Reasoning about Two-Phase Locking Concurrency Control

Supervisor:
Prof. Philippa Gardner

Author:
David Harver Pollak

Second Marker:
Dr. Mark Wheelhouse

June 19, 2017

Abstract

Transactions are the main units of execution in database systems that employ concurrency to achieve better performance. They are managed by protocols that are able to guarantee different degrees of consistency based on the application's requirements. One of the strong consistency properties is serializability, which requires the outcome of any schedule of operations, coming from concurrent transactions, to be equivalent to a serial one in terms of the effects on the database. Surprisingly, modern program logics that enable the verification of fine-grained concurrency have not yet been applied to the context of database transactions.

We present a program logic for serializable transactions that are able to manipulate a shared storage. The logic is proven to be sound with respect to operational semantics that treat transactions as real atomic blocks. Nevertheless, we are able to model the atomicity of transactions even if it is only apparent, and concurrent interleavings do actually occur. We show this by providing the first application of our logic in terms of the *Two-phase locking* (2PL) protocol which ensures serializability. We first define a formal model and semantics that fully express the two-phase locking behaviours as part of a generic software system, then we show the equivalence between its operational semantics and the truly atomic ones. This result creates the necessary link to enable client reasoning in the 2PL setting through our program logic for transactions.

Acknowledgements

I would like to thank my supervisor, Philippa Gardner, for inspiring and guiding me through the course of this project. The regular meetings we had, always led to new and interesting questions around the topic and were filled with helpful feedback. A particular thanks goes to her research team members Shale Xiong, Julian Sutherland, Azalea Raad and Andrea Cerone, in no particular order. They were true companions in this journey and offered countless hours of their time to support my work. Their insights and suggestions have been of invaluable importance and I am extremely grateful for this.

It goes without saying that I owe everything to my parents Tim and Cristina and I am thankful for all they have given me. Together with my sister Desideria, they have supported and unconditionally loved me throughout my four years at Imperial.

Finally, this section would not be complete if I did not express my eternal gratitude to my hometown football club, A.C. Fiorentina, for giving me strong vital emotions every single week of my life.

"Non exiguum temporis habemus, sed multum perdimus. Satis longa vita et in maximarum rerum consummationem large data est, si tota bene conlocaretur; sed ubi per luxum ac neglegentiam diffluit, ubi nulli bonae rei inpenditur, ultim demum necessitate cogente quam ire non intelleximus transisse sentimus. Ita est: non accipimus brevem vitam sed facimus nec inopes eiussed prodigi sumus."

— *Seneca, De Brevitate Vitae*

Contents

1 Introduction **1**
1.1 Contributions . . . 2

2 Background **3**
2.1 History of Concurrent Reasoning . . . 3
2.2 Modern Concurrent Reasoning . . . 7
2.3 Database Theory . . . 17

3 Motivating example **25**

4 The mCAP Program Logic **29**
4.1 Partial Commutative Monoid . . . 29
4.2 Worlds . . . 30
4.3 Assertions . . . 32
4.4 Environment Semantics . . . 35
4.5 Programming Language . . . 38
4.6 Proof System . . . 39
4.7 Semantics . . . 41
4.8 Soundness . . . 43

5 A Logic for Serializable Transactions **46**
5.1 Model . . . 46
5.2 Assertions . . . 48
5.3 Semantics . . . 49

6 Two-Phase Locking Semantics **50**
6.1 Model . . . 50
6.2 Operational Semantics . . . 51

7 Two-Phase Locking is Serializable **58**
7.1 Serializability . . . 58
7.2 Semantics Equivalence . . . 66
7.3 Soundness . . . 87

8 Evaluation & Comparisons **89**

9 Conclusions **92**
9.1 Future Work . . . 92

Appendices

Appendix A Auxiliary Lemmata for mCAP **97**

Appendix B Auxiliary Lemmata for Serializability 104

Appendix C Auxiliary Lemmata for Equivalence 109

1. Introduction

Modern database systems make heavy use of concurrency to increase performance and therefore to support large scale operations. Programmers access the underlying data through transactions, which are self-contained programs describing a single unit of work. To manage their concurrent executions, database systems employ various techniques depending on the degree of consistency[1] that needs to be guaranteed. This choice has an impact on the performance of the system too. A high-performance database system has to inevitably weaken its consistency model at the cost of allowing the occurrence of anomalous effects. For this reason, in practice, many commercial databases provide a relatively stronger consistency guarantee to release the developers' burden when writing applications that cannot afford such behvaiours.

One of such models is shaped around the idea of serializable executions. Serializability is a strong consistency property that requires the outcome of any schedule of operations, which are grouped by transactions, to be equivalent to a serial one, where transactions are run one after the other. A typical way to implement this consistency model is through pessimistic concurrency control. *Two-phase locking* is a popular concurrency control protocol in this group. It is a blocking approach that works at the granularity of single database entries by assigning a synchronization structure to each of them, so that it can guarantee serializability without losing too much performance.

Out of the many works on reasoning about transactions and the way they are managed, few of them are compositional and work at a program logic level. On the other hand, the area of formal reasoning about shared memory concurrency has seen a noticeable development towards logic frameworks that can verify fine-grained concurrency in a compositional way. In recent years, modern concurrent program logics, based on separation logic, have introduced two fundamental notions of abstraction. The first one, namely data abstraction, makes it possible to give abstract specifications by hiding implementation details, something that is proven to be very useful for compositional reasoning in a client-module setting. Furthermore, time abstraction, also called atomicity, is a property of operations by which they appear to happen at a single and discrete moment in time.

In the world of databases, and more specifically in the context of serializable models, transactions are often coupled with the idea of atomicity, which is very similar to the atomicity in shared memory: to the programmer's eyes, transactions are seen as units of work that get executed in one step. Thus, there is a clear connection between program logics for shared memory concurrency and database transactions. This is why we formulate a program logic to reason about serializable transactions, which we believe it is the first of its kind. The focus is then shifted to an application of our logic to the setting of two-phase locking, where we prove that its semantics conform with the required atomicity. This enables users of the framework to prove the correctness of programs running in a system that adopts a flavour of this concurrency control protocol, by only having to reason atomically about blocks of code, without the complexity of concurrent interleavings.

[1] Isolation level is usually the terminology adopted for consistency in database systems.

1.1 Contributions

The main technical contributions of this project are listed below, with references to the relevant sections where they are further discussed.

- **mCAP** (Section 4) We reformulate and extend a program logic for concurrent programs, namely CAP [12], in order to remove some of its constraints, which are hardcoded into the logic, and enable a more flexible reasoning. In fact, we change the underlying model to parametrize both the representation of machine states and of action capabilities. On top of this, we provide a new and cleaner structure for the action model that does not explicitly use interference assertions. We also considerably modify the way environment interference is modelled through the rely/guarantee relations. This is done with the goal of allowing both a thread, and the environment, to perform multiple shared region updates in one step. It follows that the repartitioning operator also has a new and extended behaviour. At the level of the programming language, we leave elementary atomic commands as a parameter to the user of the logic.

- **Logic for Transactions** (Section 5) The mCAP program logic framework is instantiated into a logic for our particular needs of reasoning about serializable transactions. All of mCAP's parameters are defined in order to model a system where transactions can access a generic indexed storage and carry a local variable stack.

- **2PL Model** (Section 6.1) The logic is then applied to the context of two-phase locking. Its details are analysed and ported to a formal model that, through its constructs and structures, is able to describe a transactional software system that uses the protocol and exhibits all of the required behaviours. The main novelties introduced are related to the way we globally manage information related to locking and track the state of running transactions.

- **Operational Semantics** (Section 6.2, 7.1) We use the constructs introduced in the 2PL model in order to shape a set of operational semantic rules that formally express the way the protocol acts at runtime. They are provided in a small-step fashion to enable the actual interleaving between concurrent transactions. Locking is implicit and does not occur as part of a language command. Instead, we take a nondeterministic approach to locking, and for this reason the semantics can model any particular pattern of lock acquisitions and releases, as long as it complies with the two-phase rule. The mentioned rules are able to reduce programs while labelling every step of the execution with the appropriate transaction or system operation. We group all such consecutive labels into a trace, which is the main structure used to construct a proof of serializability of the operational semantics as a whole.

- **Semantics Equivalence** (Section 7.2) In order to allow mCAP style reasoning on programs running under 2PL, we are required to prove its soundness with respect to the operational semantics we introduced earlier. This effort is done in two steps, as we first prove the soundness in terms of a baseline operational semantics, which does not allow any interleaving between concurrent transactions by reducing them all at once: it effectively runs transactions atomically, in complete isolation. Then, we show that any reduction that reaches a terminal state in the 2PL semantics can be replicated by the atomic one. The latter proof requires a large number of intermediate results and structures which are formally defined, and whose specific properties are proven to be sound.

2. Background

The introductory material we provide in this section sets up the scene for the work that will be later explained. It follows a path that goes from the early days of program reasoning all the way to modern techniques that allow to deal with concurrency. On top of this, we look at database theory from a transactional perspective, analyzing concurrency control paradigms and consistency models.

2.1 History of Concurrent Reasoning

Before entering the world of modern program logics for concurrency, we study the building blocks that serve as its foundation. First and foremost, the work done by Tony Hoare provides us with the starting point for reasoning about sequential programs using an axiomatic approach. The latter is expanded in order to cope with concurrency, and later enhanced through a methodology allowing the abstraction of effects that concurrent threads may have on the program state.

2.1.1 Hoare Logic

As a starting point in proving correctness, or generally properties of computer programs, Hoare [14] explains how to use deductive reasoning in applying inference rules to a set of predefined language axioms. He specifies the use of what was later on referred to as a "Hoare triple" of the shape $\{P\}\ C\ \{Q\}$ where P and Q are first-order assertions while C is the program executed. P is called the precondition of C while Q is its postcondition. The triple thus describes that whenever the assertion P holds before C runs and at some point terminates, then Q will hold on its completion. The kind of logic adopted, allows to reason about relatively simple programs involving integers and local variables living in the *stack* of a function. Classic proofs performed using Hoare logic take the form of a proof tree with axioms as leaves and rules as intermediate steps. Such notation has the downside that complete proofs tend to explode in size. For this reason, proof sketches are utilized. They represent the holding assertions before and after every command that appears in the analyzed code. We can see an example of such in Figure 2.1.

```
{⊤}
function abs(x) {
  r := x;
  {x = r}
  if (x < 0) {
    {x < 0 ∧ x = r}
    r := −x;
    {x < 0 ∧ r = −x}
    {(x < 0 ∧ r = −x) ∨ (x ≥ 0 ∧ r = x)}
  } else {
    {x ≥ 0 ∧ x = r}
    skip;
    {x ≥ 0 ∧ x = r}
    {(x < 0 ∧ r = −x) ∨ (x ≥ 0 ∧ r = x)}
  }
  {(x < 0 ∧ r = −x) ∨ (x ≥ 0 ∧ r = x)}
  return r;
}
{(x < 0 ∧ ret = −x) ∨ (x ≥ 0 ∧ ret = x)}
```

Figure 2.1: A proof sketch of the absolute value function using Hoare logic.

