independent of hc as well.

The number of pairwise different transactions for a single transaction in a consistenthc-pair hc has a finite upper bound independent of hc : Here, as in the whole section, we consider each event but commit/abort invoke and commit/abort, as atomic. We show the claim by showing a) that transactions have a finite maximum length and b) that for each event there is a finite amount of possibilities.

A transaction begins, writes to each variable and reads from each variable and then invokes a commit or abort and then commits or aborts. As hc is not in the **DM** equivalence class, we can assume that no transaction has multiple reads reading different values for the same variable. We also assume each transaction does not read the same value twice from the same variable. Thus, a transaction at most reads once from each variable. Also, transaction are assumed to not write twice from the same variable. Thus, each transaction contains at most $2 \cdot \mathit{Var}$ read and write events. Thus, the length of a transaction (considering each event but commit/abort invoke and commit/abort, as atomic) is upper bounded by $3 + |2 \cdot \mathit{Var}|$ and finite. Each event of a transaction is either a read, write, begin, commit invoke, abort invoke or commit response or abort response. Each read or write g-event has $\mathit{Var} \cdot \mathit{Val}$ possible variable value combinations. So, overall each g-event of a transaction can be one of $(2\,\mathit{Var} \cdot \mathit{Val}) + 5$ possible g-events.

So, there is a finite upper bound to the length of a transaction which is independent of hc, and each g-event in a transaction is out of a finite set of possible events whose contents depend on $\mathit{Var}\ \mathit{Val}$ and T. So, there is an upper bound to the number of pairwise different transactions which only depends on T, Var and Val.

Finally, we show that the size of each of these sets has a finite upper bound only depending on Var, Val and T.

$\boldsymbol{\{h \backslash ur(hc) \mid (h, h_c)}$ **is a consistent hc-pair** $\boldsymbol{\}}$ **:**
There is an upper bound to the number of transactions in any g-history in this set which only depends on Var and T. There is an upper bound for the length each of these g-histories only depending on T and Var. There is an upper bound to number of pairwise different events occuring in the g-histories which only depends on T, Val and Var. Thus, there is an upper bound to the number of pairwise g-histories in this set for a given T, Val and Var.

$\boldsymbol{\{h_c \backslash ur(hc) \mid (h, h_c)}$ **is a consistent hc-pair** $\boldsymbol{\}}$ **:**
This is analogue to the previous part of the proof.

$\boldsymbol{\{IWS_{h_c} \mid (h, h_c)}$ **is a consistent hc-pair** $\boldsymbol{\}}$ **:**
This IWS_{h_c} can at most have T elements which each can contain at most $2^{|\mathit{Var}|}$

elements.

$\boldsymbol{\{MC_{h_c} \mid (h, h_c)}$ **is a consistent hc-pair** $\boldsymbol{\}}$ **:**
There at most T elements in this set, which each are taken out of T possible transactions.

The overall claim follows. $\square$

Lemma 34 (Compression represents an equivalence class)**.** *Given two arbitrary hc-pairs* (h, h_c) *and* (h', h'_c)*, it holds that*

$$cmp(h, h_c) = cmp(h', h'_c) \rightarrow (h, h_c) \equiv_{ext} (h', h'_c).$$

Proof. It is to prove that if $cmp(h, h_c) = cmp(h', h'_c)$ holds, then it holds for any arbitrary g-event sequence seq that

$$\exists h_{c,ins} \in ins(h_c, seq) : h \cdot seq \equiv h_{c,ins} \leftrightarrow \exists h'_{c,ins} \in ins(h'_c, seq) : h' \cdot seq \equiv h'_{c,ins}.$$

If $cmp(h, h_c) = cmp(h', h'_c) = \mathbf{DM}$, then in the extension of both hc-pairs by any sequence the respective g-history is not equivalent to its respective candidate as they both contain mutually exclusive rf-elements between their g-history and candidate, which are all fixed in the g-history. As we argued in the proof of Lemma 30, any extension of such hc-pairs is non-consistent.

We will now prove the lemma for $cmp(h, h_c) = cmp(h', h'_c) \neq \mathbf{DM}$. We will show this by showing two statements and then proving why these imply the lemma.

1. Given an arbitrary hc-pair $hc = (h, h_c)$, it holds that

$$hc \text{ is consistent } \leftrightarrow cmp(hc) \neq \mathbf{DM}. \tag{C.2}$$

2. For any two hc-pairs $hc = (h, h_c), hc' = (h', h'_c)$ where $cmp(hc) = cmp(hc')$ and an arbitrary g-event ev it holds that

$$\begin{aligned}&\forall h_{c,ins} \in ins(h_c, ev) \exists h'_{c,ins} \in ins(h'_c, ev) : \\ &cmp(h \cdot ev, h_{c,ins}) = cmp(h' \cdot ev, h'_{c,ins}).\end{aligned} \tag{C.3}$$

Event	Proof in
Begin	Proposition 14
Read	Proposition 16
Write	Proposition 17
Commit Invoke	Proposition 18
Commit	Proposition 19
Abort	Proposition 22

Table C.1: Proofs of Equation (C.3) for each g-event

To declutter the overall proof, these statements are proven in separate propositions. Equation (C.2) is proven in Lemma 38. For Equation (C.3) we employ a case distinction over the type of g-event. We have proven these in separate propositions, an overview where the proof for each event is located can be found in Table C.1. We show that Equation (C.3) implies the following: For any two hc-pairs $hc = (h, h_c)$, $hc' = (h', h'_c)$, where $cmp(hc) = cmp(hc')$, and an arbitrary sequence of g-events seq it holds that

$$\begin{aligned}&\forall h_{c,ins} \in ins(h_c, seq) \exists h'_{c,ins} \in ins(h'_c, seq) :\\ &cmp(h \cdot seq, h_{c,ins}) = cmp(h' \cdot seq, h'_{c,ins}).\end{aligned}$$

We show this via induction over the sequence seq.

Induction start seq $=$ *ev*:
This directly follows from Equation (C.3).

Induction step seq $\rightarrow$ seq $\cdot$ *ev* :
There exists $h_{c1,ins} \in ins(h_c, seq)$, for which by induction statement then there also exists $h'_{c1,ins} \in ins(h'_c, seq)$ s.t. $cmp(h \cdot seq, h_{c1,ins}) = cmp(h \cdot seq, h'_{c1,ins})$. It follows by Equation (C.3) that

$$\begin{aligned}&\forall h_{c,ins} \in ins(h_{c1,ins}, ev) \exists h'_{c,ins} \in ins(h'_{c1,ins}, ev) :\\ &cmp(h \cdot seq \cdot ev, h_{c,ins}) = cmp(h' \cdot seq \cdot ev, h'_{c,ins}).\end{aligned}$$

This concludes the proof by induction.

From this result it trivially follows that for any two hc-pairs $hc = (h, h_c)$, $hc' = (h', h'_c)$ s.t. $cmp(h, h_c) = cmp(h', h'_c)$ it holds that

$$\begin{aligned}&\exists h_{c,ins} \in ins(h_c, seq) : h \cdot seq \equiv h_{c,ins}\\&\rightarrow \exists h'_{c,ins} \in ins(h'_c, seq) : cmp(h \cdot seq, h_{c,ins}) = cmp(h' \cdot seq, h'_{c,ins}).\end{aligned}$$

We show that the left-hand side of this statement then also implies $h' \cdot seq \equiv h'_{c,ins}$. If $(h \cdot seq, h_{c,ins})$ is consistent, then from Equation (C.2) it follows that $cmp(h \cdot seq, h_{c,ins})$ is consistent as well. Thus, $cmp(h' \cdot seq, h'_{c,ins})$ is also consistent as the compressions are equal. This implies that $(h' \cdot seq, h'_{c,ins})$ is consistent, and thus $h' \cdot seq \equiv h'_{c,ins}$. From this the left to right direction of the overall claim follows.

$$\exists h_{c,ins} \in ins(h_c, seq) : h \cdot seq \equiv h_{c,ins} \rightarrow \exists h'_{c,ins} \in ins(h'_c, seq) : h' \cdot seq \equiv h'_{c,ins}.$$

We can show the reverse direction of this equation analogue. Combining both shows the lemma to be true which concludes the proof.

□

Lemma 38 (HC-Pair consistent iff compression not in **DM**). *Given an arbitrary hc-pair hc, it holds that*

$$hc \textit{ is consistent } \leftrightarrow cmp(hc) \neq \mathbf{DM}.$$

Proof. We show both directions separately.

hc is consistent $\rightarrow$ *cmp*(hc) $\neq$ DM :

If hc is consistent, then by definition of consistency for hc-pairs the minimal completion of its candidate is legal. By Lemma 24, it then holds that a legal serial completion for the candidate exists. By Lemma 23, this implies that the value reads-from relation of the g-history of hc and the conflict reads-from relation of the candidate are identical. Then, by the definition of cmp (Definition 50), the compression is not equal to $\mathbf{DM}$.

$\boldsymbol{cmp}(\mathbf{hc}) \neq \mathbf{DM} \rightarrow$ **hc is consistent :**
We show this by contraposition. Thus, we show that

$$\textbf{hc is not consistent} \rightarrow cmp(\mathbf{hc}) = \mathbf{DM}.$$

If hc is not consistent, then by definition of consistency for hc-pairs the minimal completion of its candidate is not legal. By Lemma 23, this implies that the value reads-from relation of the g-history of hc and the conflict reads-from relation of the candidate are not identical. Thus, by the definition of the compression function (Definition 50), then it holds that $cmp(hc) = \mathbf{DM}$. □

Proposition 14. *Given a g-event* $ev = \mathbf{B}_t^{tr}$*, two arbitrary hc-pairs* $hc = (h, h_c)$ *and* $hc' = (h', h_c')$ *s.t.* $cmp(h, h_c) = cmp(h', h_c')$*, it holds that*

$$\forall h_{c,ins} \in ins(h_c, ev) \exists h'_{c,ins} \in ins(h'_c, ev) : \\ cmp(h \cdot ev, h_{c,ins}) = cmp(h' \cdot ev, h'_{c,ins}).$$

Proof. Choose an arbitrary $h_{c,ins} \in ins(h_c, ev)$. Let $n \in \mathbb{N}$ be the natural number s.t. $h_{c,ins} = st(h_c) \cdot add(en(h_c), ev, n)$ holds. We will first show that $st(h_c') \cdot add(en(h_c), ev, n) \in ins(h_c', ev)$ holds because $en(h_c) = en(h_c')$. Note that all g-events in $en(h_c)$ are of unfinished transactions. Consider each transaction tr' for which in $en(h_c)$ there exists a read of a transaction tr'' and there exists a variable var' s.t. $(tr', tr'', var') \in h_c.RF_c$. If tr' is unfinished, it is not in $ur(hc)$. If it is finished, then by Definition 37 it holds that

$$tr' \prec_{h_c} tr'' \wedge \\ \neg(\forall var \in WS_{h_c}^{vo}(tr'), \exists tr''' \in Tr : co_{h_c}(tr''') \wedge var \in WS_{h_c}^{vo}(tr''') \wedge tr' \prec_{h_c} tr''' \prec_{h_c} tr'').$$

Given Lemma 29, this implies that tr' is not in $ur(hc)$. Thus, $en(h_c)\backslash ur(hc) = en(h_c)$ and as $h_c\backslash ur(hc) = h'_c\backslash ur(hc')$ holds, it follows that $en(h_c) = en(h'_c)$. Thus, we can set $h'_{c,ins} = st(h'_c) \cdot add(en(h_c), ev, n)$ as it is in $ins(h'_c, ev)$. It also holds that $ur(hc) = ur(h \cdot ev, h_{c,ins})$ and $ur(hc') = ur(h' \cdot ev, h'_{c,ins})$. We show this in Proposition 15. We show $cmp(h \cdot ev, h_{c,ins}) = cmp(h' \cdot ev, h'_{c,ins})$ by showing the elements of both tuples match.