The abs(x) function simply returns the absolute value of the input argument x. We notice that the procedure's precondition is $\top$ (*true*), which means that there is no particular assertion which needs to hold in order for the code to perform its task. On the other hand, the post-condition states that the return value, which is referred to using a special variable named ret, will effectively be the absolute value of x. Note how the postcondition of every command in a sequence becomes the precondition of the following one, respecting the rule specified in [14].

2.1.2 Owicki-Gries

In a concurrent setting, the first approach that allowed reasoning of programs was presented in [20] and is referred to as the "Owicki-Gries" method from the name of its authors. Its main contribution is the parallel composition rule.

$$\frac{\vdash \{P_1\}\ C_1\ \{Q_1\} \quad \vdash \{P_2\}\ C_2\ \{Q_2\} \quad \text{interference-free}}{\vdash \{P_1 \wedge P_2\}\ C_1 \parallel C_2\ \{Q_1 \wedge Q_2\}}\ \textsc{Owicki-Gries}$$

The rule states that we perform a regular sequential proof for each of the threads that are composed together and the resulting postcondition will be the conjunction of all the components' ones. This holds as long as the proofs of the individual thread runs are not interfering with each other. In other words, every intermediate assertion between atomic commands in the sequential proof of C_1 must be kept valid by the actions of C_2 and vice versa [26].

$$\dfrac{\dfrac{(\dagger) \qquad\qquad (\spadesuit)}{\{x = 0 \land x = 0\}\ x := x+1 \mid\mid x := x+2\ \{(x = 1 \lor x = 3) \land (x = 2 \lor x = 3)\}}\ \textsc{Owicki-Gries}}{\{x = 0\}\ x := x+1 \mid\mid x := x+2\ \{x = 3\}}\ \textsc{Cons}$$

$$(\dagger) = \dfrac{\dfrac{\dots}{\{x = 0 \lor x = 2\}\ x := x+1\ \{x = 1 \lor x = 3\}}\ \textsc{Assign}}{\{x = 0\}\ x := x+1\ \{x = 1 \lor x = 3\}}\ \textsc{Cons}$$

$$(\spadesuit) = \dfrac{\dfrac{\dots}{\{x = 0 \lor x = 1\}\ x := x+2\ \{x = 2 \lor x = 3\}}\ \textsc{Assign}}{\{x = 0\}\ x := x+2\ \{x = 2 \lor x = 3\}}\ \textsc{Cons}$$

$$\dfrac{P \Rightarrow P' \quad \{P'\}\ C\ \{Q'\} \quad Q' \Rightarrow Q}{\{P\}\ C\ \{Q\}}\ \textsc{Cons}$$

Figure 2.2: Partial proof tree of a parallel program incrementing variable x which also uses the Cons rule (provided).

The method's main burden is the interference-free constraint which makes non-trivial commands very complicated to prove. As a consequence, most specifications that satisfy the requirement will be too weak to prove any interesting properties about the concurrent code. This is the reason why such a method is only able to give stronger specifications using auxiliary or ghost variables whose task is to keep track of past program states. In addition, the actual code needs to be augmented with statements that refer to the ghost variables themselves. Any modular use of proofs that involve auxiliary variables would require new specifications, since the number of variables needed to express strong specifications would increase based on the way the module is used (e.g. the number of threads running). This becomes clear by looking at an example of two threads incrementing the same counter concurrently. We first define the counter's methods and then show a summarized proof of clients' usage through ghost variables [10].

```
function makeCounter() {
  c := alloc(1);
  [c] := 0;
  return c;
}

function increment(c) {
  do {
    n := [c];
    b := CAS(c, n, n + 1);
  } while (b = 0);
  return n;
}
```

```
c := makeCounter(); inc₁ := 0; inc₂ := 0;
{c ↦ 0 ∧ inc₁ = 0 ∧ inc₂ = 0}

{c ↦ inc₁ + inc₂ ∧ inc₁ = 0}          ||  {c ↦ inc₁ + inc₂ ∧ inc₂ = 0}
do {                                   ||  do {
  n := [c];                            ||    n' := [c];
  ⟨b := CAS(c, n, n + 1);              ||    ⟨b' := CAS(c, n', n' + 1);
    if (b) {inc₁++}⟩                   ||      if (b') {inc₂++}⟩
} while (b = 0);                       ||  } while (b' = 0);
{c ↦ inc₁ + inc₂ ∧ inc₁ = 1}          ||  {c ↦ inc₁ + inc₂ ∧ inc₂ = 1}

{c ↦ inc₁ + inc₂ ∧ inc₁ = 1 ∧ inc₂ = 1}
{c ↦ 2}
```

As we can see, the parts of code in green are auxiliary, i.e. they are not part of the executed code but are there to aid the proof. We use them to build the invariant that each of the two concurrent commands will use to prove its postcondition. In fact, at any point in the concurrent execution, the counter value will be the sum of the two auxiliary variables. Note that the increments to the ghost variables $\texttt{inc}_1$ and $\texttt{inc}_2$ must happen atomically with the respective `CAS` commands and this is the reason why they are included in an atomic context $\langle - \rangle$.

If we now were to prove three separate threads incrementing the same counter in parallel, we would have to redo the proof, adding a new auxiliary variable $\texttt{inc}_3$ and changing the invariants in all of the other bodies. The solution proposed is therefore not compositional. This implies that whenever we add a new method to a verified module which interacts with some shared state, we need to cross check every new command with all the others already present.

2.1.3 Rely/Guarantee

Jones [15] introduces a way of explicitly stating the interference coming from the environment as part of concurrent composition of code. This replaces the implicit and tedious side condition in the parallel rule of the "Owicki-Gries" method. The result is rely/guarantee reasoning which is a compositional method. The specifications that arise from the use of such technique, adopt binary relations to express how the state might change as a result of the running thread's or the environment's actions.

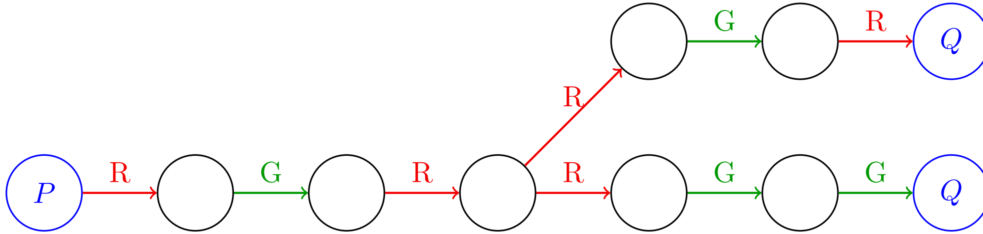


Figure 2.3: An interleaving of environment and thread actions abstracted using rely/guarantee relations.

A predicate P that refers to the structure of a single state, describes a set of possible states, while binary relations represent a set of transitions that can happen in the system [26]. The latter are predicates of the form $P_a(\vec{\sigma}, \sigma)$ that relate the state before action a to the one right after. Given a single predicate we can define the two-state relation by placing no constraint on the old state, $P(\sigma) \triangleq P(\vec{\sigma}, \sigma)$ or on the other hand by not restricting the new state σ, $P(\vec{\sigma}) \triangleq P(\vec{\sigma}, \sigma)$. State relations can be sequentially composed in order to reflect the effect of actual program commands as follows $(P; Q)(\vec{\sigma}, \sigma) \triangleq \exists \alpha. P(\vec{\sigma}, \alpha) \wedge Q(\alpha, \sigma)$. We can now define that a binary relation P is stable under another relation Q if and only if $(P; Q) \Rightarrow P \wedge (Q; P) \Rightarrow P$ which means that whenever an action in Q is done before or after a transition in P, this must not invalidate P itself.

Every specification will now take the form $R, G \vdash \{P\}\ C\ \{Q\}$ and include the standard pre and postconditions, plus R and G, the rely and guarantee relations. The rely relation models all actions that the environment can perform while the guarantee one describes what the thread executing C has the ability to do. In order for the proof to be valid, G needs to be stable under the interference of R. Stability is only explicitly checked at the atomic command level [26] as the following inference rule describes.

$$\frac{\mathsf{ID}, \top \vdash \{P\}\ C\ \{Q\} \quad (P, Q) \in G \quad \mathsf{stable}(P, R) \quad \mathsf{stable}(Q, R)}{R, G \vdash \{P\}\ \langle C \rangle\ \{Q\}}\ \textsc{RG-Atomic}$$

Given that C executes atomically, there can be no interference from the environment, so $R \equiv \mathsf{ID}$, the identity relation, meaning that the state will be kept as it is. On the other hand,

the guarantee relation $G \equiv \top$, allows any operation, although, as part of the premiss, we require the state pair (P, Q) to be part of the original guarantee relation. The hard requirement is the explicit check of stability of P and Q under relation R.

$$\frac{R \cup G_2, G_1 \vdash \{P_1\}\ C_1\ \{Q_1\} \quad R \cup G_1, G_2 \vdash \{P_2\}\ C_2\ \{Q_2\}}{R, G_1 \cup G_2 \vdash \{P_1 \land P_2\}\ C_1 \mid\mid C_2\ \{Q_1 \land Q_2\}} \text{ RG-PAR}$$

Commands are composed in parallel using the RG-PAR rule that makes the guarantee of the first command be part of the rely relation of the second one and vice-versa. This makes sure that we model all interference coming from concurrent threads.

2.2 Modern Concurrent Reasoning

The explanation of logic reasoning techniques continues by considering approaches that are currently used in concurrent systems. These frameworks allow to tackle the verification of heap-manipulating programs thanks to the advances brought by separation logic. Key results in this area have been achieved through the use of abstraction in specifications, which allows to use program logics in a client-module setting, and the formalisation of atomicity, a fundamental concept in concurrency where an operation is seen as happening in one single step.

2.2.1 Concurrent Separation Logic

In order to reason about heap manipulating programs, separation logic was introduced in [24] as a program logic which revolves around the concept of resource owning. The program state does not just involve values local to a method, but also ones that live in the shared *heap* and that variables refer to through pointers. In terms of assertions, the main additions to Hoare's constructs are explained in the following list.

- The separating conjunction $P * Q$ asserts that the state can be divided into two disjoint parts, one of which satisfies predicate P and the other one Q.
- $E_1 \mapsto E_2$ asserts that there exists a memory cell whose address can be obtained by evaluating E_1 and whose value is E_2.
- The empty assertion *emp* states that the *heap* must contain no allocated cells.

The separating conjunction empowers the program logic to have a frame rule which allows to prove a specification by temporarily not considering additional resources in the heap that are not accessed as part of the command we are verifying.

$$\frac{\vdash \{P\}\ C\ \{Q\} \quad \mathsf{mod}(C) \cap \mathsf{fv}(R) \equiv \emptyset}{\vdash \{P * R\}\ C\ \{Q * R\}}$$

The requirement for it to work, is that no variable which is modified as part of running command C can be free in predicate R. Another way of looking at the rule is that if we are able to prove a triple, then adding an arbitrary resource, which is not touched by the command, will not invalidate our original proof. In later work, O'Hearn [19] included support for disjoint concurrency through a new rule.

$$\frac{\vdash \{P_1\}\ C_1\ \{Q_1\} \quad \vdash \{P_2\}\ C_2\ \{Q_2\} \quad \mathsf{nc}(C_1, C_2, P_2, Q_2) \quad \mathsf{nc}(C_2, C_1, P_1, Q_1)}{\vdash \{P_1 * P_2\}\ C_1 \mid\mid C_2\ \{Q_1 * Q_2\}}$$

$$\mathsf{nc}(C, C', P, Q) \triangleq \mathsf{mod}(C) \cap (\mathsf{fv}(C') \cup \mathsf{fv}(P) \cup \mathsf{fv}(Q)) \equiv \emptyset$$

The rule states that as long as two threads only need access to disjoint parts of memory in order to execute, then they can safely be composed in parallel. The disjointness requirement appears as a side-condition to the rule, expressed through the non conflict predicate `nc`.

When dealing with process interaction, programs need to be syntactically structured in a new way. This novelty is introduced in order to declare and model the shared resources which are accessible to concurrent commands in a program. On top this, a new command is also introduced to allow a process to access shared resources through a conditional critical region [19] [5].

$$\begin{array}{l|l} \textit{init}; & \\ \texttt{resource}\ r_1(\text{vars}), \ldots r_m(\text{vars}) & C ::= \cdots \mid \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C\ \texttt{endwith} \\ C_1 \| \cdots \| C_n & \end{array}$$

Shared resources are uniquely named and have a list of input variables that they manage. A process can interact with them using the "`with`" command, representing a unit of mutual exclusion. Access is only granted if no other region on r is currently executing and the boolean condition B (which is heap independent) evaluates to true. In all other cases the process must wait for the described conditions to become true.

Each declared resource r_i is assigned a resource invariant in the shape of a formula RI_{r_i}. The latter needs to be precise and any assignment $x := \cdots$, where x is a free variable in RI_{r_i}, must occur within a conditional critical region for r_i. The proof rules are enhanced to be able to cover the new syntactic constructs. First, all of the resource invariants must be separately established by the initialization list of commands *init*, together with a part of the state, i.e. P', that can be accessed outside of critical regions. The latter is the only assertion used to prove that Q holds on conclusion of the parallel execution of commands $C_1 \| \cdots \| C_n$. All of the invariants are then established again as part of the rule's conclusion.

$$\frac{\vdash \{P\}\ \textit{init}\ \{RI_{r_1} * \cdots * RI_{r_m} * P'\} \quad \vdash \{P'\}\ C_1 \| \cdots \| C_n\ \{Q\}}{\vdash \{P\} \begin{array}{l} \textit{init}; \\ \texttt{resource}\ r_1(\text{vars}), \ldots r_m(\text{vars}) \\ C_1 \| \cdots \| C_n \end{array} \{RI_{r_1} * \cdots * RI_{r_m} * Q\}}$$

Whenever a conditional critical region command for r is encountered, and no other process is within a region for the same resource, we add the resource invariant RI_{r_i} to the process' local state. Using this approach, when a process is outside a region for r, it is effectively not able to see r's state.

$$\frac{\vdash \{(P * RI_r) \wedge B\}\ C\ \{Q * RI_r\}}{\vdash \{P\}\ \texttt{with}\ r\ \texttt{when}\ B\ \texttt{do}\ C\ \texttt{endwith}\ \{Q\}} \quad \begin{array}{l}\text{No other process modifies} \\ \text{variables free in } P \text{ or } Q\end{array}$$

It follows that the parallel composition rule does not change from the disjoint case, given that any shared resource invariant is absent from $P_1, \ldots, P_n$ and $Q_1, \ldots, Q_n$ and will only be added at the appropriate moment, within a critical region.

$$\frac{\vdash \{P_1\}\ C_1\ \{Q_1\} \quad \cdots \quad \vdash \{P_n\}\ C_n\ \{Q_n\}}{\vdash \{P_1 * \cdots * P_n\}\ C_1 \| \cdots \| C_n\ \{Q_1 * \cdots * Q_n\}}$$

As part of the presented rule, we still need to enforce that no variable free in P_i or Q_i is modified in C_j when $i \neq j$.

2.2.2 RGSep

The efforts developed as part of separation logic and rely/guarantee reasoning were brought together by RGSep [26]. Its main contribution was related to the separation between the local

state l and shared one s. In an abstract sense, l and s are members of a separation algebra $(M, \odot, \mathsf{u})$ such that $l \odot s$ is defined if and only if the two are disjoint [6]. In the latter case, the entirety of the state is defined to be the union of l and s. In order to distinguish them, a new kind of assertion, the shared assertion, is introduced and often referred to as the "boxed assertion". In fact, $\boxed{P}$ expresses a generic separation logic assertion P which refers to the shared state s. The separating conjunction $*$ works in the usual way inside of the local state but behaves additively for the shared one. As a consequence we have that:

$$\boxed{P} * \boxed{Q} \Leftrightarrow \boxed{P \wedge Q}$$

Interference between parallel threads is modelled using actions of the form $P \rightsquigarrow Q$. The action meaning is that the part of the shared state where P holds will be replaced with a part that satisfies Q leaving the rest of s intact. For example, if we were describing the action that allows to increment a counter stored at address x, we would have something along the lines of $x \mapsto n \rightsquigarrow x \mapsto m \wedge m \geq n$.

RGSep requires all preconditions and postconditions of commands in a proof to be stable under interference coming from the environment. We can syntactically check for stability as follows.

$$\mathsf{stable}(S, [\![P \rightsquigarrow Q]\!]) \Leftrightarrow \models ((P -\!\!\circledast S) * Q) \Rightarrow S$$
$$\mathsf{stable}(S, (R_1 \cup R_2)^*) \Leftrightarrow \models \mathsf{stable}(S, R_1) \wedge \mathsf{stable}(S, R_2)$$

The first property states that if we take a state S and remove the part satisfying P and add one where Q holds then S will hold again. On the other hand, the second property defines that a statement is stable with respect to a set of actions when it is stable under the interference of every action in the set. Given that actions can only affect the shared state s, the stability check only needs to be performed on assertions which refer to s itself.

2.2.3 Concurrent Abstract Predicates

Concurrent Abstract Predicates (CAP) is a program logic explained in [12] which makes use of separation logic constructs to allow abstraction on shared data structures using predicates. These provide a disjointness of shared resources at the abstraction level which might not be reflected in the actual implementation. For example, if we consider a set module and formulate a predicate stating that "the set is $\{1, 2\}$", then we want to remove element 2 and be able to assert that the set is now $\{1\}$. In order to reason about the module in such a fine-grained manner, we want element 2 to be manipulated independently from the rest of the set. This kind of disjointness expressed at the level of abstraction does not need to be part of the implementation. In fact one could implement the set module using a singly linked list which requires traversing some elements before reaching the desired item.

Concurrent abstract predicates also hold information regarding what kind of interference is allowed on the shared memory from the thread and the environment. Fractional permissions are utilized in order to model the type of control a thread has on a particular shared structure [4]. On top of this, implementation code can be formally verified against high level specifications and later be *hidden* to clients that make use of it by allowing them not to refer to low-level details in their own proof derivations.

Standard separation logic is augmented with two novel assertions, the shared region and the permission one. The first takes the shape $\boxed{P}^{r}_{I(r,\vec{x})}$ and represents a part of memory, associated with an identifier r, which is shared among several threads and satisfies predicate P. The region is indivisible so that all threads accessing it always get a consistent view of it; we can enforce such behaviour as follows:

$$\boxed{P}^{r}_{I(r,\vec{x})} * \boxed{Q}^{r}_{I(r,\vec{x})} \iff \boxed{P \wedge Q}^{r}_{I(r,\vec{x})}$$

The permission assertion $[\mathrm{A}]^r_\pi$ gives a thread the possibility to perform action A on region r with fractional permission π. The latter will be a real value in $(0,1)$ when both the thread and the environment can execute A or equals to 1 in the case where the thread is the only one to be able to do so. We can combine permissions to perform the same action in the same region in the following way:

$$[\mathrm{A}]^r_{\pi_1} * [\mathrm{A}]^r_{\pi_2} \iff [\mathrm{A}]^r_{\pi_1+\pi_2}$$

Actions in CAP are similar to the ones seen in Section 2.2 and have the form $\mathrm{A} : P \rightsquigarrow Q$ where the label is followed by two assertions that describe the part of the region needed for A and the same part after the action has taken place. All possible actions for a particular region r are summarized inside $I(r, \vec{x})$. We can create a shared region from an assertion P as $P \Rrightarrow \exists r. \boxed{P}^{\,r}_{I(r,\vec{x})} * \mathsf{all}(I(r,\vec{x}))$, by including every action permission available.

We now have all the ingredients to prove the specification of a lock module in CAP [12]. The latter will provide three methods, namely `makeLock()`, `lock(x)` and `unlock(x)`. As the names suggest, the first method will allocate the necessary resources for a lock and initiate it, while the other two will acquire or release the lock at address `x`. Following these English specifications we can provide formal ones.

$$\{emp\}\ \texttt{makeLock()}\ \{\exists x. \texttt{ret} = x \wedge \mathsf{isLock}(x) * \mathsf{Locked}(x)\}$$
$$\{\mathsf{isLock(x)}\}\ \texttt{lock(x)}\ \{\mathsf{isLock(x)} * \mathsf{Locked(x)}\}$$
$$\{\mathsf{Locked(x)}\}\ \texttt{unlock(x)}\ \{emp\}$$

The actions allowed on the lock are L and U which are used to interpret the abstract predicates used in the specification.

$$\mathsf{isLock}(x) \equiv \exists r, \pi.\, [\mathrm{L}]^r_\pi * \boxed{(x \mapsto 0 * [\mathrm{U}]^r_1) \vee x \mapsto 1}^{\,r}_{I(r,x)}$$
$$\mathsf{Locked}(x) \equiv \exists r.\, [\mathrm{U}]^r_1 * \boxed{x \mapsto 1}^{\,r}_{I(r,x)}$$
$$I(r,x) \triangleq \left\{ \begin{array}{ll} \mathrm{L}: & [\mathrm{U}]^r_1 * x \mapsto 0 \rightsquigarrow x \mapsto 1 \\ \mathrm{U}: & x \mapsto 1 \rightsquigarrow [\mathrm{U}]^r_1 * x \mapsto 0 \end{array} \right\}$$

The $\mathsf{isLock}(x)$ predicate gives a thread knowledge of the existence of a lock at address x which can either be in the locked or unlocked state based on the value x points to. It also provides a permission to perform the L action to acquire the lock. Given the nature of the predicate we can state that $\mathsf{isLock}(x) \Rightarrow \mathsf{isLock}(x) * \mathsf{isLock}(x)$ holds as we are simply sharing the knowledge of x being a lock.

```
function makeLock() {
  x := alloc(1);
  [x] := 1;
  return x;
}

function lock(x) {
  do {
    ⟨b := CAS(x, 0, 1)⟩;
  } while (b = 0);
}

function unlock(x) {
  ⟨[x] := 0⟩;
}
```

Figure 2.4: A spin lock implementation using `CAS`.