$\boldsymbol{(h \cdot ev)\backslash ur(h \cdot ev, h_{c,ins}) = (h' \cdot ev)\backslash ur(h' \cdot ev, h'_{c,ins}) :}$
This is equal to

$$(h \cdot ev)\backslash ur(hc) = (h' \cdot ev)\backslash ur(hc').$$

As tr is not in $ur(hc)$ or $ur(hc')$, it holds that

$$(h\backslash ur(hc)) \cdot ev = (h'\backslash ur(hc)) \cdot ev.$$

As $h\backslash ur(hc) = h'\backslash ur(hc')$, the claim follows.

$\boldsymbol{h_{c,ins}\backslash ur(h \cdot ev, h_{c,ins}) = h'_{c,ins}\backslash ur(h' \cdot ev, h'_{c,ins}) :}$
This is equal to

$$(h_{c,ins})\backslash ur(hc) = (h'_{c,ins})\backslash ur(hc').$$

This in turn is equal to

$$(st(h_c) \cdot add(en(h_c), ev, n))\backslash ur(hc) = (st(h'_c) \cdot add(en(h_c), ev, n))\backslash ur(hc').$$

As we have discussed above, no events in $add(en(h_c), ev, n)$ can be of unreadable transactions nor are reads from unreadable transactions in it. Thus, it follows that

$$(st(h_c)\backslash ur(hc) \cdot add(en(h_c), ev, n)) = (st(h'_c)\backslash ur(hc') \cdot add(en(h_c), ev, n)).$$

From $cmp(h, h_c) = cmp(h', h'_c)$ it then follows that this equivalent to

$$(st(h_c)\backslash ur(hc) \cdot add(en(h_c), ev, n)) = (st(h_c)\backslash ur(hc) \cdot add(en(h_c), ev, n)).$$

This is trivially true.

$\boldsymbol{IWS_{h_{c,ins}} = IWS_{h'_{c,ins}}}$ **:**
For any transaction tr' that is not tr, it holds that

$$IWS_{h_c}(tr') = IWS_{h_{c,ins}}(tr')$$

and

$$IWS_{h'_c}(tr') = IWS_{h'_{c,ins}}(tr').$$

This is because the order of all events excluding ev is identical in h_c and $h_{c,ins}$. Thus, it holds that $h_c.RF_c = h_c.RF_{c,ins}$ as the begin does not alter the reads-from relation. The same holds for h'_c and $h'_{c,ins}$. Thus, it holds that $IWS_{h'_{c,ins}}(tr') = IWS_{h_{c,ins}}(tr')$. As we discussed above, each transaction in $en(h_c)$ and $en(h'_c)$ is not in $ur(hc)$ and $ur(hc')$, respectively, and each transaction being read by an event in these sequences is also not in $ur(hc)$ and $ur(hc')$ respectively. Also, note that we showed that $add(en(h_c), ev, n) = add(en(h'_c), ev, n)$ Thus, tr' is interrupting for the same rf-elements in both.

$$IWS_{h_{c,ins}}(tr) = IWS_{h'_{c,ins}}(tr).$$

$\boldsymbol{MC_{h_{c,ins}} = MC_{h'_{c,ins}}}$ **:**
For any transaction tr' that is not tr, it holds that $mc_{h_c}(tr') = mc_{h_{c,ins}}(tr')$. This is because except ev the events and their order of h_c and $h_{c,ins}$ are identical. The same holds for h'_c and $h'_{c,ins}$. As tr consists only of a begin, $mc_{h_{c,ins}}(tr)$ is false, analogue for $h'_{c,ins}$. Thus, as $MC_{h_c} = MC_{h'_c}$, it follows that $MC_{h_{c,ins}} =$

$MC_{h'_{c,ins}}$.

$\square$

Proposition 15. *Given a g-event* $ev = \mathbf{B}_t^{tr}$ *and one arbitrary hc-pair* $hc = (h, h_c)$, *it holds that*

$$\forall h_{c,ins} \in ins(h_c, ev) : ur(hc) = ur(h \cdot ev, h_{c,ins}).$$

Proof. It holds that $ur(hc) = (ur(h) \cap ur(h_c)) \setminus tr_{lf}(h_c)$. Trivially the last finished transaction is identical in h_c and $h_{c,ins}$. Also, $ur(h) = ur(h \cdot ev)$ as one can easily verify by looking at Lemma 26 as the appended begin event does not abort a transaction or add or remove a transaction with a non-empty write set. We show that the conditions of Lemma 29 hold for h_c iff they hold for $h_{c,ins}$. It is trivial to see that tr is not unreadable as it is unfinished. Any other transaction occurs in h_c iff it occurs in $h_{c,ins}$. We exclude any transactions that have an empty write set as they are unreadable by the conditions of this lemma and their write set is also empty after inserting ev. Reusing the notation from the proof of this lemma, it is left to show that for all transactions tr_x in h_c

$$\begin{aligned}
&ab_{h_c}(tr_x) \vee (co_{h_c}(tr_x) \wedge obe(h_c, tr_x) \wedge \\
&(\forall tr' \in Tr_{unc,h_c} : \neg(tr_x \prec_{h_c} tr') \vee obt(h_c, tr_x, tr')) \\
&\qquad\qquad\qquad\qquad \leftrightarrow \\
&ab_{h_{c,ins}}(tr_x) \vee (co_{h_{c,ins}}(tr_x) \wedge obe(h_{c,ins}, tr_x) \\
&\wedge(\forall tr' \in Tr_{unc,h_c} : \neg(tr_x \prec_{h_{c,ins}} tr') \vee (obt(h_{c,ins}, tr_x, tr') \wedge \\
&(\neg(tr_x \prec_{h_{c,ins}} tr) \vee obt(h_{c,ins}, tr_x, tr))).
\end{aligned}$$

We show this by first showing the following five separate statements:

1. $ab_{h_c}(tr_x) \leftrightarrow ab_{h_{c,ins}}(tr_x)$,
2. $co_{h_c}(tr_x) \leftrightarrow co_{h_{c,ins}}(tr_x)$,
3. $obe(h_c, tr_x) \leftrightarrow obe(h_{c,ins}, tr_x)$,
4. $obe(h_c, tr_x) \leftrightarrow \neg(tr_x \prec_{h_{c,ins}} tr) \vee obt(h_{c,ins}, tr_x, tr)$

5. and $\forall tr' \in Tr_{unc,h_c} : \neg(tr_x \prec_{h_c} tr') \vee obt(h_c, tr_x, tr') \leftrightarrow \forall tr' \in Tr_{unc,h_c} : \neg(tr_x \prec_{h_{c,ins}} tr') \vee obt(h_{c,ins}, tr_x, tr')$.

All of these statements combined show the overall claim.

$\boldsymbol{ab_{h_c}(tr_x) \leftrightarrow ab_{h_{c,\mathrm{ins}}}(tr_x)}$ **:** This is true as inserting a begin of *tr* does not alter tr_x.

$\boldsymbol{co_{h_c}(tr_x) \leftrightarrow co_{h_{c,\mathrm{ins}}}(tr_x)}$ **:** This is true as inserting a begin of *tr* does not alter tr_x.

$\boldsymbol{obe(h_c, tr_x) \leftrightarrow obe(h_{c,\mathrm{ins}}, tr_x)}$ **:**
Consider $obe(h_{c,ins}, tr_x)$:

$$\forall var \in WS^{vo}_{h_{c,ins}}(tr_x), \exists tr' \in Tr : co_{h_{c,ins}}(tr') \wedge var \in WS^{vo}_{h_{c,ins}}(tr') \wedge tr_x \prec_{h_{c,ins}} tr'.$$

It holds that *tr* has an empty write set; thus, if the quantified tr' is substituted by *tr* in this formula, it is false. Trivially for any transaction that occurs both in $h_{c,ins}$ and h_c has the same write set in both. It is also either committed in both or neither and is real-time ordered identically in both to any other transaction occurring in h_c. Thus, this is equivalent to

$$obe(h_c, tr_x) = \forall var \in WS^{vo}_{h_c}(tr_x), \exists tr' \in Tr : co_{h_c}(tr') \wedge var \in WS^{vo}_{h_c}(tr') \wedge tr_x \prec_{h_c} tr'.$$

$\boldsymbol{obe(h_c, tr_x) \leftrightarrow \neg(tr_x \prec_{h_{c,ins}} tr) \vee obt(h_{c,ins}, tr_x, tr)}$ **:**

We show that

$$\forall var \in WS^{vo}_{h_c}(tr_x), \exists tr' \in Tr : co_{h_c}(tr') \wedge var \in WS^{vo}_{h_c}(tr') \wedge tr_x \prec_{h_c} tr'$$
$$\leftrightarrow$$
$$\forall var \in WS^{vo}_{h_{c,ins}}(tr_x), \exists tr' \in Tr : co_{h_{c,ins}}(tr') \wedge var \in WS^{vo}_{h_{c,ins}}(tr') \wedge tr_x \prec_{h_{c,ins}} tr' \prec_{h_{c,ins}} tr.$$

Note that transactions except tr are identical in h_c and h_c and their real-time order when excluding tr is also equal in h_c and $h_{c,ins}$. It also holds that tr is inserted after every finished transaction in h_c. In the right-hand side of the formula the exist quantifier cannot be true for the substitution $tr' = tr$ as tr is not real-time ordered after itself. Thus, the equivalence holds true.

$\boldsymbol{\forall tr' \in Tr_{unc,h_c} : \neg(tr_x \prec_{h_c} tr') \vee obt(h_c, tr_x, tr')}$
$\boldsymbol{\leftrightarrow \forall tr' \in Tr_{unc,h_c} : \neg(tr_x \prec_{h_{c,ins}} tr') \vee obt(h_{c,ins}, tr_x, tr') :}$
Trivially,

$$\forall tr' \in Tr_{unc,h_c} : \neg(tr_x \prec_{h_c} tr') \leftrightarrow \forall tr' \in Tr_{unc,h_c} : \neg(tr_x \prec_{h_{c,ins}} tr')$$

holds. Now, we show

$$\forall tr' \in Tr_{unc,h_c} : \forall var \in WS^{vo}_{h_c}(tr_x), \exists tr'' \in Tr : co_{h_c}(tr'') \wedge$$
$$var \in WS^{vo}_{h_c}(tr'') \wedge tr_x \prec_{h_c} tr'' \prec_{h_c} tr'$$
$$\leftrightarrow$$
$$\forall tr' \in Tr_{unc,h_c} : \forall var \in WS^{vo}_{h_{c,ins}}(tr_x), \exists tr'' \in Tr : co_{h_{c,ins}}(tr'') \wedge$$
$$var \in WS^{vo}_{h_{c,ins}}(tr'') \wedge tr_x \prec_{h_{c,ins}} tr'' \prec_{h_{c,ins}} tr'.$$

As tr is not committed, on neither side of the equivalence the formula after the exist quantifier can be true when substituting tr for tr''. Any other transaction is identical in h_c and $h_{c,ins}$ and the event order when excluding tr is also equal in h_c and $h_{c,ins}$. Thus, both sides are equivalent.

Thus, the overall equivalency holds.

□

Proposition 16. *Given a g-event* $ev = \mathbf{R}_t^{tr}(var, val)$ *and two arbitrary hc-pairs* (h, h_c) *and* (h', h'_c) *s.t.* $cmp(h, h_c) = cmp(h', h'_c)$, *it holds that*

$$\forall h_{c,ins} \in ins(h_c, ev) \exists h'_{c,ins} \in ins(h'_c, ev) :$$
$$cmp(h \cdot ev, h_{c,ins}) = cmp(h' \cdot ev, h'_{c,ins}).$$

Proof. Note that tr is not visible in in $h \cdot ev$ and $h' \cdot ev$. Let $h_c = ev_0 \dots ev_{h_c}^{tr,ls} \dots ev_n$ and $h'_c = ev'_0 \dots ev_{h'_c}^{tr,ls} \dots ev'_l$. It holds that $ins(h_c, ev)$ contains exactly the element $ev_0 \dots ev_{h_c}^{tr,ls} ev \dots ev_n$ and $ins(h'_c, ev)$ contains exactly the element $ev'_0 \dots ev_{h'_c}^{tr,ls} ev \dots ev'_l$. Thus, we set $h_{c,ins}$ to the former and $h'_{c,ins}$ to the latter candidate. There must exist one rf-element (tr', tr, var) in $(h \cdot ev).RF_{val}$ and one new rf-element (tr'', tr, var) in $h_{c,ins}.RF_c$. If $tr' \neq tr''$, then $cmp(h \cdot ev, h_{c,ins}) = cmp(h' \cdot ev, h'_{c,ins}) = \mathbf{DM}$ holds. Thus, in the further we assume $tr'' = tr'$ and use the identifier tr'. It holds that $ur(hc) = ur(h \cdot ev, h_{c,ins})$ as the pairs h and $h \cdot ev$, and h_c and $h_{c,ins}$, each contain the same events in the same order, except that for tr (a non-committed) transaction a read event was added. It is easy to see that this does not affect the conditions of Lemmas 26 and 29. Analogue, $ur(hc') = ur(h' \cdot ev, h'_{c,ins})$ holds.