On the other hand, $\mathsf{Locked}(x)$ describes that the lock at x is currently locked and has been previously acquired by the thread that holds the predicate. As the $\mathsf{Locked}(x)$ predicate gives us full permission of $[\mathrm{U}]^r_1$, we have that $\mathsf{Locked}(x) * \mathsf{Locked}(x) \Rightarrow \bot$ since a thread cannot acquire the lock twice sequentially. Action L's effect requires the lock to be unlocked and for the region to include the permission to perform U. In fact, once the action is performed, the latter permission will be removed from the region and included in the thread's local state so that no other thread can release the lock. U has instead the opposite behaviour: it requires the

lock to be acquired and once done, it will put the unlock permission back in the shared region. The specification provided will be used to verify a concrete implementation of the lock module using a spin lock whose code is given in Figure 2.4.

The `lock` method uses a popular mechanism to handle synchronization, namely compare-and-swap. CAS(a, c, v) works by atomically comparing the dereferenced value of a with c and in case of a match, a is set to point to v. In case of a successful swap, CAS returns 1, otherwise it returns 0.

$$
\begin{array}{l}
\{\mathsf{isLock}(\mathtt{x})\} \\
\texttt{function lock}(\mathtt{x})\ \{ \\
\quad \left\{\exists r, \pi.\ [\mathrm{L}]^r_\pi * \boxed{(\mathtt{x} \mapsto 0 * [\mathrm{U}]^r_1) \vee \mathtt{x} \mapsto 1}^{\,r}_{I(r,x)}\right\} \\
\quad \texttt{do}\ \{ \\
\quad\quad \langle \mathtt{b} :=\ \mathtt{CAS}(\mathtt{x}, 0, 1)\rangle; \\
\quad\quad \left\{\begin{array}{l}\exists r, \pi.\ \left([\mathrm{L}]^r_\pi * \boxed{(\mathtt{x} \mapsto 0 * [\mathrm{U}]^r_1) \vee \mathtt{x} \mapsto 1}^{\,r}_{I(r,x)} \wedge \mathtt{b} = 0\right) \\ \vee \left([\mathrm{L}]^r_\pi * [\mathrm{U}]^r_1 * \boxed{\mathtt{x} \mapsto 1}^{\,r}_{I(r,\vec{x})} \wedge \mathtt{b} = 1\right)\end{array}\right\} \\
\quad \}\ \texttt{while}\ (\mathtt{b} = 0); \\
\quad \left\{[\mathrm{L}]^r_\pi * [\mathrm{U}]^r_1 * \boxed{\mathtt{x} \mapsto 1}^{\,r}_{I(r,\vec{x})}\right\} \\
\} \\
\{\mathsf{isLock}(\mathtt{x}) * \mathsf{Locked}(\mathtt{x})\}
\end{array}
$$

Figure 2.5: The spin lock's `lock` function implementation proof.

In Figure 2.5 we give a CAP sketch proof of the `lock(x)` method which is shown to satisfy its specification. Note how the use of CAS is combined with a while loop whose condition is the outcome of the swap. This way we can be sure that once the control flow of the program exits the loop, the lock's state is updated and the L action has taken place.

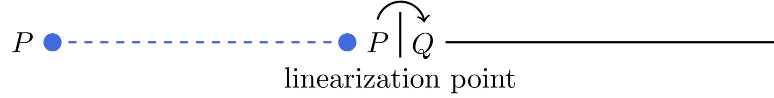
2.2.4 Abstract atomicity and Linearizability

We refer to an operation as atomic when it happens at a single discrete point in time [10]. Whenever atomic actions are performed concurrently, the actual execution will always be an interleaving of those. In general, even if a command is built from multiple operations, abstract atomicity can be obtained if the overall effect appears to be atomic.

Linearizability [13] is a correctness condition which allows methods of a concurrent module to be used by clients as atomic. All of such module operations are provided with sequential specifications that are proven to behave atomically with respect to each other. As a consequence, the moment a new method is added to the module, the linearizability proof for all methods needs to be performed. The main contribution of the approach is the fact that specifications that guarantee linearizability are an abstraction which can be directly used by clients of the module to reason without having to worry about the implementation details.

2.2.5 TaDA

TaDA [9] is a modern program logic that combines the perks of the CAP approach and of linearizability to allow modular proofs of concurrent programs. Its main novelty is the introduction of atomic specifications as a first-order construct through the use of atomic triples. These have the form $\vdash \mathbb{V}x \in X.\langle P(x)\rangle\ C\ \langle Q(x)\rangle$ and indicate that command C will atomically update the state from P to Q in a single, discrete step.



More specifically, the triple actually states that, due to the environment's interference, the variable bound by the pseudo-quantifier $\mathbb{V}$ (in this case x) can vary within the values of set X as long as it still satisfies precondition $P(x)$. Then, right at the function's linearization point, the C command will atomically bring the state to satisfy $Q(x)$. After this, the command does not give any guarantees on the validity of $Q(x)$ since it might be changed by another thread running concurrently. On the other hand, C will not change the state of x again after the linearization point but it can use it to refer to its value right before the event. The specifications give a notion of atomicity which is only maintained at the level of abstraction defined by P. In fact, observing the state at a lower level might show several distinct steps involved as part of the command.

We will explore TaDA's rules and constructs while going through a concrete program example of a stack data structure module, which supports concurrent access. The concurrent stack module will have a simple interface for clients to use that allows the creation of a new stack and the ability to push and pop items from it.

Note that the full atomic triple has the following shape, where the first part of the state is private to the thread:

$$\vdash \mathbb{V}x \in X. \langle P_p \mid P(x) \rangle \; C \quad \exists y \in Y. \langle Q_p(x,y) \mid Q(x,y) \rangle$$

This includes all assertions which are stable under the environment's intereference. Everything which is instead on the public side of the state, accepts interference as described by the guarded transitions for regions. Whenever we encounter a standard Hoare triple $\vdash \{P\} \; C \; \{Q\}$ in TaDA, we can see it as pure syntactic sugar for $\vdash \langle P \mid \top \rangle \; C \; \langle Q \mid \top \rangle$ where there is effectively no public state.

$$\{emp\} \; \texttt{makeStack()} \; \{\exists r. \, \mathsf{S}(r, \texttt{ret}, [])\}$$

$$\vdash \mathbb{V}l. \langle \mathsf{S}(r, \texttt{x}, l) \wedge \texttt{v} \neq 0 \rangle \; \texttt{push(x, v)} \; \langle \mathsf{S}(r, \texttt{x}, \texttt{v}{:}l) \rangle$$

$$\vdash \mathbb{V}l. \quad \begin{array}{c} \langle \mathsf{S}(r, \texttt{x}, l) \rangle \\ \texttt{pop(x)} \\ \langle (\mathsf{S}(r, \texttt{x}, []) \wedge \texttt{ret} = 0) \vee (\exists l'. \, \mathsf{S}(r, \texttt{x}, l') \wedge l = \texttt{ret}{:}l' \wedge \texttt{ret} \neq 0) \rangle \end{array}$$

The `makeStack()` method is given a sequential specification, since it would be unreal to use a shared structure during its creation. In a more likely setup, a single thread creates the stack and then passes its reference to other (potentially child) threads. The function's return value will be a pointer to the new data structure. Adding elements to the stack is done through the use of `push(x,v)` which puts the item `v` on top of the stack. We require an atomic specification for the method as the environment might interact with the structure while we are pushing to it. The inserted value must be different from 0 due to the fact that the return value of `pop(x)` will be 0 in the case of an empty stack. This is just a simplified convention, given the abscence of error handling in this demonstrating example. The abstract predicate $\mathsf{S}(r, x, l)$, in a style similar to CAP, indicates the presence of a stack in the shared region identified by r, at memory address x with contents l. We can formally define it as follows.

$$\begin{aligned} \mathsf{S}(r, x, l) &\triangleq \exists h, ns, ds. \, \mathbf{Stack}_r(x, h, ns, ds) * [\mathrm{G}]_r \wedge l = \mathsf{values}(ns) \\ \text{where } \mathsf{values}([]) &\triangleq [] \\ \mathsf{values}((-, v){:}l) &\triangleq v{:}\mathsf{values}(l) \end{aligned}$$

The region type $\mathbf{Stack}_r(x, h, ns, ds)$ describes the structure of a shared region in memory. In addition to that, $[\mathrm{G}]_r$ is a guard that is associated to the region as an abstract resource. It gives a thread the permission to perform action G on shared region r. Actions in TaDA are defined inside a labelled transition system that maps guards to possible updates to the region. In our case, we express a single guard which gives the ability to both push and pop from the stack. In the second case, the update actually only happen when the stack is non-empty.

$$\mathrm{G} : \forall n, v \neq 0, ns, ds.\, (ns, ds) \rightsquigarrow ((n, v){:}ns, ds)$$

$$\mathrm{G} : \forall n, v, ns, ds.\, (ns, ds) \rightsquigarrow \underline{\mathbf{if}}\ ns = (n, v){:}ns'\ \underline{\mathbf{then}}\ (ns', (n, v){:}ds)\ \underline{\mathbf{else}}\ (ns, ds)$$

We unveil the meaning of the additional arguments of $\mathbf{Stack}_r$ by providing a region interpretation for the latter.

$$I(\mathbf{Stack}_r(x, h, ns, ds)) \triangleq x \mapsto h * \mathsf{stack}(h, ns) * \circledast_{(n,v)\in ds} n \mapsto v, -$$

$$\text{where } \mathsf{stack}(h, []) \triangleq h = \mathtt{null}$$

$$\mathsf{stack}(h, (h, v){:}l) \triangleq v \neq 0 \wedge \exists z. h \mapsto v, z * \mathsf{stack}(z, l)$$

The stack's address x points to the head node, h, which is the first node of the structure that will be either `null` for an empty stack or point to a value and a subsequent node. On top of this, ns is a logical list including all active node pairs (n, v) in the stack where n is the node's address and v the value it points to. Those nodes that were at one point part of the stack but have been popped since, are called dead nodes and included in the set ds. As we can see from the transition system, every time an item is popped, it is virtually moved from ns to ds to keep track of it. This additional information, which can be seen as a ghost state of the program, is particularly helpful when proving the `pop` function implementation. In fact, if the function pops an item n which has been concurrently removed since we first read it, then it is necessary to know that it had the $n \mapsto -, -$ structure and this information is provided as part of the dead nodes assertion.

```
function makeStack() {
  x := alloc(1);
  return x;
}
```

```
function push(x, v) {
  fst := alloc(2);
  [fst] := v;

  do {
    h := [x];
    [fst + 1] := h;
    b := CAS(x, h, fst);
  } while (b = 0);
}
```

```
function pop(x) {
  r := 0;
  do {
    b := 0;
    h := [x];
    if (h = null) {
      return 0;
    } else {
      r := [h];
      next := [h + 1];
      b := CAS(x, h, next);
    }
  } while (b = 0);

  return r;
}
```

Figure 2.6: The concurrent stack WHILE language implementation with the addition of the compare-and-swap (`CAS`) command.

We provide an implementation of the three methods in Figure 2.6 followed by a full TaDA proof of the `push(x, v)` function in Figure 2.7. The proof begins by unwrapping the $\mathsf{S}(r, \mathtt{x}, l)$

abstract predicate into its definition which includes the $\mathbf{Stack}_r$ region predicate and the $[\mathrm{G}]_r$ action guard. We use the latter as part of the MakeAtomic rule, in order to modify the underlying stack structure to insert a new element, `v`. This step gives us the atomicity tracking component ⧫ and the ability of treating a sequence of commands as if they happened atomically. As part of the function's body, we allocate memory for the node (`fst`) that will host the new element and begin a loop whose task is to read the stack's head node into `h` and use `CAS` to atomically set it to `fst` only when `h` is equivalent to [`x`] (meaning that the value of the stack's head has not changed since we first read it into `h`). We prove all of this using the following inference rules:

- Existential in order to get rid of the existential quantifiers on logical variables h, ns, ds and to be able to apply the next rules.
- The AWeakening rule to turn the logical variables h, ns, ds into pseudo-quantified ones (bound by $\mathbb{V}$) and to prove an atomic triple referring to the update that follows.
- UpdateRegion consumes the atomicity tracking component to try and update the shared region $\mathbf{Stack}_r$ by first opening it up and revealing its interpretation. Later, the update can either succeed or fail, as described by the rule's postcondition that includes a disjunction.
- We Frame out all assertions only leaving $\mathtt{x} \mapsto h$ since this is all we need to conclude the proof. In fact, the `CAS(x, h, fst)` command does not modify any other variables a part from `x`.
- The CAS rule finally allows us to condition the actual update on the boolean return value, `b` which is used inside the following postconditions to understand whether the swap has happened or not.

$\mathbb{V}l.$

$\langle \mathsf{S}(r, \mathtt{x}, l) \wedge \mathtt{v} \neq 0 \rangle$

`function push(x, v) {`

$\quad\langle \mathsf{S}(r, \mathtt{x}, l) \wedge \mathtt{v} \neq 0 \rangle$

[abstract]

$\quad\quad\langle \exists h, ns, ds.\, \mathbf{Stack}_r(\mathtt{x}, h, ns, ds) * [\mathrm{G}]_r \wedge l = \mathsf{values}(ns) \wedge \mathtt{v} \neq 0 \rangle$

[atomic exists on ns, ds]

$\quad\quad\quad \mathbb{V}ns, ds.$

$\quad\quad\quad\langle \exists h.\, \mathbf{Stack}_r(\mathtt{x}, h, ns, ds) * [\mathrm{G}]_r \wedge l = \mathsf{values}(ns) \wedge \mathtt{v} \neq 0 \rangle$

[make atomic]

$\quad\quad\quad\quad r : (ns, ds) \rightsquigarrow ((-, \mathtt{v}){:}ns, ds) \wedge \mathtt{v} \neq 0 \vdash$

$\quad\quad\quad\quad \{ \exists h, ns, ds.\, \mathbf{Stack}_r(\mathtt{x}, h, ns, ds) \wedge l = \mathsf{values}(ns) * r \Rrightarrow \blacklozenge \wedge \mathtt{v} \neq 0 \}$

$\quad\quad\quad\quad \mathtt{fst} := \mathtt{alloc}(2); [\mathtt{fst}] := \mathtt{1};$

$\quad\quad\quad\quad \{ \exists h, ns, ds.\, \mathbf{Stack}_r(\mathtt{x}, h, ns, ds) \wedge l = \mathsf{values}(ns) * r \Rrightarrow \blacklozenge * \mathtt{fst} \mapsto \mathtt{v}, \mathtt{null} \wedge \mathtt{v} \neq 0 \}$

$\quad\quad\quad\quad \mathtt{do}\ \{$

$\quad\quad\quad\quad\quad \{ \exists h, ns, ds, s.\, \mathbf{Stack}_r(\mathtt{x}, h, ns, ds) \wedge l = \mathsf{values}(ns) * r \Rrightarrow \blacklozenge * \mathtt{fst} \mapsto \mathtt{v}, s \wedge \mathtt{v} \neq 0 \}$

[open region]

$\quad\quad\quad\quad\quad\quad \mathbb{V}h, ns, ds.\langle \mathtt{x} \mapsto h * \mathsf{stack}(h, ns) * \circledast_{(n,v) \in ds}\, n \mapsto v, - \rangle$

[frame]

$\quad\quad\quad\quad\quad\quad\quad \mathbb{V}h.\langle \mathtt{x} \mapsto h \rangle$

$\quad\quad\quad\quad\quad\quad\quad \mathtt{h} := [\mathtt{x}];$

$\quad\quad\quad\quad\quad\quad\quad \langle \mathtt{x} \mapsto h \wedge \mathtt{h} = h \rangle$

$\quad\quad\quad\quad\quad\quad \langle \mathtt{x} \mapsto h * \mathsf{stack}(h, ns) * \circledast_{(n,v) \in ds}\, n \mapsto v, - \wedge \mathtt{h} = h \rangle$

$\quad\quad\quad\quad\quad \{ \exists h, ns, ds, s, a.\, \mathbf{Stack}_r(\mathtt{x}, h, ns, ds) \wedge l = \mathsf{values}(ns) * r \Rrightarrow \blacklozenge * \mathtt{fst} \mapsto \mathtt{v}, s \wedge \mathtt{h} = a \wedge \mathtt{v} \neq 0 \}$

$\quad\quad\quad\quad\quad [\mathtt{fst} + 1] := \mathtt{h};$

$\quad\quad\quad\quad\quad \{ \exists h, ns, ds.\, \mathbf{Stack}_r(\mathtt{x}, h, ns, ds) \wedge l = \mathsf{values}(ns) * r \Rrightarrow \blacklozenge * \mathtt{fst} \mapsto \mathtt{v}, \mathtt{h} \wedge \mathtt{v} \neq 0 \}$

[existential on h, ns, ds]

$\quad\quad\quad\quad\quad\quad \{ \mathbf{Stack}_r(\mathtt{x}, h, ns, ds) \wedge l = \mathsf{values}(ns) * r \Rrightarrow \blacklozenge * \mathtt{fst} \mapsto \mathtt{v}, \mathtt{h} \wedge \mathtt{v} \neq 0 \}$

[weakening on h, ns, ds]

$\quad\quad\quad\quad\quad\quad\quad \mathbb{V}h, ns, ds.$

$\quad\quad\quad\quad\quad\quad\quad \langle \mathbf{Stack}_r(\mathtt{x}, h, ns, ds) \wedge l = \mathsf{values}(ns) * r \Rrightarrow \blacklozenge * \mathtt{fst} \mapsto \mathtt{v}, \mathtt{h} \wedge \mathtt{v} \neq 0 \rangle$

[update region]

$\quad\quad\quad\quad\quad\quad\quad\quad \langle \mathtt{x} \mapsto h * \mathsf{stack}(h, ns) * \circledast_{(n,v) \in ds}\, n \mapsto v, - \wedge l = \mathsf{values}(ns) * \mathtt{fst} \mapsto \mathtt{v}, \mathtt{h} \wedge \mathtt{v} \neq 0 \rangle$

[frame, CAS]

$\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathbb{V}h.$

$\quad\quad\quad\quad\quad\quad\quad\quad\quad \langle \mathtt{x} \mapsto h \rangle$

$\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathtt{b} := \mathtt{CAS}(\mathtt{x}, \mathtt{h}, \mathtt{fst});$

$\quad\quad\quad\quad\quad\quad\quad\quad\quad \langle (\mathtt{b} = 0 \wedge \mathtt{x} \mapsto h \wedge \mathtt{h} \neq h) \vee (\mathtt{b} = 1 \wedge \mathtt{x} \mapsto \mathtt{fst} \wedge \mathtt{h} = h) \rangle$

$$\left\langle \begin{array}{l} (\mathtt{b} = 0 \wedge \mathtt{h} \neq h \wedge \mathtt{x} \mapsto h * \mathsf{stack}(h, ns) * \mathtt{fst} \mapsto \mathtt{v}, \mathtt{h}) \\ \vee (\mathtt{b} = 1 \wedge \mathtt{h} = h \wedge \mathtt{x} \mapsto \mathtt{fst} * \mathtt{fst} \mapsto \mathtt{v}, \mathtt{h} * \mathsf{stack}(\mathtt{h}, ns)) \\ * \circledast_{(n,v) \in ds}\, n \mapsto v, - \wedge l = \mathsf{values}(ns) \wedge \mathtt{v} \neq 0 \end{array} \right\rangle$$

$$\left\langle \begin{array}{l} l = \mathsf{values}(ns) \wedge \mathtt{v} \neq 0 \wedge ((\mathtt{b} = 0 \wedge \mathbf{Stack}_r(\mathtt{x}, h, ns, ds) * \mathtt{fst} \mapsto \mathtt{v}, \mathtt{h} * r \Rrightarrow \blacklozenge) \\ \vee (\mathtt{b} = 1 \wedge \mathbf{Stack}_r(\mathtt{x}, h, (\mathtt{fst}, \mathtt{v}){:}ns, ds)) * r \Rrightarrow ((ns, ds), ((-, \mathtt{v}){:}ns, ds))) \end{array} \right\rangle$$

$$\left\{ \begin{array}{l} ((\mathtt{b} = 0 \wedge \mathbf{Stack}_r(\mathtt{x}, h, ns, ds) * \mathtt{fst} \mapsto \mathtt{v}, \mathtt{h} * r \Rrightarrow \blacklozenge) \\ \vee (\mathtt{b} = 1 * r \Rrightarrow ((ns, ds), ((-, \mathtt{v}){:}ns, ds))) \wedge l = \mathsf{values}(ns) \wedge \mathtt{v} \neq 0 \end{array} \right\}$$

$$\left\{ \begin{array}{l} \exists h, ns, ds.\, ((\mathtt{b} = 0 \wedge \mathbf{Stack}_r(\mathtt{x}, h, ns, ds) * \mathtt{fst} \mapsto \mathtt{v}, \mathtt{h} * r \Rrightarrow \blacklozenge) \\ \vee (\mathtt{b} = 1 * r \Rrightarrow ((ns, ds), ((-, \mathtt{v}){:}ns, ds))) \wedge l = \mathsf{values}(ns) \wedge \mathtt{v} \neq 0 \end{array} \right\}$$

$\quad\quad\quad\quad \}\ \mathtt{while}(\mathtt{b} = 0);$

$\quad\quad\quad\quad \{ \exists h, ns, ds.\, l = \mathsf{values}(ns) * r \Rrightarrow ((ns, ds), ((-, \mathtt{v}){:}ns, ds)) \}$

$\quad\quad\quad \langle \exists h.\, \mathbf{Stack}_r(\mathtt{x}, h, (-, \mathtt{v}){:}ns, ds) * [\mathrm{G}]_r \wedge l = \mathsf{values}(ns) \rangle$

$\quad\quad \langle \exists h, ns, ds.\, \mathbf{Stack}_r(\mathtt{x}, h, (-, \mathtt{v}){:}ns, ds) * [\mathrm{G}]_r \wedge l = \mathsf{values}(ns) \rangle$

$\quad \langle \mathsf{S}(r, \mathtt{x}, \mathtt{v}{:}l) \rangle$

`}`

Figure 2.7: Sketch TaDA proof of the `push` concurrent stack method.