$\boldsymbol{(h \cdot ev) \backslash ur(h \cdot ev, h_{c,ins}) = (h' \cdot ev) \backslash ur(h' \cdot ev, h'_{c,ins})}$ **:**
Given that $ur(hc) = ur(h \cdot ev, h_{c,ins})$ and $ur(hc') = ur(h' \cdot ev, h'_{c,ins})$ holds, this is equivalent to

$$(h \cdot ev) \backslash ur(hc) = (h' \cdot ev) \backslash ur(hc').$$

It holds that the unfinished transaction tr is not part of $ur(hc)$ and the transaction that ev is reading from in $h \cdot ev$ and $h_{c,ins}$ trivially is not unreadable in h and thus not in $ur(hc)$. The same holds for hc' and $h' \cdot ev$ and $h'_{c,ins}$. Thus, the above statement is equivalent to

$$(h \backslash ur(hc)) \cdot ev = (h' \backslash ur(hc')) \cdot ev.$$

Then, given that $cmp(hc) = cmp(hc')$ this is equivalent to

$$(h \backslash ur(hc)) \cdot ev = (h \backslash ur(hc)) \cdot ev,$$

which is a trivially true statement.

$\boldsymbol{h_{\mathbf{c,ins}} \backslash ur(h \cdot ev, h_{c,ins}) = h'_{\mathbf{c,ins}} \backslash ur(h' \cdot ev, h'_{c,ins})}$ **:**
Given that $ur(hc) = ur(h \cdot ev, h_{c,ins})$ and $ur(hc') = ur(h' \cdot ev, h'_{c,ins})$ holds, this is equivalent to

$$h_{c,ins} \backslash ur(hc) = h'_{c,ins} \backslash ur(hc').$$

This is equivalent to

$$(ev_0 \dots ev_{h_c}^{tr,ls} ev \dots ev_n) \backslash ur(hc) = (ev'_0 \dots ev_{h'_c}^{tr,ls} ev \dots ev'_l) \backslash ur(hc').$$

Note that $h_{c,ins}.RF_c = h_c.RF_c \cup \{(tr', tr, var)\}$ and $h'_{c,ins}.RF_c = h'_c.RF_c \cup \{(tr', tr, var)\}$ holds. Since the removal function is applied to each element and individually and checks whether it belongs to a transaction in the input set or it is a read from a transaction in the input set, its results for each event when applied above are the same as when applied to h_c and h'_c. Also, it does not alter ev for both sides of the equation as it is a read of a transaction not in $ur(hc)$ or $ur(hc')$ and also as discussed above does not read from an unreadable transaction. Let ev_i^- be the event at index i of h_c after applying $\backslash ur(hc)$ to it. Let x be the index of $ev_{h_c}^{tr,ls}$ in h_c. Let $seq_{1,h_c \backslash ur(hc)}$ denote $ev_0^- \dots ev_x^-$ and $seq_{2,h_c \backslash ur(hc)}$ the remaining part of $h_c \backslash ur(hc)$. Define the same analogue for h'_c. Then, the previous equation is equivalent to

$$seq_{1,h_c \backslash ur(hc)} \cdot ev \cdot seq_{2,h_c \backslash ur(hc)} = seq_{1,h'_c \backslash ur(hc)} \cdot ev \cdot seq_{2,h'_c \backslash ur(hc)}.$$

This is true as $cmp(hc) = cmp(hc')$.

$\boldsymbol{IWS_{h_{c,ins}} = IWS_{h'_{c,ins}}}$ **:**

There is no new transaction in both insertions. In comparison to $h_c.RF_c$, in $h_{c,ins}.RF_c$ there is one new rf-element $(tr', tr, var) = rf$ in the latter and no rf-elements are removed. Then, for any transactions tr_i that interrupt rf it holds that $IWS_{h_{c,ins}}(tr_i) = IWS_{h_{c,ins}}(tr_i) \cup \{var\}$, for any other the interrupting write set is unchanged. The same holds for $h'_{c,ins}$. We show that if a transaction tr_i is interrupting for rf in one candidate, it is also in the other. We only show this from $h_{c,ins}$ to $h'_{c,ins}$. The other direction is completely analogue. Assume an arbitrary transaction tr_i s.t. it is interrupting for rf. Thus, it holds that $tr' \prec_{mCl(h_{c,ins})} tr_i \prec_{mCl(h_{c,ins})} tr$ and tr_i is unfinished and either not commit pending or commit pending and has $var \in WS^{vo}_{h_{c,ins}}(tr_i)$. Given that the insertion function only inserts one event, the above statements are also true for h_c. It holds then $tr \notin ur(hc)$ as it is unfinished in hc, $tr_i \notin ur(hc)$ since it is unfinished and $tr' \notin ur(hc)$ since its write on var is not overwritten before tr as else rf would not exist in $h_{c,ins}$ and thus also not before tr_i. Since $cmp(hc) = cmp(hc')$ holds, the same holds for h'_c. Thus, inserting ev at the end tr leads to rf existing in $h'_{c,ins}.RF_c$ and tr_i being interrupting for it. Thus, for any transaction tr_i that interrupts rf in $h_{c,ins}$ it holds that $IWS_{h'_{c,ins}}(tr_i) = IWS_{h'_{c,ins}}(tr_i) \cup \{var\}$. for any other the interrupting write set is unchanged.

$\boldsymbol{MC_{h_{c,ins}} = MC_{h'_{c,ins}}}$ **:**
It holds that $h_{c,ins}.RF_c = h_c.RF_c \cup \{(tr', tr, var)\}$. All rf-elements except rf are abortable in h_c iff they are abortable in $h_{c,ins}$. This holds because all the rf-elements of h_c also exists in $h_{c,ins}$ and all events except ev are identical in h_c and $h_{c,ins}$. By an analogue argument, the same holds for the h'_c and $h'_{c,ins}$. Thus, $MC_{h_{c,ins}} \backslash \{tr'\} = MC_{h'_{c,ins}} \backslash \{tr'\}$ holds.

We show that $tr' \in MC_{h_{c,ins}} \leftrightarrow tr' \in MC_{h'_{c,ins}}$ via case distinction over whether rf is abortable or not. If rf is abortable in $h_{c,ins}$, tr' is commit pending in h_c and $h_{c,ins}$. Then, it holds that $tr' \in MC_{h_{c,ins}}$. Given that $(tr', tr, var) \in h_{c,ins}.RF_c$, tr' is trivially not unreadable in h_c, meaning $tr' \notin ur(hc)$. Given that $cmp(hc) = cmp(hc')$, it holds that tr' is also commit pending in h'_c. As

$(tr', tr, var) \in h'_{c,ins}.RF_c$ holds, then it holds that $tr' \in MC_{h'_{c,ins}}$.

If rf is not abortable in $h_{c,ins}$, then trivially $tr' \notin MC_{h_{c,ins}}$ and tr' is committed in h_c and $h_{c,ins}$. Given that $cmp(hc) = cmp(hc')$ and $tr' \notin ur(hc)$, it holds that tr' is also committed in h'_c and $h'_{c,ins}$. Then, by an analogue argument $tr' \notin MC_{h'_{c,ins}}$ holds.

The overall claim follows because $MC_{h_{c,ins}} \backslash \{tr'\} = MC_{h'_{c,ins}} \backslash \{tr'\}$ and $tr' \in MC_{h_{c,ins}}$ iff $tr' \in MC_{h'_{c,ins}}$.

□

Proposition 17. *Given a g-event* $ev = \mathbf{W}_t^{tr}(var, val)$, *two arbitrary hc-pairs* (h, h_c) *and* (h', h'_c) *s.t.* $cmp(h, h_c) = cmp(h', h'_c)$, *it holds that*

$$\forall h_{c,ins} \in ins(h_c, ev) \exists h'_{c,ins} \in ins(h'_c, ev) :$$
$$cmp(h \cdot ev, h_{c,ins}) = cmp(h' \cdot ev, h'_{c,ins}).$$

Proof. Let $h_c = ev_0 \dots ev_{h_c}^{tr,ls} \dots ev_n$ and $h'_c = ev'_0 \dots ev_{h'_c}^{tr,ls} \dots ev'_l$. It holds that $ins(h_c, ev)$ contains exactly the element $ev_0 \dots ev_{h_c}^{tr,ls} ev \dots ev_n$ and $ins(h'_c, ev)$ contains exactly the element $ev'_0 \dots ev_{h'_c}^{tr,ls} ev \dots ev'_l$. Thus, we set $h_{c,ins}$ to the former and $h'_{c,ins}$ to the latter candidate. It holds that $ur(hc) = ur(h \cdot ev, h_{c,ins})$ as both contain the same transactions in the same order, except that for tr (a non-committed and non-commit pending) transaction a write event was added. It is easy to see that this does not affect the conditions of Lemmas 26 and 29. Analogue, $ur(hc') = ur(h' \cdot ev, h'_{c,ins})$ holds.

$\boldsymbol{(h \cdot ev) \backslash ur(h \cdot ev, h_{c,ins}) = (h' \cdot ev) \backslash ur(h' \cdot ev, h'_{c,ins}) :}$
Given that $ur(hc) = ur(h \cdot ev, h_{c,ins})$ and $ur(hc') = ur(h' \cdot ev, h'_{c,ins})$ holds, this is equivalent to

$$(h \cdot ev) \backslash ur(hc) = (h' \cdot ev) \backslash ur(hc').$$

It holds that the unfinished transaction tr is not part of $ur(hc)$ and $ur(hc')$, and ev is not a read event; thus, this is equivalent to

$$(h\backslash ur(hc)) \cdot ev = (h'\backslash ur(hc')) \cdot ev.$$

Then, given that $cmp(hc) = cmp(hc')$ this is equivalent to

$$(h\backslash ur(hc)) \cdot ev = (h\backslash ur(hc)) \cdot ev,$$

which is a trivially true statement.

$\boldsymbol{h_{c,ins}\backslash ur(h \cdot ev, h_{c,ins}) = h'_{c,ins}\backslash ur(h' \cdot ev, h'_{c,ins}):}$
Given that $ur(hc) = ur(h \cdot ev, h_{c,ins})$ and $ur(hc') = ur(h' \cdot ev, h'_{c,ins})$ holds, this is equivalent to

$$h_{c,ins}\backslash ur(hc) = h'_{c,ins}\backslash ur(hc').$$

This is equivalent to

$$(ev_0 \dots ev^{tr,ls}_{h_c} ev \dots ev_n)\backslash ur(hc) = (ev'_0 \dots ev^{tr,ls}_{h'_c} ev \dots ev'_l)\backslash ur(hc').$$

Note that $h_{c,ins}.RF_c = h_c.RF_c$ and $h'_{c,ins}.RF_c = h'_c.RF_c$. Since the removal function is applied to each element and individually and checks whether it belongs to a transaction in the input set or it is a read from a transaction in the input set, its results for each event when applied above are the same as when applied to h_c and h'_c. Also, it does not alter ev for both sides of the equation as it is a write of a transaction not in $ur(hc)$ or $ur(hc')$. Let ev^-_i be the event at index i of h_c after applying $\backslash ur(hc)$ to it. Let x be the index of $ev^{tr,ls}_{h_c}$ in h_c. Let $seq_{1,h_c\backslash ur(hc)}$ denote $ev^-_0 \dots ev^-_x$ and $seq_{2,h_c\backslash ur(hc)}$ the remaining part of $h_c\backslash ur(hc)$. Define the same analogue for h'_c. Then, the previous equation is equivalent to

$$seq_{1,h_c\backslash ur(hc)} \cdot ev \cdot seq_{2,h_c\backslash ur(hc)} = seq_{1,h'_c\backslash ur(hc)} \cdot ev \cdot seq_{2,h'_c\backslash ur(hc)}.$$

This is true as $cmp(hc) = cmp(hc')$.