As part of the logic, there are some key inference rules that are displayed in the `push` proof. For an exhaustive list and explanation we point the reader to [9]. First of all, MakeAtomic allows to take an atomic specification and prove the abstract atomicity of a sequence of non-atomic commands where a single linearization point will appear and update the shared state. In order to do so, we must hold the guard of the appropriate action for the region. In the premiss, we obtain the atomicity tracker component in the form of an abstract resource (♦). This works like a token used to recognize whether the atomic update declared in the atomicity context ($a : x \rightsquigarrow f(x)$) has already happened or not. On a successful update, the token will be converted into the state transition described by the guarded action. As we can see from the postcondition of the premiss, the atomic update *must* happen at some point in order to satisfy the rule.

$$\frac{f : X \to Y \quad \{(x,y) \mid x \in X, y \in f(x)\} \subseteq \mathcal{T}_{\mathbf{t}}^{*} \qquad a : x \in X \rightsquigarrow f(x) \vdash \left\{\exists x \in X.\, \mathbf{t}_a(\vec{z},x) * a \Rrightarrow \blacklozenge\right\}\ C\ \left\{\exists x \in X, y \in f(x).\, a \Rrightarrow (x,y)\right\}}{\vdash \mathbb{V} x \in X. \left\langle \mathbf{t}_a(\vec{z},x) * [\mathrm{G}]_a \right\rangle\ C\ \left\langle \exists y \in f(x).\, \mathbf{t}_a(\vec{z},y) * [\mathrm{G}]_a \right\rangle}$$

MakeAtomic

The moment we need to access the contents of a shared region to unveil its interpretation, we can use one of two rules: OpenRegion and UpdateRegion. The first one allows to view the underlying content of a region but it forbids any update to its abstract state. For this reason, it neither needs an explicit guard nor a tracking component to be used. On the other hand the UpdateRegion rule is employed to perform the atomic update of the region's abstract state. It requires the update not to have happened already and we can guarantee that with the presence of the atomicity token in the precondition. Inside the postcondition we need to take into account whether the update has succeeded or not and we do that by having a disjunction on the two possible predicates that hold for the region.

$$\frac{\vdash \mathbb{V} x \in X. \left\langle I(\mathbf{t}_a(\vec{z},x)) * P(x) \right\rangle\ C\ \left\langle \begin{array}{c} \exists y \in f(x).\, I(\mathbf{t}_a(\vec{z},y)) * Q_1(x,y) \\ \vee I(\mathbf{t}_a(\vec{z},x)) * Q_2(x) \end{array} \right\rangle}{a : x \in X \rightsquigarrow f(x) \vdash \mathbb{V} x \in X. \left\langle \begin{array}{c} \mathbf{t}_a(\vec{z},x) * P(x) \\ * a \Rrightarrow \blacklozenge \end{array} \right\rangle\ C\ \left\langle \begin{array}{c} \exists y \in f(x).\, \mathbf{t}_a(\vec{z},y) \\ * Q_1(x,y) * a \Rrightarrow (x,y) \vee \\ \mathbf{t}_a(\vec{z},x) * Q_2(x) * a \Rrightarrow \blacklozenge \end{array} \right\rangle}$$

UpdateRegion

Another way to update a shared region is to apply the UseAtomic rule which grants the permission to perform such change through an explicit guard held in the precondition, in contrast with the atomicity tracking component in UpdateRegion. The use of this inference rule implies that command C acts atomically at an abstraction level which is lower than the one of region a and as a consequence, it will be atomic at the higher level as well. This way we can stack a layer of abstraction on top of another which gives a lot of flexibility in terms of writing program modules that extend or make use of other pre-verified modules.

$$\frac{f : X \to Y \quad \{(x,y) \mid x \in X, y \in f(x)\} \subseteq \mathcal{T}_{\mathbf{t}}^{*} \qquad \vdash \mathbb{V} x \in X. \left\langle I(\mathbf{t}_a(\vec{z},x)) * [\mathrm{G}]_a \right\rangle\ C\ \left\langle \exists y \in f(x).\, I(\mathbf{t}_a(\vec{z},y)) * [\mathrm{G}]_a \right\rangle}{\vdash \mathbb{V} x \in X. \left\langle \mathbf{t}_a(\vec{z},x) * [\mathrm{G}]_a \right\rangle\ C\ \left\langle \exists y \in f(x).\, \mathbf{t}_a(\vec{z},y) * [\mathrm{G}]_a \right\rangle}$$

UseAtomic

Finally, the Frame rule works in the same way as in separation logic, by adding resources to the precondition and postcondition of a command that does not modify them. As we can

notice in the `push` proof, we can also frame out predicates referring to variables bound by $\mathbb{V}$.

$$\dfrac{\mathsf{mod}(C) \cap \mathsf{vars}(R_p) \equiv \emptyset \quad \mathsf{mod}(C) \cap \mathsf{vars}(R(x)) \equiv \emptyset \\ \vdash \mathbb{V}x \in X.\big\langle P_p \,\big|\, P(x) \big\rangle \ C \quad \exists y \in Y.\big\langle Q_p(x,y) \,\big|\, Q(x,y) \big\rangle}{\vdash \mathbb{V}x \in X.\big\langle P_p * R_p \,\big|\, P(x) * R(x) \big\rangle \ C \ \big\langle Q_p(x,y) * R_p \,\big|\, Q(x,y) * R(x) \big\rangle} \ \text{FRAME}$$

2.3 Database Theory

This section shifts the focus from program logics to the main theory behind a database. This is generically abstracted from a particular instantiation, which enables us to describe its properties from a high-level point of view, without considering its implementation. We are mainly interested in the concurrent aspects of modern databases, therefore we concentrate on transactions and the way they are managed.

2.3.1 Transactions

A database is a collection of data items that hold values. A database system (*DBS*) is a tool comprised of software and hardware components that manages access to the underlying database. It allows its users to perform specific operations on the database items. Those can be summarized as being one of two kinds, reads and writes. In the following, we will use the notation $\mathsf{R}[x]$ to represent a read operation of the database item named x while $\mathsf{W}[x,v]$ to depict a write operation which sets item x's value to v. We will use the shorthand notation $\mathsf{W}[x]$ to mean that some value is written to item x. Both the mentioned kind of operations are executed atomically [3]. This means that the execution of a set of such operations appears to be sequential in terms of its constituents.

It is often the case that several operations are grouped together in order to make them behave as if they were a single unit of work. In such situations, database transactions are used: an ordered collection of operations. For example, considering a database for bank accounts, we consider a transfer of funds as a single operation from the user's point of view even if in reality the *DBS* performs multiple sequential reads and writes to achieve the goal, as shown in Figure 2.8. A transaction execution must provide guarantees on the well known **ACID** properties [25], an acronym for:

1. Atomicity: all of the transaction's operations are executed or none of them; there is no possible partial execution of a transaction.

2. Consistency: an isolated execution of a transaction maintains the consistency constraints of the database. In our bank example, the property would be maintained when money is neither created nor destroyed as part of a transfer, but simply *moved*.

3. Isolation: a transaction executes as if no other one is running in the *DBS* at the same time. More generally, a given isolation level determines the correct behaviours of transactions that run concurrently. For example, serializability implies that any two transactions running in parallel believe that the other one has either finished before they started or it will start after they finished executing.

4. Durability: once a transaction has completed its work, all of its changes are kept in the database even if failures happen in the future.

In order to maintain property (1), a transaction needs to communicate the completion of its operations or its interruption in case of failure. This is done by introducing two new types of operation, namely commit and abort, which are notated C and A respectively. In the case of

an abort, the *DBS* must roll-back the state of the database to what it was prior to the start of the transaction in order to undo the changes performed up to the failure point.

```
BEGIN
  balance := R[a]
  W[a, balance-100]
  balance := R[b]
  W[b, balance+100]
  C
END
```

```
BEGIN
  balance := R[a]
  W[a, balance-5000]
  A
END
```

Figure 2.8: Two examples of transactions on the bank database. Note how the second one aborts due to a failure (potentially insufficient funds).

Following the explanation in [3], we formally define a transaction T as the tuple $(\theta, \sqsubset)$ where θ is the set of all operations in T and $\sqsubset$ is a partial order on θ. The latter is defined as:

$$\theta \subseteq \left\{ op \mid op \in \{\mathsf{R}[x]^i, \mathsf{W}[x]^i\}, x \in \mathsf{items}(db), i \in \mathbb{N} \right\} \cup \{\mathsf{C}, \mathsf{A}\}$$

The set of operations, θ, maintains the constraint $(\mathsf{C} \in \theta \Leftrightarrow \mathsf{A} \notin \theta)$ while the ordering relation, $\sqsubset$, must satisfy the property that if the transaction contains an operation t which is either a commit or an abort, then all operations must appear in a transaction before t. Formally,

$$\forall t \in \{\mathsf{C}, \mathsf{A}\}.\, t \in \theta \Rightarrow (\forall op \in \theta.\, op \neq t \Rightarrow op \sqsubset t)$$

Also, we state that, as part of a single transaction, any two operations that involve a particular item, with at least one of them being a write, must be ordered according to $\sqsubset$:

$$\forall x \in \mathsf{items}(db), op, i, j.\, \left(op \in \left\{\mathsf{R}[x]^i, \mathsf{W}[x]^i\right\} \wedge \{op, \mathsf{W}[x]^j\} \subseteq \theta \wedge i \neq j\right) \\ \implies (\mathsf{W}[x]^j \sqsubset op \vee op \sqsubset \mathsf{W}[x]^j)$$

For the example in Figure 2.8 we would have the following transaction definition:

$$\theta \equiv \left\{\mathsf{R}[\mathsf{a}]^1, \mathsf{R}[\mathsf{b}]^2, \mathsf{W}[\mathsf{a}]^3, \mathsf{W}[\mathsf{b}]^4, \mathsf{C}\right\}$$

$$\sqsubset \equiv \{(\mathsf{R}[\mathsf{a}]^1, \mathsf{W}[\mathsf{a}]^3), (\mathsf{R}[\mathsf{b}]^2, \mathsf{W}[\mathsf{b}]^4), (\mathsf{R}[\mathsf{a}]^1, \mathsf{C}), (\mathsf{R}[\mathsf{b}]^2, \mathsf{C}), (\mathsf{W}[\mathsf{a}]^3, \mathsf{C}), (\mathsf{W}[\mathsf{b}]^4, \mathsf{C})\}$$

This gives the *DBS* the option to arbitrarily schedule the operations which are not ordered according to $\sqsubset$. In fact, under the mentioned example, one could invert the order of the two reads (or even run them in parallel) and still always get to the same final result. As we will see in section 2.3, if a history is proven to be serializable, then we can guarantee valuable properties about it.