$\boldsymbol{IWS}_{\boldsymbol{h_{c,ins}}} = \boldsymbol{IWS}_{\boldsymbol{h'_{c,ins}}}$ **:**
There is no new transaction in both insertions. The interrupting write set of each transaction is unchanged as there no are new rf-elements in both $h_{c,ins}$ and $h'_{c,ins}$ compared to h_c and h'_c, the order of events is identical, and no events have been modified by the insertion function. As these sets are identical for h_c and h'_c, the claim follows.

$\boldsymbol{MC}_{\boldsymbol{h_{c,ins}}} = \boldsymbol{MC}_{\boldsymbol{h'_{c,ins}}}$ **:**
There are no new (abortable) rf-elements thus both sets are identical compared to the respective sets in h_c and h'_c. As these sets are identical for h_c and h'_c, the claim follows. $\square$

Proposition 18. *Given a g-event* $ev = \mathbf{Inv}_t^{tr}(\mathbf{C})$, *two arbitrary hc-pairs* (h, h_c) *and* (h', h'_c) *s.t.* $cmp(h, h_c) = cmp(h', h'_c)$, *it holds that*

$$\forall h_{c,ins} \in ins(h_c, ev) \exists h'_{c,ins} \in ins(h'_c, ev) : \\ cmp(h \cdot ev, h_{c,ins}) = cmp(h' \cdot ev, h'_{c,ins}).$$

Proof. Note that tr is not visible in $h \cdot ev$ and $h' \cdot ev$. Let $h_c = ev_0 \dots ev_{h_c}^{tr,ls} \dots ev_n$ and $h'_c = ev'_0 \dots ev_{h'_c}^{tr,ls} \dots ev'_l$. It holds that $ins(h_c, ev)$ contains exactly the element $ev_0 \dots ev_{h_c}^{tr,ls} ev \dots ev_n$ and $ins(h'_c, ev)$ contains exactly the element $ev'_0 \dots ev_{h'_c}^{tr,ls} ev \dots ev'_l$. Thus, we set $h_{c,ins}$ to the former and $h'_{c,ins}$ to the latter candidate. It holds that $ur(hc) = ur(h \cdot ev, h_{c,ins})$ as both contain the same transactions in the same order, except that for tr (a non-committed) transaction a commit invoke was added. It is easy to see that this does not affect the conditions of Lemmas 26 and 29. Analogue, $ur(hc') = ur(h' \cdot ev, h'_{c,ins})$ holds.

$\boldsymbol{(h \cdot ev) \backslash ur(h \cdot ev, h_{c,ins}) = (h' \cdot ev) \backslash ur(h' \cdot ev, h'_{c,ins})}$ **:**
Given that $ur(hc) = ur(h \cdot ev, h_{c,ins})$ and $ur(hc') = ur(h' \cdot ev, h'_{c,ins})$ holds, this

is equivalent to

$$(h \cdot ev)\backslash ur(hc) = (h' \cdot ev)\backslash ur(hc').$$

It holds that the unfinished transaction tr is not part of $ur(hc)$ and $ur(hc')$, and it holds that ev is not a read event. Thus, this is equivalent to

$$(h\backslash ur(hc)) \cdot ev = (h'\backslash ur(hc')) \cdot ev.$$

Then given that $cmp(hc) = cmp(hc')$ this is equivalent to

$$(h\backslash ur(hc)) \cdot ev = (h\backslash ur(hc)) \cdot ev,$$

which is a trivially true statement.

$\boldsymbol{h_{c,ins}\backslash ur(h \cdot ev, h_{c,ins}) = h'_{c,ins}\backslash ur(h' \cdot ev, h'_{c,ins}):}$
Given that $ur(hc) = ur(h \cdot ev, h_{c,ins})$ and $ur(hc') = ur(h' \cdot ev, h'_{c,ins})$ holds, this is equivalent to

$$h_{c,ins}\backslash ur(hc) = h'_{c,ins}\backslash ur(hc').$$

This is equivalent to

$$(ev_0 \dots ev_{h_c}^{tr,ls} ev \dots ev_n)\backslash ur(hc) = (ev'_0 \dots ev_{h'_c}^{tr,ls} ev \dots ev'_l)\backslash ur(hc').$$

Note that $h_{c,ins}.RF_c = h_c.RF_c$ and $h'_{c,ins}.RF_c = h'_c.RF_c$. Since the removal function is applied to each element and individually and checks whether it belongs to a transaction in the input set or it is a read from a transaction in the input set, its results for each event when applied above are the same as when applied to h_c and h'_c. Also, it does not alter ev for both sides of the equation as it is a commit invoke of a transaction not in $ur(hc)$ or $ur(hc')$. Let ev_i^- be the event at index i of h_c after applying $\backslash ur(hc)$ to it. Let x be the index of $ev_{h_c}^{tr,ls}$ in h_c. Let $seq_{1,h_c\backslash ur(hc)}$ denote $ev_0^- \dots ev_x^-$ and $seq_{2,h_c\backslash ur(hc)}$ the remaining part of $h_c\backslash ur(hc)$. Define the same analogue for h'_c. Then, the previous equation is equivalent to

$$seq_{1,h_c\backslash ur(hc)} \cdot ev \cdot seq_{2,h_c\backslash ur(hc)} = seq_{1,h'_c\backslash ur(hc)} \cdot ev \cdot seq_{2,h'_c\backslash ur(hc)}.$$

This is true as $cmp(hc) = cmp(hc')$.

$\boldsymbol{IWS_{h_{c,ins}} = IWS_{h'_{c,ins}}}$ **:**
There is no new transaction in both insertions. The interrupting write set of each transaction is unchanged as there are new rf-elements in both $h_{c,ins}$ and $h'_{c,ins}$ compared to h_c and h'_c. This is because tr is not visible in $h \cdot ev$ and $h' \cdot ev$, and thus aborted in the minimal completion of their candidates. As these sets are identical for h_c and h'_c, the claim follows.

$\boldsymbol{MC_{h_{c,ins}} = MC_{h'_{c,ins}}}$ **:**
As in the previous case, there are no new abortable rf-elements thus both sets are identical compared to the respective sets in h_c and h'_c. This is because tr is not visible in $h \cdot ev$ and $h' \cdot ev$, and thus aborted in the minimal completion of their candidates. As these sets are identical for h_c and h'_c, the claim follows.

□

Proposition 19. *Given a g-event* $ev = \mathbf{C}_t^{tr}$*, two arbitrary hc-pairs* (h, h_c) *and* (h', h'_c) *s.t.* $cmp(h, h_c) = cmp(h', h'_c)$*, it holds that*

$$\forall h_{c,ins} \in ins(h_c, ev) \exists h'_{c,ins} \in ins(h'_c, ev) : \\ cmp(h \cdot ev, h_{c,ins}) = cmp(h' \cdot ev, h'_{c,ins}).$$

Proof. Let $h_c = ev_0 \dots ev_{h_c}^{tr,ls} \dots ev_n$ and $h'_c = ev'_0 \dots ev_{h'_c}^{tr,ls} \dots ev'_n$. It holds that $ins(h_c, ev)$ contains exactly the element $ev_0 \dots ev_{h_c}^{tr,ls} ev \dots ev_n$ and $ins(h'_c, ev)$ contains exactly the element $ev'_0 \dots ev_{h'_c}^{tr,ls} ev \dots ev'_l$. Thus, we set $h_{c,ins}$ to the former and $h'_{c,ins}$ to the latter candidate.

Assume $IWS_{h_c}(tr) \cap WS_{h_c}^{vo}(tr) \neq \emptyset$. Then, $h.RF_{val} \neq h_c.RF_c$ holds by the definition of an interrupting write set. Thus, $cmp(h \cdot ev, h_{c,ins}) = \mathbf{DM}$ holds. It holds that $tr \notin ur(hc)$ because it is unfinished in h. Because of this and since $cmp(hc) = cmp(hc')$, it holds that $tr \in h'$ and its write set is identical in h and h'. Also, $IWS_{h_c}(tr) = IWS_{h'_c}(tr)$ holds as $IWS_{h_c} = IWS_{h'_c}$ holds because of $cmp(hc) = cmp(hc')$. Thus, $IWS_{h'_c}(tr) \cap WS^{vo}_{h'_c}(tr) \neq \emptyset$ holds. From this $h' \cdot ev.RF_{val} \neq h'_{c,ins}.RF_c$ follows given the definition of an interrupting write set. Thus,

$$cmp(h' \cdot ev, h'_{c,ins}) = cmp(h \cdot ev, h_{c,ins}) = \mathbf{DM}$$

holds.

We now show each element of both compression tuples is identical for the case $IWS_{h_c}(tr) \cap WS^{vo}_{h_c}(tr) = \emptyset$. For this we prove that

$$ur(h \cdot ev, h_{c,ins}) \backslash ur(hc) = ur(h' \cdot ev, h'_{c,ins}) \backslash ur(hc') = X,$$

in Proposition 20.

$(h \cdot ev) \backslash ur(h \cdot ev, h_{c,ins}) = (h' \cdot ev) \backslash ur(h' \cdot ev, h'_{c,ins})$:

As discussed in the proof of Proposition 20, it holds that

$$ur(h \cdot ev, h_{c,ins}) \backslash ur(hc) = ur(h' \cdot ev, h'_{c,ins}) \backslash ur(hc') = X.$$

Also, we show in Proposition 21 that the following holds:

$$(h \cdot ev) \backslash ur(h \cdot ev, h_{c,ins}) = (h \cdot ev \backslash ur(hc)) \backslash X.$$

Given that $tr \notin ur(hc)$ and that ev is not a read event, it follows that

$$(h \cdot ev) \backslash ur(h \cdot ev, h_{c,ins}) = ((h \backslash ur(hc)) \cdot ev) \backslash X.$$

By $cmp(hc) = cmp(hc')$, it follows that

$$(h \cdot ev) \backslash ur(h \cdot ev, h_{c,ins}) = ((h' \backslash ur(hc')) \cdot ev) \backslash X.$$

Then, given that $tr \notin ur(hc')$,

$$ur(h \cdot ev, h_{c,ins}) \backslash ur(hc) = ur(h' \cdot ev, h'_{c,ins}) \backslash ur(hc') = X,$$

and by Proposition 21 it follows that

$$(h \cdot ev) \backslash ur(h \cdot ev, h_{c,ins}) = (h' \cdot ev) \backslash ur(h' \cdot ev, h'_{c,ins}).$$

$h_{c,ins} \backslash ur(h \cdot ev, h_{c,ins}) = h'_{c,ins} \backslash ur(h' \cdot ev, h'_{c,ins})$:
As discussed in the proof of Proposition 20, it holds that

$$ur(h \cdot ev, h_{c,ins}) \backslash ur(hc) = ur(h' \cdot ev, h'_{c,ins}) \backslash ur(hc') = X.$$

Given $cmp(hc) = cmp(hc')$, we define

$$h_c \backslash ur(hc) = ev^a_0 \dots ev^a_n = h'_c \backslash ur(hc').$$

For the sake of simplicity, we assume that we do consider two event sequences equal whenever the subsequences of them with all ϵ events removed are equal. Given Proposition 21, $h_{c,ins} \backslash ur(h \cdot ev, h_{c,ins}) = h'_{c,ins} \backslash ur(h' \cdot ev, h'_{c,ins})$ is equivalent to

$$((ev_0 \dots ev \dots ev_n) \backslash ur(hc)) \backslash X = (ev'_0 \dots ev \dots ev'_l) \backslash ur(hc')) \backslash X.$$

To show this, we show

$$(ev_0 \dots ev \dots ev_n) \backslash ur(hc) = (ev'_0 \dots ev \dots ev'_l) \backslash ur(hc').$$

Firs,t as ev is a commit and $tr \notin ur(hc)$, it is trivially not changed by the removal function. Then, let

$$(ev_0 \dots ev \dots ev_n) \backslash ur(hc) = ev^b_0 \dots ev \dots ev^b_n.$$

We prove that for an arbitrary index i with $0 \leq i \leq n$, that $ev_i^a = ev_i^b$. We do this via case distinction for an arbitrary g-event ev_x :

$\boldsymbol{tr_{h_{c,ins}}(ev_x) \in ur(hc)}$ **:**
In this case $ev_x^b = \epsilon$ by Definition 49. Also, as $h_{c,ins}$ is h_c with ev inserted it holds that

$$tr_{h_{c,ins}}(ev_x) = tr_{h_c}(ev_x),$$

so $ev_x^a = \epsilon = ev_x^b$.

$\boldsymbol{ev_x = \mathbf{R}_{t'}^{tr'}(y, val'), \exists tr'' \in ur(hc) \exists y \in Var : (tr'', tr', y) \in h_{c,ins}.RF_c}$ **:**
Let $tr'' \in ur(hc)$ be the transaction s.t.

$$ev_x = \mathbf{R}_{thr(tr')}^{tr'}(y, val'), (tr'', tr', y) \in h_{c,ins}.RF_c.$$

We do a case distinction over whether $tr \in ur(h \cdot ev, h_{c,ins})$ or not. It holds that $tr \in ur(h \cdot ev, h_{c,ins})$ iff it is not the last transaction in h_c and its write set is empty. Else it is readable in $h \cdot ev$ as its commit ev is the last g-event of $h \cdot ev$, or it is not in the unreadable set because it is the last transaction in $h_{c,ins}$. If the write set of tr is empty, then in this case it holds that $h_c.RF_c = h_{c,ins}.RF_c$. Thus,

$$(tr'', tr', y) \in h_{c,ins}.RF_c \leftrightarrow (tr'', tr', y) \in h_c.RF_c,$$

and thus $ev_x^a = ev_x^b = \epsilon$.

Now if $tr \notin ur(h \cdot ev, h_{c,ins})$ its write set is not empty, or it is the last transaction in $h_{c,ins}$ and its write set is empty. In the first case, if there were to exist an rf-element with tr as its writer in $h_{c,ins}$ that did not exist in h_c, then $IWS_{h_c}(tr) \cap WS_{h_c}^{vo}(tr) \neq \emptyset$. In this part of the proof, we explicitly assume this not to be the case as discussed before. So, inserting ev does not remove from any rf-elements from h_c or add rf-elements to h_c. Thus, $(tr'', tr', y) \in h_{c,ins}.RF_c \leftrightarrow (tr'', tr', y) \in h_c.RF_c$, and thus $ev_x^a = ev_x^b = \epsilon$.

In the second case, where the write set of tr is empty, it trivially cannot be involved in rf-elements, and thus $(tr'', tr', y) \in h_{c,ins}.RF_c \leftrightarrow (tr'', tr', y) \in h_c.RF_c$, and thus $ev^a_x = ev^b_x = \epsilon$.

else:
We have shown for all other cases they hold for ev_x in h_c iff they hold for it in $h_{c,ins}$. Thus, in the "else" case also holds for ev_x in h_c iff it holds for it in $h_{c,ins}$. Given that, in this case $ev^a_x = ev^b_x = ev_x$ holds.

Thus, we have shown that

$$((ev_0 \dots ev \dots ev_n) \backslash ur(hc)) \backslash X = (ev'_0 \dots ev \dots ev'_n) \backslash ur(hc')) \backslash X.$$

From which it follows that

$$h_{c,ins} \backslash ur(h \cdot ev, h_{c,ins}) = h'_{c,ins} \backslash ur(h' \cdot ev, h'_{c,ins}).$$

$\boldsymbol{IWS_{h_{c,ins}} = IWS_{h'_{c,ins}}}$ **:**
As ev is a commit, and $h_c.RF_c = h_{c,ins}.RF_c$ (else as discussed above the resulting hc-pair is in **DM**), the interrupting write set of any transaction but tr in h_c is identical to its counterpart in $h_{c,ins}$. By $cmp(hc) = cmp(hc')$, it follows that $h_c.RF_c = h'_c.RF_c$, and by an analogue argument to the one above it follows that $h'_c.RF_c = h'_{c,ins}.RF_c$. As tr is committed in $h_{c,ins}$, it holds that $IWS_{h_{c,ins}}(tr) = \emptyset$. By an analogue argument, it holds that $IWS_{h'_{c,ins}}(tr) = \emptyset$. Thus, the claim follows.

$\boldsymbol{MC_{h_{c,ins}} = MC_{h'_{c,ins}}}$ **:**
If $tr \in MC_{h_c}$ it follows that $tr \notin MC_{h_{c,ins}}$. If $tr \notin MC_{h_c}$ it follows that $tr \notin MC_{h_{c,ins}}$. As $tr \notin ur(hc)$, and $cmp(hc) = cmp(hc')$, in both cases it follows that $tr \notin MC_{h_{c,ins}}$. As $MC_{h_c} = MC_{h'_c}$, the claim follows. □

Proposition 20. *Given a g-event $ev = \mathbf{C}_t^{tr}$, two arbitrary hc-pairs (h, h_c) and (h', h'_c) s.t. $cmp(h, h_c) = cmp(h', h'_c)$, it holds that*

$$\forall h_{c,ins} \in ins(h_c, ev), \forall h'_{c,ins} \in ins(h'_c, ev) :$$
$$ur(h \cdot ev, h_{c,ins}) \backslash ur(hc) = ur(h' \cdot ev, h'_{c,ins}) \backslash ur(hc').$$

Proof. We will first handle to specific cases to then prove the remaining cases together after.

$\boldsymbol{WS_h^{vo}(tr) = \emptyset}$ **and** $\boldsymbol{tr \neq tr_{lf}(h_c)}$ **:**
From the assumption and Lemma 31, it follows that $tr \in ur(h \cdot ev)$ and $tr \in ur(h_{c,ins})$. Trivially, all other readable transactions in h and h_c are then also readable in their extended counterparts. It follows that

$$ur(h \cdot ev, h_{c,ins}) \backslash ur(hc) = \{tr\}.$$

Given that $tr \notin ur(hc)$ as it is unfinished in hc, it is also not in $ur(hc')$. Then by this and the fact that $cmp(hc) = cmp(hc')$, it holds that $WS_{h'}^{vo}(tr) = \emptyset$, and that tr is not the last transaction in h'_c. Thus, also $ur(h' \cdot ev, h'_{c,ins}) \backslash ur(hc') = \{tr\}$.

$\boldsymbol{WS_h^{vo}(tr) = \emptyset}$ **and** $\boldsymbol{tr \neq tr_{lf}(h_c)}$ **:**
From the assumption and Lemmas 28 and 29, it follows that $tr \notin ur(h \cdot ev)$ and $tr \notin ur(h_{c,ins})$. Trivially, all other readable transactions in h and h_c are then also readable in their extended counterparts. It follows that $ur(h \cdot ev, h_{c,ins}) \backslash ur(hc) = \emptyset$. Given that $tr \notin ur(hc)$ as it is unfinished in hc, it is also not in $ur(hc')$. Then by this and the fact that $cmp(hc) = cmp(hc')$, it holds that $WS_{h'}^{vo}(tr) = \emptyset$, and that tr is not the last transaction in h'_c. Thus, also $ur(h' \cdot ev, h'_{c,ins}) \backslash ur(hc) = \emptyset$.

In the further we thus assume $WS_h^{vo}(tr) \neq \emptyset$. This trivially implies that $tr \notin ur(h \cdot ev, h_{c,ins})$ and $tr \notin ur(h' \cdot ev, h'_{c,ins})$ as tr is readable in $h \cdot ev$ and $h' \cdot ev$ with its commit being the last event of them. We first show $ur(h \cdot ev) \backslash ur(h) = ur(h' \cdot ev) \backslash ur(h')$ and then that $(ur(h_{c,ins}) \backslash ur(h_c)) \backslash \{tr\} = (ur(h'_{c,ins}) \backslash ur(h'_c)) \backslash \{tr\}$. We exclude tr from the latter because as mentioned above it holds that $tr \notin ur(h \cdot ev, h_{c,ins})$ and $tr \notin ur(h' \cdot ev, h'_{c,ins})$.

$\boldsymbol{ur(h \cdot ev) \backslash ur(h) = ur(h' \cdot ev) \backslash ur(h')}$ **:**
We first define the needed notation. For the sake of this, let tr_x be a committed transaction in an arbitrary g-history h and var be a variable in the write set of tr_x. We define $obe(h, tr_x, var) = \exists tr' \in Tr : co_h(tr') \wedge var \in WS_h^{vo}(tr') \wedge tr_x \prec_h tr'$. Then, we define $RdV_h(tr_x) = \{var \in WS_h^{vo}(tr_x) \mid \neg obe(h, tr_x, var)\}$, which is the set of variables of tr_x which are not overwritten before the end of h. By Lemma 28, it holds that $RdV_h(tr_x) = \emptyset$, iff tr_x is in $ur(h)$.

We now prove $ur(h \cdot ev) \backslash ur(h) = ur(h' \cdot ev) \backslash ur(h')$. First, note again that $tr \notin ur(h \cdot ev)$ as we assume its write set to not be empty. We show the following two formulae hold true, which then shows the overall claim to be true:

$$\begin{gathered} \forall tr_x \in Tr \backslash \{tr\} : tr_x \in ur(h \cdot ev) \backslash ur(h) \\ \leftrightarrow \\ (RdV_h(tr_x) \neq \emptyset \wedge RdV_h(tr_x) \subseteq WS_h^{vo}(tr) \wedge tr_x \prec_h tr), \end{gathered} \tag{C.4}$$

and

$$\forall tr_x \in h, co_{h_c}(tr_x), tr_x \notin ur(hc), tr_x \neq tr : RdV_h(tr_x) = RdV_{h'}(tr_x). \tag{C.5}$$

We show both directions of the first statement and then the second statement.

$\boldsymbol{\forall tr_x \in (Tr \backslash \{tr\}) : tr_x \in ur(h \cdot ev) \backslash ur(h) \rightarrow (RdV_h(tr_x) \neq \emptyset \wedge RdV_h(tr_x) \subseteq WS_h^{vo}(tr) \wedge tr_x \prec_h tr)}$ **:**
Note that tr_x can only be in $ur(h \cdot ev)$ if it is finished in h as here the case $tr_x = tr$ is excluded. Then, from $tr_x \notin ur(h)$, $RdV_h(tr_x) \neq \emptyset$ follows.

Given that $tr_x \in ur(h \cdot ev)$ holds, it then also holds that $RdV_{h \cdot ev}(tr_x) = \emptyset$. And thus, $\forall var \in RdV_h(tr_x)$ it holds that

$$obe(h \cdot ev, tr_x, var), \text{ and } \neg obe(h, tr_x, var).$$

As all g-events except ev are identical and have the same order in h and $h \cdot ev$, it must hold that

$$RdV_h(tr_x) \subseteq WS_h^{vo}(tr) \wedge tr_x \prec_h tr.$$

$\boldsymbol{\forall tr_x \in (Tr \backslash \{tr\}):}$
$\boldsymbol{(RdV_h(tr_x) \neq \emptyset \wedge RdV_h(tr_x) \subseteq WS_h^{vo}(tr) \wedge tr_x \prec_h tr) \rightarrow}$
$\boldsymbol{tr_x \in ur(h \cdot ev) \backslash ur(h):}$

From $RdV_h(tr_x) \neq \emptyset$ it follows that $tr_x \notin ur(h)$ holds. For any variable var in $RdV_h(tr_x)$, it holds that $var \in WS_h^{vo}(tr)$. Also, $tr_x \prec_h tr$ holds (which thus also holds for $h \cdot ev$), and tr is committed in $h \cdot ev$. Thus, it is true that $var \notin RdV_{h \cdot ev}(tr_x)$. As var is an arbitrary variable in $RdV_h(tr_x)$, it follows that $RdV_{h \cdot ev}(tr_x) = \emptyset$. This implies $tr_x \in ur(h \cdot ev)$.