2.3.2 Histories

Modern database systems employ concurrency in order to run multiple transactions at the same time to achieve a higher performance and better operations throughput. On the downside, running operations in parallel on shared data items can lead to race conditions and a potential loss of two of the aforementioned ACID properties, consistency and isolation.

The interleaving of operations which arises from a concurrent run of transactions, is referred to as a history. It records the order in which operations actually happen on the database and given that some of them could effectively execute in parallel, they are not totally ordered. A history H [3] is therefore a tuple $(\Theta, \dot{\sqsubset})$ such that for a set of n transactions $T_H = \{T_i\}_{i \in I}$, where $I = \{i \mid 1 \leq i \leq n\}$, we have the operations set Θ and the partial order $\dot{\sqsubset}$ such that:

$$\Theta \triangleq \bigcup_{i=1}^{n} \theta_i \;\text{ and }\; \bigcup_{i=1}^{n} \sqsubset_i \subseteq \dot{\sqsubset}$$

When inside a history, single operations are tagged with the transaction they belong to, e.g. $\mathsf{R}[x]_i^n$ refers to a read with identifier n performed on item x by transaction i. Whenever we omit the superscript and the subscript, we assume that they are implicitly existentially quantified.

Let's now define two operations as conflicting if they both access the same data item and one of them is a write; formally we describe the property as:

$$\mathsf{conflict}(op_\alpha, op_\beta)$$
$$\Longleftrightarrow$$
$$op_\alpha \neq op_\beta \wedge ((op_\alpha \in \{\mathsf{R}[x], \mathsf{W}[x]\} \wedge op_\beta = \mathsf{W}[x]) \vee (op_\alpha = \mathsf{W}[x] \wedge op_\beta \in \{\mathsf{R}[x], \mathsf{W}[x]\}))$$

We then require that all conflicting operations appearing in a history must be somehow ordered by $\dot{\sqsubset}$.

$$\forall op_\alpha, op_\beta. (\mathsf{conflict}(op_\alpha, op_\beta) \wedge \{op_\alpha, op_\beta\} \subseteq \Theta) \Rightarrow (op_\alpha \dot{\sqsubset} op_\beta \vee op_\beta \dot{\sqsubset} op_\alpha)$$

When considering a history H_{total} which is a total order over the operations of its transactions, we can refer to it simply by $\mathsf{W}[x]_i^1$, $\mathsf{W}[y]_j^1$, C_i, $\mathsf{R}[x]_j^2$, $\mathsf{W}[x]_j^3$, C_j for example.

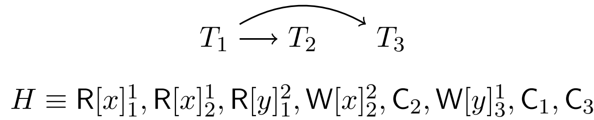
For every history $H = (\Theta, \dot{\sqsubset})$ we can compute the group of transactions which committed as part of it as the set:

$$\mathsf{committed}(H) \triangleq \{T_i \mid \mathsf{C}_i \in \Theta\}$$

This definition will aid us in describing a fundamental structure for the analysis of histories, the precedence or serialization graph. The latter allows to establish relationships between the transactions committing their operations as part a history. We can now build a history's precedence graph $G(H) = (V, E)$ where the set of vertices $V = \mathsf{committed}(H)$ includes all committed transactions in H and an edge (T_i, T_j) is in E, if transaction T_i has an operation which is ordered before one in T_j and the two are conflicting [25]. Formally, the set of edges is defined as:

$$E \triangleq \{(T_i, T_j) \mid i \neq j \wedge \exists op_\alpha, op_\beta.\, op_\alpha \in \theta_i \wedge op_\beta \in \theta_j \wedge op_\alpha \mathrel{\dot{\sqsubset}} op_\beta \wedge \mathsf{conflict}(op_\alpha, op_\beta)\}$$

In Figure 2.9 we can see a concrete instance of a history with 3 transactions and its corresponding precedence graph. Note that the graph's nodes include the three transactions since they all commit as part of H. On top of this, the edge from T_1 to T_2 is justified by pair the conflicting operations $(\mathsf{R}[x]_1^1, \mathsf{W}[x]_2^2)$, while the one from T_1 to T_2 by the pair $(\mathsf{R}[y]_1^2, \mathsf{W}[y]_3^1)$.



$$H \equiv \mathsf{R}[x]_1^1, \mathsf{R}[x]_2^1, \mathsf{R}[y]_1^2, \mathsf{W}[x]_2^2, \mathsf{C}_2, \mathsf{W}[y]_3^1, \mathsf{C}_1, \mathsf{C}_3$$

Figure 2.9: An example history with its corresponding precedence graph.

A history H_i is said to be equivalent to another history H_j [3], when they have the same set of transactions which committed as part of them, and they order conflicting operations in the same way. We formally write $H_i \equiv H_j$, and this holds if and only if:

$$\mathsf{committed}(H_i) \equiv \mathsf{committed}(H_j)$$
$$\wedge \forall op_\alpha, op_\beta. (\mathsf{conflict}(op_\alpha, op_\beta) \wedge \{op_\alpha, op_\beta\} \subseteq \Theta_i) \Rightarrow (op_\alpha \mathrel{\dot{\sqsubset}_i} op_\beta \Leftrightarrow op_\alpha \mathrel{\dot{\sqsubset}_j} op_\beta)$$

The latter condition is necessary because the result of a concurrent execution of a set of transactions only depends upon the relative ordering of conflicting operations. We describe a history as serial if all of its constituent transactions are run one after the other, without an interleaving of their operations. Such histories guarantee the highest possible database isolation. It is possible to further state that any history H is serializable if and only if it is equivalent to some serial history H'. The serializability theorem states that any history H, such that its precedence graph $G(H)$ contains no cycles, is serializable [3].

2.3.3 Anomalies and Isolation Levels

The ANSI/ISO SQL-92 standard [1] defines a set of different transaction isolation levels based on partial histories they allow as part of the possible interleavings. We will call these sequences of actions phenomena. The original standard describes 3 of such phenomena, while Berenson et al. [2] add an extra one (**P0**) for completeness.

(**P0**) A **dirty write** appears when a transaction T_i writes to a data item and, before it commits or aborts, another transaction T_j writes to the same item. If any of T_i and T_j were to abort, it is not clear what value the item should contain.
E.g. $H_{DW} \equiv \ldots \mathsf{W}[x]_i \ldots \mathsf{W}[x]_j \ldots$

(**P1**) We have the **dirty read** phenomenon in the case where transaction T_i writes to an item, then, before it commits or aborts, another transaction T_j reads that same data item. In fact if T_i would abort after T_j has already read, the latter would have retrieved a value for the item that never actually existed.
E.g. $H_{DR} \equiv \ldots \mathsf{W}[x]_i \ldots \mathsf{R}[x]_j \ldots$

(**P2**) A **non-repeatable read** happens when a transaction T_i reads a data item which is later written to by another transaction T_j that also commits. At this point, if T_i is to read the same item again, it would either discover a new value or fail due to a removal.
E.g. $H_{NR} \equiv \ldots \mathsf{R}[x]_i \ldots \mathsf{W}[x]_j \ldots \mathsf{C}_j \ldots$

(**P3**) The **phantom** phenomenon appears when transaction T_i queries all data items satisfying a particular condition γ. Transaction T_j then inserts some new items that satisfy γ. Now, if T_i were to reproduce the initial query, it would see new items appear.
E.g. $H_P \equiv \ldots \mathsf{R}[\gamma]_i \ldots \mathsf{W}[\gamma]_j \ldots \mathsf{R}[\gamma]_i \ldots$ here we abuse the transaction notation in the following way $\mathsf{R}[\gamma] \triangleq \{x \mid x \in \mathsf{items}(db) \land \gamma(x)\}$ and $\mathsf{W}[\gamma] \triangleq \exists x, v.\, \gamma(x) \land \mathsf{W}[x, v]$.

The listed definitions can be used to formulate the ANSI isolation levels as they appear in Table 2.1. The choice of how to concurrently run several transactions given by a DBS allows clients to achieve a better performance or stronger consistency depending on how strict the isolation level is. In fact, allowing some of the phenomena can lead to failing constraint checks in the database.

Table 2.1: ANSI isolation levels and the phenomena they allow [1] [2].