$\boldsymbol{\forall tr_x \in h, co_{h_c}(tr_x), tr_x \notin ur(hc), tr_x \neq tr : RdV_h(tr_x) = RdV_{h'}(tr_x):}$
Let tr_x be an arbitrary transaction meeting the requirements of the all quantifier of the claim. The claim then is equivalent to

$$\forall var \in WS_h^{vo}(tr_x) : obe(h, tr_x, var) \leftrightarrow obe(h', tr_x, var),$$

as $WS_h^{vo}(tr_x) = WS_{h'}^{vo}(tr_x)$ because of $cmp(hc) = cmp(hc')$. We will show the direction from the left to right, the reverse direction can be proven analogue. Let var be an arbitrary variable in $WS_h^{vo}(tr_x)$ s.t. $obe(h, tr_x, var)$ is true. By definition of obe, it holds that

$$\exists tr' \in Tr : co_h(tr') \wedge var \in WS_h^{vo}(tr') \wedge tr_x \prec_h tr'.$$

Then, it also holds that

$$\begin{array}{l} \exists tr' \in Tr : co_h(tr') \wedge var \in WS_h^{vo}(tr') \wedge tr_x \prec_h tr' \wedge \\ \neg(\exists tr'' : var \in WS_h^{vo}(tr'') \wedge tr' \prec_h tr'') \end{array}.$$

Let tr' be an arbitrary transaction s.t. the above formula is true for it when substituting tr' with it. Then, trivially $\neg obe(h, tr', var)$ holds. Thus, $tr' \notin ur(h)$ holds from which $tr' \notin ur(hc)$ follows. This is then equivalent to $tr' \notin ur(hc')$ as $cmp(hc) = cmp(hc')$. Then, from $tr_x \prec_{h \backslash ur(hc)} tr'$ it follows that $tr_x \prec_{h' \backslash ur(hc')} tr'$ as $cmp(hc) = cmp(hc')$. From this it follows that $tr_x \prec_{h'} tr'$. As the events of both transactions excluding reads are identical in h and h' since again $cmp(hc) = cmp(hc')$ holds, $obe(h', tr_x, var)$ follows which proves this direction.

We can now use these two statements. Given $tr_x \in ur(h \cdot ev) \backslash ur(h)$, $tr_x \neq tr$, by Formula C.4 this is equivalent to

$$(RdV_h(tr_x) \neq \emptyset \wedge RdV_h(tr_x) \subseteq WS_h^{vo}(tr) \wedge tr_x \prec_h tr).$$

Note that for any transaction tr_x in h or h' s.t. that it is not in $ur(hc)$ or $ur(hc')$ it holds $tr_x \prec_h tr \leftrightarrow tr_x \prec_{h'} tr$, $co_h(tr) \leftrightarrow co_{h'}(tr)$ and $WS_h^{vo}(tr) = WS_{h'}^{vo}(tr)$. This is because of $cmp(hc) = cmp(hc')$. Then, by Formula C.5 the above statement is equivalent to

$$(RdV_{h'}(tr_x) \neq \emptyset \wedge RdV_{h'}(tr_x) \subseteq WS_{h'}^{vo}(tr) \wedge tr_x \prec_{h'} tr).$$

Then by Formula C.4 this is equivalent to $tr_x \in ur(h' \cdot ev) \backslash ur(h')$. Overall it follows that $ur(h \cdot ev) \backslash ur(h) = ur(h' \cdot ev) \backslash ur(h')$.

$\boldsymbol{(ur(h_{c,ins})\backslash ur(h_c))\backslash\{tr\} = (ur(h'_{c,ins})\backslash ur(h'_c))\backslash\{tr\}}$ **:**
We first define the needed notation. For the sake of the definition let his_c be an arbitrary candidate, tr' be an arbitrary transaction in Tr_{unc,h_c}, tr_x be an arbitrary committed transaction in h_c and var be a member of the write set of tr_x. We first define

$$obt(h_c, tr_x, tr', var) =$$
$$\neg(tr' \prec_{h_c} tr_x) \vee \exists tr'' \in Tr : \mathbf{C}_{h_c}^{tr''} \wedge var \in WS_{h_c}^{vo}(tr'') \wedge tr_x \prec_{h_c} tr'' \prec_{h_c} tr'.$$

Next we define $RdV_{h_c}(tr_x, tr') = \{var \in WS_{h_c}^{vo}(tr_x) \mid \neg obt(h_c, tr_x, tr', var)\}$, which is the set of variables written to by tr_x that can be read by tr' in h_c. W.l.o.g. for this proof we assume that at the end of a candidate there is a committed transaction which reads every variable once and is real-time ordered after each other transaction. This allows us to model the *obe* and the corresponding *RdV* sets by *obt* and its *RdV* sets, through choosing the all reading transaction at the end as the third argument. By Lemma 29, it holds for all unfinished and not commit pending transactions tr' that $RdV_{h_c}(tr_x, tr') = \emptyset$ iff $tr_x \in ur(h_c)$.

Now we prove $(ur(h_{c,ins})\backslash ur(h_c))\backslash\{tr\} = (ur(h'_{c,ins})\backslash ur(h'_c))\backslash\{tr\}$. We show two formulae hold true, which then shows the overall claim to be true. Note that we denote the set of all unfinished and not commit or abort pending transactions in a candidate h_c as Tr_{unc,h_c}.

$$\forall tr_x \in (Tr\backslash\{tr\}) : \left(\begin{array}{c} tr_x \in ur(h_{c,ins})\backslash ur(h_c) \\ \leftrightarrow \\ \forall tr' \in Tr_{unc,h_c} : RdV_{h_c}(tr_x, tr') \neq \emptyset \wedge, RdV_{h_c}(tr_x, tr') \subseteq WS_{h_c}^{vo}(tr) \wedge tr_x \prec_{h_{c,ins}} tr \prec_{h_{c,ins}} tr' \end{array}\right) \tag{C.6}$$

and

$$\forall tr_x \in h_c, tr_x \neq tr, tr_x \notin ur(hc), co_{h_c}(tr_x), \forall tr' \in h_c : RdV_{h_c}(tr_x, tr') = RdV_{h'_c}(tr_x, tr'). \tag{C.7}$$

We show both directions of Formula C.6 and then Formula C.7.

$\boldsymbol{\forall tr_x \in (Tr\backslash\{tr\}) : tr_{\mathrm{x}} \in ur(h_{c,ins})\backslash ur(h_c) \rightarrow}$

$\boldsymbol{\forall tr' \in Tr_{\mathbf{unc},h_c}}$ **:**
$\boldsymbol{(RdV_h(tr_{\mathrm{x}}, tr') \neq \emptyset \wedge RdV_{h_c}(tr_{\mathrm{x}}, tr') \subseteq WS^{vo}_{h_c}(tr) \wedge tr_x \prec_{h_{\mathbf{c,ins}}} tr \prec_{h_{\mathbf{c,ins}}} tr')}$ **:**
Fix tr_x to an arbitrary transaction in $ur(h_{c,ins})\backslash ur(h_c)$. From $tr_x \notin ur(h_c)$ it follows that

$$\exists tr' \in Tr_{unc,h_c} : RdV_{h_c}(tr_x, tr') \neq \emptyset,$$

and from $tr_x \notin ur(h_{c,ins})$ it follows that

$$\forall tr' \in Tr_{unc,h_c} : RdV_{h_{c,ins}}(tr_x, tr') = \emptyset.$$

Note that $Tr_{unc,h_c} = Tr_{unc,h_{c,ins}}$. Let tr' be an arbitrary transaction in the set $Tr_{unc,h_{c,ins}}$. Assume it holds $\neg(tr_x \prec_{h_{c,ins}} tr \prec_{h_{c,ins}} tr')$ then trivially it also holds that $RdV_{h_c}(tr_x, tr') = RdV_{h_{c,ins}}(tr_x, tr')$ by the definition of the insertion function which is a contradiction. Thus, $tr_x \prec_{h_{c,ins}} tr \prec_{h_{c,ins}} tr'$ holds. Now if $\neg(RdV_h(tr_x, tr') \subseteq WS^{vo}_h(tr))$ holds, consider an arbitrary variable var' in the former but not in the latter set. Then it holds that $\neg obt(h_{c,ins}, tr_x, tr', var')$ as it held in h_c and the only change in $h_{c,ins}$ is that tr is committed. In this case, by the conditions in Lemma 29 tr_x would not be in $ur(h_{c,ins})$ which is in contradiction to the assumption. Thus, it holds that $RdV_h(tr_x, tr') \subseteq WS^{vo}_h(tr)$, and the implication overall holds.

$\boldsymbol{\forall tr_x \in (Tr\backslash\{tr\}) : (\forall tr' \in Tr_{unc,h_c} : (RdV_h(tr_x, tr') \neq \emptyset \wedge RdV_{h_c}(tr_x, tr') \subseteq WS^{vo}_{h_c}(tr) \wedge tr_x \prec_{h_{c,ins}} tr \prec_{h_{c,ins}} tr') \rightarrow tr_x \in ur(h_{c,ins})\backslash ur(h_c))}$:
Consider an arbitrary $tr_x \in Tr$. From $\exists tr' \in Tr_{unc,h_c} : RdV_{h_c}(tr_x, tr') \neq \emptyset$ it follows that $tr_x \notin ur(h_c)$. Consider an arbitrary transaction tr' s.t. $RdV_{h_c}(tr_x, tr') \neq \emptyset$. From $RdV_h(tr_x, tr') \subseteq WS^{vo}_h(tr)$, $tr_x \prec_{h_{c,ins}} tr \prec_{h_{c,ins}} tr'$, $co_{h_{c,ins}}(tr)$ and that, except the insertion of ev, $h_{c,ins}$ is identical to h_c it follows that $RdV_{h_{c,ins}}(tr_x, tr') = \emptyset$. Since this holds for any transaction tr' s.t. $RdV_{h_c}(tr_x, tr') \neq \emptyset$, it follows that $tr_x \in ur(h_{c,ins})$. Thus, we have

shown for an arbitrary $tr_x \in (Tr \backslash \{tr\})$ it holds that $tr_x \notin ur(h_c)$ and $tr_x \in ur(h_{c,ins})$, if the left-hand side of the implication is true, which proves the implication is true.

$\boldsymbol{\forall tr_x \in h_c, tr_x \notin ur(hc), fin_{tr_x}(h_c), \forall tr' \in h_c : RdV_{h_c}(tr_x, tr') = RdV_{h'_c}(tr_x, tr') :}$
Fix tr_x and tr' to transactions meeting the conditions of the quantifiers. Given that $tr_x \notin ur(hc)$ and $cmp(hc) = cmp(hc')$, it holds that $tr_x \in h'_c$. Given that tr' is unfinished, $tr' \notin ur(hc)$ holds. From this and additionally $cmp(hc) = cmp(hc')$, $tr' \in h'_c$ follows. Now, we show that for any variable var $obt(h_c, tr_x, tr', var) \leftrightarrow obt(h'_c, tr_x, tr', var)$. We only show the left to right direction. The other direction follows analogue. First, assume that $\neg(tr' \prec_{h_c} tr_x)$ holds. This is equivalent to $\neg(tr' \prec_{h'_c} tr_x)$ because both transactions are not in $ur(hc)$ and $ur(hc')$ and $cmp(hc) = cmp(hc')$. Now, assume $\exists tr'' \in Tr : \mathbf{C}_{h_c}^{tr''} \wedge var \in WS_{h_c}^{vo}(tr'') \wedge tr_x \prec_{h_c} tr'' \prec_{h_c} tr'$. Fix tr'' to a transaction s.t. the above formula is true, and it holds that

$$\neg(\exists tr''' \in Tr : tr_x \prec_{h_c} tr''' \prec_{h_c} tr' \wedge tr_x \prec_{h_c} tr'' \prec_{h_c} tr''').$$

Such a transaction trivially exists in a candidate. Then, by Lemma 29 $tr'' \notin ur(h_c)$ and thus $tr'' \notin ur(hc), tr'' \notin ur(hc')$ as $cmp(hc) = cmp(hc')$. Then, it holds that

$$\exists tr'' \in Tr : \mathbf{C}_{h'_c}^{tr''} \wedge var \in WS_{h'_c}^{vo}(tr'') \wedge tr_x \prec_{h'_c} tr'' \prec_{h'_c} tr'$$

as $tr_x, tr'', tr' \notin ur(hc)$ and $cmp(hc) = cmp(hc')$. Thus, it follows that $obt(h_c, tr_x, tr', var) \rightarrow obt(h'_c, tr_x, tr', var)$. Given the other direction is analogue, it follows that

$$obt(h_c, tr_x, tr', var) \leftrightarrow obt(h'_c, tr_x, tr', var).$$

From this and the definition of RdV

$$\forall tr_x \in h_c, tr_x \notin ur(hc), fin_{tr_x}(h_c), \forall tr' \in h_c : RdV_{h_c}(tr_x, tr') = RdV_{h'_c}(tr_x, tr')$$

follows.

Now given Formula C.6 holds $tr_x \in (ur(h_{c,ins}) \backslash ur(h_c)) \backslash \{tr\}$ is equivalent to

$$\forall tr' \in Tr_{unc,h_c} : RdV_{h_c}(tr_x, tr') \neq \emptyset \wedge RdV_{h_c}(tr_x, tr') \subseteq WS^{vo}_{h_c}(tr) \wedge tr_x \prec_{h_{c,ins}} tr \prec_{h_{c,ins}} tr',$$

which by the Formula C.7, $tr, tr_x \notin ur(hc)$, the fact that any transaction in Tr_{unc,h_c} also occurs in Tr_{unc,h'_c} and $cmp(hc) = cmp(hc')$, is equivalent to

$$\forall tr' \in Tr_{unc,h'_c} : RdV_{h'_c}(tr_x, tr') \neq \emptyset \wedge RdV_{h'_c}(tr_x, tr') \subseteq WS^{vo}_{h'_c}(tr) \wedge tr_x \prec_{h'_{c,ins}} tr \prec_{h'_{c,ins}} tr')$$

which by Formula C.6 is equivalent to $tr_x \in (ur(h'_{c,ins}) \backslash ur(h'_c)) \backslash \{tr\}$.

We have shown that

$$ur(h \cdot ev) \backslash ur(h) = ur(h' \cdot ev) \backslash ur(h'),$$

and then that

$$(ur(h_{c,ins}) \backslash ur(h_c)) \backslash \{tr\} = (ur(h'_{c,ins}) \backslash ur(h'_c)) \backslash \{tr\}.$$

Also, it holds that

$$tr \notin ur(h \cdot ev, h_{c,ins}) \text{ and } tr \notin ur(h' \cdot ev, h'_{c,ins}).$$

Thus,

$$ur(h \cdot ev, h_{c,ins}) \backslash ur(hc) = ur(h' \cdot ev, h'_{c,ins}) \backslash ur(hc')$$

follows for this case. As we have proven this as well for the other case at the start of this proof, the overall claim follows. □

Proposition 21. *Given an arbitrary history h, its candidate h_c and two sets of transactions Tr_x, Tr_y, s.t. the transactions in both sets occur in h, it holds that*

1. $h \backslash (Tr_x \cup Tr_y) = (h \backslash \{Tr_x\}) \backslash \{Tr_y\}$
2. *and* $h_c \backslash (Tr_x \cup Tr_y) = (h_c \backslash \{Tr_x\}) \backslash \{Tr_y\}$.

Proof. Let $h = ev_0 \dots ev_n$, $h \backslash (Tr_x \cup Tr_y) = ev'_0 \dots ev'_n$, $h \backslash \{Tr_x\} = ev''_0 \dots ev''_n$ and $h \backslash \{Tr_x\} \backslash \{Tr_y\} = ev'''_0 \dots ev'''_n$. Here we assume the results each contain the ϵ "events" used in Definition 49, so we can avoid convoluted index mappings. We show

$$\forall i, 0 \leq i \leq n : ev'_i = ev'''_i.$$

Let i be an arbitrary natural number between 0 and n. We give the proof for h, the proof for h_c is analogue replacing RF_{val} with RF_c. We proceed via case distinction.

$\boldsymbol{tr_h(ev_i) \in Tr_x :}$
Then, by Definition 49 $ev'_i = \epsilon$ and $ev''_i = \epsilon$ from which $ev'''_i = \epsilon$ trivially follows.

$\boldsymbol{tr_h(ev_y) \in Tr_y :}$
Then, by Definition 49 $ev'_i = \epsilon$ and $ev'''_i = \epsilon$ holds.

$\boldsymbol{ev_i = \mathrm{R}^{tr}_t(z, val), \exists tr' \in Tr_x : (tr', tr, z) \in h.RF_{val} :}$
Then, by Definition 49 $ev'_i = \epsilon$ and $ev''_i = \epsilon$ hold, from which $ev'''_i = \epsilon$ trivially follows.

$\boldsymbol{ev_i = \mathrm{R}^{tr}_t(z, val), \exists tr' \in Tr_y : (tr', tr, z) \in h.RF_{val} :}$
Let $tr' \in Tr_y$ be the transaction s.t. $(tr', tr, z) \in h.RF_{val}$. If tr' is also in Tr_x, the previous case shows the claim.

So we show the claim for $tr' \notin Tr_x$ in the following. By Definition 49, $ev'_i = \epsilon$ holds. By Lemma 32, we know that $(h \setminus Tr_x).RF_{val} = h.RF_{val} \setminus \{rf \in h.RF_{val} \mid \exists tr \in Tr_x : tr \in rf\}$. Thus, as $tr' \notin Tr_x$, it holdsthat $(tr', tr, z) \in (h \setminus Tr_x).RF_{val}$. By Definition 49, the removal function does not change the order of events nor does it add events. So each event of tr' has the same index in $ev_0 \dots ev_n$, $ev''_0 \dots ev''_n$, and $ev'''_0 \dots ev'''_n$. Combined with the fact that $(tr', tr, z) \in (h \setminus Tr_x).RF_{val}$ and by Definition 49, it follows that $ev'''_i = \epsilon$.

else:
In this case by Lemma 32, $ev'_i = ev_i = ev''_i = ev'''_i$. holds.

□

Proposition 22. *Given a g-event $ev = \mathbf{A}_t^{tr}$, two arbitrary hc-pairs (h, h_c) and (h', h'_c) s.t. $cmp(h, h_c) = cmp(h', h'_c)$, it holds that*

$$\forall h_{c,ins} \in ins(h_c, ev) \exists h'_{c,ins} \in ins(h'_c, ev) :$$
$$cmp(h \cdot ev, h_{c,ins}) = cmp(h' \cdot ev, h'_{c,ins}).$$

Proof. We first prove the case $tr \in MC_{h_c}$. Then, $cmp(h \cdot ev, h_{c,ins}) = \mathbf{DM}$ holds. Also, because of $cmp(hc) = cmp(hc')$, it then holds that $tr \in MC_{h'_c}$, and thus $cmp(h' \cdot ev, h'_{c,ins}) = \mathbf{DM}$. So, the proposition is true for this case.

Thus, in the further we assume $tr \notin MC_{h_c}$ and $tr \notin MC_{h'_c}$. Let $h_c = ev_0 \dots ev_{h_c}^{tr,ls} \dots ev_n$ and $h'_c = ev'_0 \dots ev_{h'_c}^{tr,ls} \dots ev'_l$. It holds that $ins(h_c, ev)$ contains exactly the element $ev_0 \dots ev_{h_c}^{tr,ls} ev \dots ev_n$ and $ins(h'_c, ev)$ contains exactly the element $ev'_0 \dots ev_{h'_c}^{tr,ls} ev \dots ev'_l$. Thus, we set $h_{c,ins}$ to the former and $h'_{c,ins}$ to the latter candidate. It holds that

$$ur(hc) \cup \{tr\} = ur(h \cdot ev, h_{c,ins}),$$

as both contain the same transactions in the same order, except that tr is aborted and thus unreadable in $h \cdot ev$ and $h_{c,ins}$. It is easy to see that this does not affect the conditions of Lemmas 26 and 29. Analogue $ur(hc') \cup \{tr\} = ur(h' \cdot ev, h'_{c,ins})$

holds.

$\boldsymbol{(h \cdot ev)\backslash ur(h \cdot ev, h_{c,ins}) = (h' \cdot ev)\backslash ur(h' \cdot ev, h'_{c,ins}) :}$
Given $ur(hc) \cup \{tr\} = ur(h \cdot ev, h_{c,ins})$ and $ur(hc') \cup \{tr\} = ur(h' \cdot ev, h'_{c,ins})$ holds, this is equivalent to

$$(h \cdot ev)\backslash(ur(hc) \cup \{tr\}) = (h' \cdot ev)\backslash(ur(hc') \cup \{tr\}).$$

It holds that *tr* is not part of $ur(hc)$ and $ur(hc')$, thus by Proposition 21 this is equivalent to

$$((h\backslash ur(hc)) \cdot ev)\backslash\{tr\} = ((h'\backslash ur(hc')) \cdot ev)\backslash\{tr\}.$$

As $cmp(hc) = cmp(hc')$ holds, this is equivalent to

$$((h\backslash ur(hc)) \cdot ev)\backslash\{tr\} = ((h\backslash ur(hc)) \cdot ev)\backslash\{tr\}),$$

which is a trivially true statement.

$\boldsymbol{h_{c,ins}\backslash ur(h \cdot ev, h_{c,ins}) = h'_{c,ins}\backslash ur(h' \cdot ev, h'_{c,ins}) :}$
Given $ur(hc) \cup \{tr\} = ur(h \cdot ev, h_{c,ins})$ and $ur(hc') \cup \{tr\} = ur(h' \cdot ev, h'_{c,ins})$ holds, by Proposition 21 this is equivalent to

$$h_{c,ins}\backslash ur(hc)\backslash\{tr\} = h'_{c,ins}\backslash ur(hc')\backslash\{tr\}.$$

This is equivalent to

$$(ev_0 \dots ev_{h_c}^{tr,ls} ev \dots ev_n)\backslash ur(hc)\backslash\{tr\} = (ev'_0 \dots ev_{h'_c}^{tr,ls} ev \dots ev'_l)\backslash ur(hc')\backslash\{tr\}.$$

Note that $h_{c,ins}.RF_c = h_c.RF_c$, $h'_{c,ins}.RF_c = h'_c.RF_c$ and $tr \notin ur(hc)$ Since the removal function is applied to each element, and it individually and checks whether it belongs to a transaction in the input set or it is a read from a transaction in the input set, its results for each event when applied above are the same as when applied to h_c and h'_c. Also, it does not alter *ev* for both sides of the equation as it is a write of a transaction not in $ur(hc)$ or $ur(hc')$. Let ev_i^- be the event at index i of h_c after applying $\backslash ur(hc)$ to it. Let x be the index of $ev_{h_c}^{tr,ls}$ in h_c. Let $seq_{1,h_c\backslash ur(hc)}$ denote $ev_0^- \dots ev_x^-$ and $seq_{2,h_c\backslash ur(hc)}$ the remaining part of $h_c\backslash ur(hc)$. Define the same analogue for h_c'. Then the previous equation is equivalent to

$$(seq_{1,h_c\backslash ur(hc)} \cdot ev \cdot seq_{2,h_c\backslash ur(hc)})\backslash\{tr\} = (seq_{1,h_c'\backslash ur(hc)} \cdot ev \cdot seq_{2,h_c'\backslash ur(hc)})\backslash\{tr\}.$$

This is true as $seq_{1,h_c\backslash ur(hc)} = seq_{1,h_c'\backslash ur(hc)}$ and $seq_{1,h_c\backslash ur(hc)} = seq_{2,h_c'\backslash ur(hc)}$ because of $cmp(hc) = cmp(hc')$.

$\boldsymbol{IWS_{h_{c,ins}} = IWS_{h'_{c,ins}}}$ **:**
There is no new transaction in both insertions. The interrupting write set of each transaction is unchanged as there no are new rf-elements in both $h_{c,ins}$ and $h'_{c,ins}$ compared to h_c and h_c' and the order of events is identical and no events have been modified by the insertion function. As these sets are identical for h_c and h_c', the claim follows.

$\boldsymbol{MC_{h_{c,ins}} = MC_{h'_{c,ins}}}$ **:**
It holds that $tr \notin MC_{h_{c,ins}}$ and $tr \notin MC_{h'_{c,ins}}$. There are also no new (abortable) rf-elements thus both sets are identical compared to the respective sets in h_c and h_c'. As these sets are identical for h_c and h_c', the claim follows. □

Lemma 35. *Two g-histories h, h' are OP^--extension equivalent iff the following two conditions hold:*

1. $\forall (h, h_c) \in HC_h, \exists (h', h_c') \in HC_{h'} : (h, h_c) \equiv_{ext} (h', h_c')$,
2. $\forall (h', h_c') \in HC_{h'}, \exists (h, h_c) \in HC_h : (h', h_c') \equiv_{ext} (h, h_c)$.

Proof. By the definition of OP^--extension equivalency (Definition 31), we need to show that $\forall seq \in Ev^*$ either

- $h \cdot seq$ and $h' \cdot seq$ are both value opaque under OP^-
- or $h \cdot seq$ and $h' \cdot seq$ are both not value opaque under OP^-.

Consider an arbitrary sequence $seq \in Ev^*$. We show the case where $h \cdot seq$ is not value opaque and $h' \cdot seq$ is value opaque and the case where $h' \cdot seq$ is not value opaque and $h \cdot seq$ is value opaque cannot occur.

We only show one of them as the other is completely analogue.

$h \cdot$ seq is not value opaque and $h' \cdot$ seq is value opaque :

If $h' \cdot seq$ is value opaque then by Lemma 22 there exists a candidate $h'_c \in C_h$ s.t. $\exists h'_{c,ins} \in ins(h'_c, seq) : mCl(h'_{c,ins})$ is legal. Trivially, $(h', h'_c) \in HC_{h'}$ holds. Now given condition 2 of the assumptions of the lemma, it holds that $\exists (h, h_c) \in HC_h : (h', h'_c) \equiv_{ext} (h, h_c)$. Then as $\exists h'_{c,ins} \in ins(h'_c, seq) : mCl(h'_{c,ins})$ is legal, by the definition of extension equivalence for hc-pairs (Definition 41) it holds that $\exists h_{c,ins} \in ins(h_c, seq) : mCl(h_{c,ins})$ is legal. By Lemma 22, it follows that $h_c \cdot seq$ is value opaque under OP^- which is a contradiction. Thus, $h' \cdot seq$ not is value opaque.

So we have shown that under the assumptions the requirements of Definition 31 hold for h and h', meaning they are OP^--extension equivalent. □

Lemma 36. *Two g-histories h, h' are OP^--extension equivalent if $op(h) = op(h')$.*

Proof. By Definition 51, it holds that $\{cmp(hc) \mid hc \in HC_h\} = \{cmp(hc') \mid hc' \in HC'_h\}$.

1. $\forall (h, h_c) \in HC_h, \exists (h', h'_c) \in HC_{h'} : cmp(h, h_c) = cmp(h', h'_c)$,
2. $\forall (h', h'_c) \in HC_{h'}, \exists (h, h_c) \in HC_h : cmp(h, h_c) = cmp(h', h'_c)$.

by Lemma 34 it follows that

1. $\forall (h, h_c) \in HC_h, \exists (h', h'_c) \in HC_{h'} : (h, h_c) \equiv_{ext} (h', h'_c)$,

2. $\forall (h', h'_c) \in HC_{h'}, \exists (h, h_c) \in HC_h : (h', h'_c) \equiv_{ext} (h, h_c)$.

By Lemma 35, this means that h and h' are OP^--extension equivalent

$\square$

Theorem 5. *Let I be an implementation automaton. Then I only produces g-histories opaque under OP^- iff $L(E(I)) = \emptyset$.*

Proof. Let $I = (Q, \delta, q_0, F)$ be an implementation automaton and its OP^--automaton $E(I)$ be $(Q_E, \delta_E, q_{0,E}, F_E)$. We show that $E(I)$ is a deterministic finite automaton in Proposition 23.

I only produces g-histories opaque under $\boldsymbol{OP^-} \to \mathbf{L(E(I)) = \emptyset}$:
We show the contraposition. Let the accepted word be $h = ev_0 \dots ev_n$ then there exists a run of $E(I)$

$$(q_0, op(\epsilon)) \dots (q_{n+1}, op(h)), \text{ s.t.} (q_{n+1}, op(h)) \in F_E.$$

It must hold that $op(h) \in OP^-_{\emptyset\ T, Var}$, and thus there exists no consistent compressed hc-pair in $op(h)$. As shown in Lemma 38, if a compressed hc-pair is not consistent, then all of its uncompressed versions are also not consistent. This means that there exists no consistent hc-pair for h, meaning its not opaque under OP^-.
It must also hold that $q_{n+1} \in F$ and for all indices $i < n+1$ that

$$((q_i, op(ev_0 \dots ev_{i-1})), ev_i, (q_{i+1}, op(ev_0 \dots ev_i))) \in \delta_E.$$

Thus, for the word h in I the run $q_0 \dots q_{n+1}$ exist s.t. for all indices $i < n+1$ it holds that $(q_i, ev_i, q_{i+1}) \in \delta$. As $q_{n+1} \in F$, this is an accepting run meaning I is not opaque under OP^-.

$\mathbf{L(E(I)) = \emptyset} \to$ **I only produces g-histories opaque under $\boldsymbol{OP^-}$:**
We show the contraposition. Let $h = ev_0 \dots ev_n$ be a g-history that is not

opaque under OP^- and accepted by I. Then, in I a run $q_0 \dots q_{n+1}$ exist s.t. for all indices $i < n+1$ it holds that $(q_i, ev_i, q_{i+1}) \in \delta$ and $q_{n+1} \in F$ holds. Consider the following sequence of states of Q_E:

$$(q_0, op(\epsilon)) \dots (q_{n+1}, op(h))$$

s.t. for all indices $i < n+1$ $((q_i, op(ev_0 \dots ev_{i-1})), ev_i, (q_{i+1}, op(ev_0 \dots ev_i))) \in \delta_E$ holds. This sequence exists by the definition of an OP^--automaton. As h is not opaque under OP^-, for all of its candidates h_c the hc-pair (h, h_c) is not consistent, and as shown in Lemma 38 its compression $cmp(h, h_c)$ is then also not consistent. Thus, there exists no consistent compressed hc-pair in $op(h)$ thus $op(h) \in OP^-_{\emptyset\ T,Var}$. As $q_{n+1} \in F$, this means $(q_{n+1}, op(h)) \in F_E$, and thus $E(I)$ accepts h. It follows that its language is not empty.

□

Proposition 23 (OP^- construction is DFA)**.** *Let $I = (Q, \delta, q_0, F)$ be an implementation automaton and its OP^--automaton of $E(I)$ be $(Q_E, \delta_E, q_{0,E}, F_E)$. Then $E(I)$ is a deterministic finite automaton.*

Proof. We assume I to be an DFA. We show that $E(I)$ is a finite automaton and then that it is a deterministic automaton.

$\mathbf{Q_E}$ is finite:
Note that Q_E equals $Q \times OP^-_{T,var}$. As I is a DFA, Q is finite. As proven in Lemma 33, there is only a finite amount of compressed hc-pairs for a given T and Var thus $OP^-_{T,Var}$ is finite as well as each element in it is a set of hc-pairs. Combining both facts means Q_E is a finite set.

$\boldsymbol{\delta}_\mathbf{E}$ is deterministic:
Consider a given state/OP^--data pair $(q, op) \in Q_E$, $q \in Q$ and $op \in OP^-_{T,var}$ and an arbitrary g-event ev. We show there is only exactly one pair of $q' \in Q$

and OP^{-}-data op' that fulfils the condition for a transition, which is

$$(q, ev, q') \in \delta \text{ and } \exists h \in \mathcal{H} : op(h) = op \wedge op(h \cdot ev) = op'.$$

We do this by a proof via contradiction. Assume a different second pair (q'', op'') fulfils the condition for the transition. So, at least $q'' \neq q'$ or $op'' \neq op'$ must hold. Trivially, as Q is a DFA q'' equals q'. So $op'' \neq op'$ must hold. Thus, there must exist a g-history $h' \neq h$ s.t. $op(h') = op \wedge op(h' \cdot ev) = op''$. This implies $op(h) = op(h')$ is true. As we show in the proof of Lemma 37 in this appendix,

$$\forall h, h' \in \mathcal{H} : op(h) = op \wedge op(h') = op \rightarrow op(h \cdot ev) = op(h' \cdot ev)$$

holds. This implies that $op(h' \cdot ev) = op(h \cdot ev) = op$, which is a contradiction to the assumption. Thus, there exists only one pair (q', op') that fulfils the condition for a transition. ☐

Lemma 37. *Given two arbitrary g-histories h and h', it holds that*

$$\forall seq \in Ev^{*} : op(h) = op(h') \rightarrow op(h \cdot seq) = op(h' \cdot seq).$$

Proof. Let ev be an arbitrary g-event. By definition, it holds that

$$op(h \cdot ev) = \{cmp(h \cdot ev, h_{c+}) \mid h_{c+} \in C_{h \cdot ev}\}.$$

Using Proposition 4 it follows that

$$op(h \cdot ev) = \{cmp(h \cdot ev, h_{c+}) \mid h_{c+} \in \bigcup_{h_c \in C_h} ins(h_c, ev)\}.$$

The same holds for $h' \cdot ev$.

$$op(h' \cdot ev) = \{cmp(h' \cdot ev, h'_{c+}) \mid h'_{c+} \in \bigcup_{h'_c \in C_{h'}} ins(h'_c, ev)\}.$$

$\mathbf{op}(\boldsymbol{h \cdot ev}) \subseteq \mathbf{op}(\boldsymbol{h' \cdot ev})$ **:**
Let (h, h_{c+}) be an arbitrary element in $op(h \cdot ev)$. Consider a candidate $h_c \in C_h$ s.t. $h_{c+} \in ins(h_c, ev)$, which exists by Proposition 4. Given that $op(h) = op(h')$, there exists $h'_c \in C_{h'}$ s.t. $cmp(h, h_c) = cmp(h', h'_c)$. As we have shown in the proof of Lemma 34 in this appendix, it then holds that

$$\exists h'_{c,ins} \in ins(h'_c, ev) : cmp(h \cdot ev, h_{c,ins}) = cmp(h' \cdot ev, h'_{c,ins}).$$

Let $h'_{c,ins}$ be one of these candidate extensions. As $h'_{c,ins} \in \bigcup_{h'_c \in C_{h'}} ins(h'_c, ev)$, it holds that $cmp(h' \cdot ev, h'_{c,ins}) \in op(h' \cdot ev)$.

$\mathbf{op}(\boldsymbol{h' \cdot ev}) \subseteq \mathbf{op}(\boldsymbol{h \cdot ev})$ **:**
This proof is analogue to the previous case.

Combining both cases shows that $op(h \cdot ev) = op(h' \cdot ev)$ holds. By induction, it follows that

$$\forall seq \in Ev^* : op(h) = op(h') \rightarrow op(h \cdot seq) = op(h' \cdot seq).$$

□