Isolation Level	**Dirty Write**	**Dirty Read**	**N-R Read**	**Phantom**
Read Uncommitted	×	✓	✓	✓
Read Committed	×	×	✓	✓
Repeatable Read	×	×	×	✓
Serializable	×	×	×	×

A concrete example of how the presence of the described phenomena can lead to a lack of correctness is shown in Table 2.2 where transactions run at the read committed level. Let's assume that T_i is executed by a bank's agent that wants to read what percentage of the overall funds are held by customer a and T_j is directly performed by a as a cash withdrawal from an ATM machine. The former would first read the balance of a then sum the funds of all customers (in this case a, b, c) and perform a simple division. On the other hand, T_j will first read the cash availability, then subtract \$100 and provide the money to the customer. Given the isolation level selected, a non-repeatable read appears as part of history H as T_j removes the money after T_i has read the initial balance for a and before it has computed the overall sum. Therefore

the end result ($\mathtt{b} \div \mathtt{total}$) will be incorrect and larger than the actual one. Moreover, this is a consequence to the fact that H is not serializable. We can show this by first noticing that there are two possible serial histories:

$$H_1 = \mathsf{R}[a]_1^1, \mathsf{R}[a]_1^2, \mathsf{R}[b]_1^3, \mathsf{R}[c]_1^4, \mathsf{C}_1, \mathsf{R}[a]_2^1, \mathsf{W}[a]_2^2, \mathsf{C}_2$$

$$H_2 = \mathsf{R}[a]_2^1, \mathsf{W}[a]_2^2, \mathsf{C}_2, \mathsf{R}[a]_1^1, \mathsf{R}[a]_1^2, \mathsf{R}[b]_1^3, \mathsf{R}[c]_1^4, \mathsf{C}_1$$

The conflicting operations that happen as part of the execution are the pairs $(\mathsf{R}[a]_1^1, \mathsf{W}[a]_2^2)$ and $(\mathsf{R}[a]_1^2, \mathsf{W}[a]_2^2)$. The relative order in which they appear in the histories we consider is expressed as follows:

$$H : \mathsf{R}[a]_1^1 \mathrel{\dot\sqsubset_H} \mathsf{W}[a]_2^2 \mathrel{\dot\sqsubset_H} \mathsf{R}[a]_1^2$$

$$H_1 : \mathsf{R}[a]_1^1 \mathrel{\dot\sqsubset_{H_1}} \mathsf{R}[a]_1^2 \mathrel{\dot\sqsubset_{H_1}} \mathsf{W}[a]_2^2 \qquad H_2 : \mathsf{W}[a]_2^2 \mathrel{\dot\sqsubset_{H_2}} \mathsf{R}[a]_1^1 \mathrel{\dot\sqsubset_{H_2}} \mathsf{R}[a]_1^2$$

It is clear that the particular ordering of conflicts in H is neither the same as the one in H_1, nor to the one in H_2. Therefore since H is not equivalent to any of H_1 and H_2, it is clearly not serializable. A plausible solution for this kind of issue is to change the isolation level of the DBS to repeatable read which forbids the presence of the culprit phenomenon.

Table 2.2: History H showing a concurrent run of two transactions T_i and T_j.

Time t	T_i	T_j
0	$\mathtt{b} = \mathsf{R}[a]_i^1$	$\mathtt{c} = \mathsf{R}[a]_j^1$
1	-	$\mathsf{W}[a, \mathtt{c} - 100]_j^2$
2	-	C
3	$\mathtt{total} = \mathsf{R}[a]_i^2 + \mathsf{R}[b]_i^3 + \mathsf{R}[c]_i^4$	-
4	C	-

2.3.4 Locking Protocols

The most popular mechanism to enforce a particular isolation level is to use some form of locking. We can consider each data item $x \in \mathsf{items}(db)$ to be associated with a lock that manages the access to its value. When a transaction runs an operation which accesses x, it is required to hold the lock relative to the item. The exact synchronization structure used in this case is a read/write lock that works under two modes, a shared one and an exclusive one. The former allows multiple read operations to happen in parallel while the latter makes sure that there is only one write operation executing at any point in time. This way, execution blocking only happens when transactions perform concurrent conflicting operations on the same data item. In terms of notation, a transaction T_i can lock an item using $\mathsf{L}[\kappa, x]_i$ and unlock it with $\mathsf{U}[\kappa, x]_i$ where $\kappa \in \{\mathrm{S}, \mathrm{X}\}$ is the lock mode (shared or exclusive). If a transaction requests access to item x and gets an immediate grant even if x's lock is held by another transaction, we say that the two locks are compatible [25]. We define this property for lock modes as follows:

$$\mathsf{compatible}(\kappa, \kappa') \iff (\kappa = \mathrm{S} \land \kappa' = \mathrm{S})$$

We can specify the usage of locks in a formal way by describing histories H that arise from locking protocols. Given that all accesses to database items are well locked, we know that they must be guarded by the corresponding and appropriate locks which are later released, so:

$$\forall x, i.\, (\mathsf{W}[x]_i \in \Theta_H \implies \mathsf{L}[\mathrm{X}, x]_i \mathrel{\dot\sqsubset} \mathsf{W}[x]_i \mathrel{\dot\sqsubset} \mathsf{U}[\mathrm{X}, x]_i)$$
$$\land\, (\mathsf{R}[x]_i \in \Theta_H \implies \exists \kappa.\, \mathsf{L}[\kappa, x]_i \mathrel{\dot\sqsubset} \mathsf{R}[x]_i \mathrel{\dot\sqsubset} \mathsf{U}[\kappa, x]_i)$$

On top of this, when that a transaction requests an already acquired and non-compatible lock, it must wait for the counterpart to release. This implies that as part of locking protocols' histories, we will see that:

$$\forall x, \kappa_a, \kappa_b, i, j.$$
$$(i \neq j \land \neg\mathsf{compatible}(\kappa_a, \kappa_b) \land \{\mathsf{L}[\kappa_a, x]_i, \mathsf{U}[\kappa_a, x]_i, \mathsf{L}[\kappa_b, x]_j, \mathsf{U}[\kappa_b, x]_j\} \subseteq \Theta_H)$$
$$\implies (\mathsf{U}[\kappa_b, x]_j \mathrel{\dot\sqsubset} \mathsf{L}[\kappa_a, x]_i \lor \mathsf{U}[\kappa_a, x]_i \mathrel{\dot\sqsubset} \mathsf{L}[\kappa_b, x]_j)$$

Two-Phase-Locking (2PL) is a concurrency control protocol which is vastly used in commercial *DBS* products. In its base version, as the name suggests, each transaction T_i's execution can be divided into two phases [3]: a growing phase where T_i sequentially acquires locks for all the data items it is going to access, and a shrinking phase that starts on the first lock release and will unlock all the items it holds, one after the other. According to 2PL, during the growing phase no locks are released, and once the shrinking phase has started, no locks are acquired. Formally, all histories that result from the use of 2PL will have an additional property in comparison to the ones already mentioned for locking protocols. In fact lock operations will be further ordered as to comply with the two phase separation:

$$\forall x, y, \kappa_a, \kappa_b, i.\, (\{\mathsf{L}[\kappa_a, x]_i, \mathsf{U}[\kappa_b, y]_i\} \subset \Theta_H \implies \mathsf{L}[\kappa_a, x]_i \mathrel{\dot\sqsubset} \mathsf{U}[\kappa_b, y]_i)$$

A variation of the method is Conservative Two-Phase-Locking (C2PL) that works in a similar way with the key difference that all locks ever needed as part of a transaction are acquired before any read or write operation happens. As a consequence, we can state that all histories H adhering to C2PL will be such that:

$$\forall x, op, \kappa, i.\, (op \neq \mathsf{L} \land \{\mathsf{L}[\kappa, x]_i, op\} \subseteq \Theta_H) \Rightarrow \mathsf{L}[\kappa, x]_i \mathrel{\dot\sqsubset} op$$

This is done primarily in order to prevent situations in which 2PL would exhibit deadlocks, since lock acquisitions can be done in some precise order that all transactions need to follow.

Another version of 2PL which is widely used inside concrete implementations is Strict Two-Phase-Locking (S2PL) which imposes a different kind of constraint. Specifically, it only allows transactions to release the locks they hold after they have either committed or aborted. Its histories follow the rules listed for standard 2PL and moreover they guarantee that:

$$\forall x, op, \kappa, i.\, (op \in \{\mathsf{C}_i, \mathsf{A}_i\} \land \{\mathsf{U}[\kappa, x]_i, op\} \subset \Theta_H) \Rightarrow op \mathrel{\dot\sqsubset} \mathsf{U}[\kappa, x]_i$$

The Strong-Strict Two-Phase-Locking (SS2PL) protocol works by combining the two previous approaches in a way that all locking operations happen before any access to items and all unlocking operations appear after a transaction's commit or abort completion. We summarize the pecularities of the four variants of the protocol in Table 2.3. Gradual locking refers to a transaction acquiring locks for cells as it needs to, while the extreme version makes it obtain all needed locks before starting. Similarly, gradual unlocking releases locks whenever they are no longer needed (while still following the two phases rule) and, on the other hand, extreme unlocking only releases locks after a transaction has committed.

Table 2.3: Two-Phase-Locking variants' constraints.

Constraint	2PL	C2PL	S2PL	SS2PL
Gradual Locking	✓	×	✓	×
Extreme Locking	×	✓	×	✓
Gradual Unlocking	✓	✓	×	×
Extreme Unlocking	×	×	✓	✓

2.3.5 Serializability & Transactional Models

The theory for database transactions that we introduced so far has recently had a large number of applications to the more general context of programming languages. Transactions are being effectively used in the latter to provide an overall simplification to shared-memory concurrency. Popular commercial programming languages provide syntactic constructs to embed a chunk of code into an atomic transaction. This allows to free programmers from the burden of mantaining the fictional atomicity of operations, and delegate everything to the language implementation. We explore two models of transactions in this setting. On top of this, we later explain work related to reasoning about consistency models within distributed databases.

Push/Pull Model The model of transactions described in [16] is presented as a general theory of serializability, that abstracts away all of the algorithm and implementation details, in order to find a compact set of operations used in the vast majority of cases. The model does not use a concrete machine state, i.e. a global storage or memory heap, to track the effect of transactions' operations such as writes. Instead, it assigns a local *log* of operations to each transaction and considers a unique shared log, which records the history of all globally visible operations. The semantics allow transactions to PUSH or UNPUSH, in order to share their local effects with the global log or to recall them from it. Conversely, they can PULL an operation from the shared log into their local view, together with detangling from one through an UNPULL. These two types of actions can be seen as reading or *forgetting* a read from the global storage. The framework comes with a proof of serializability within the semantic model, meaning that once users map their algorithms to the *Push/Pull* semantic rules, they obtain a proof of serializability as a consequence.

Semantics for Transactions The work done in [18] presents a set of formal languages which are able to describe particular behaviours that arise from the use of transactions within a programming language. Those are defined in terms of small-step operational semantics that allow the interleaving of parallel operations. They are high-level in order not to appear too complicated to the eyes of a programmer, while still detailed enough to express the required features. The languages of the AtomsFamily are listed here, in increasing order of consistency strictness:

1. As part of StrongBasic only one single thread is allowed to execute a transaction at one time. On top of this, a running transaction cannot spawn a new thread and other parallel threads cannot read or write memory cells from the global heap. This is the strongest and simplest of the languages.

2. The StrongNestedParallel language is an extension of the one described in (1). It allows to spawn threads within transactions in multiple ways. The expression $\mathsf{spawn}_{\mathsf{oc}}$, *spawn on commit*, is allowed anywhere in a program, but if included in a transaction, the spawned thread only starts reducing once the parent transaction completes execution. $\mathsf{spawn}_{\mathsf{ip}}$, *internally parallel*, is only allowed within a transaction and the latter can only commit once the spawned thread has completed.

3. Weak is a modified version of the language in (1) with the removal of the nontransactional code constraint. As a consequence, threads running in parallel to a transaction can access the shared heap through nontransactional commands.

4. Finally, the WeakUndo language has all the features of the one in (3) together with the possibility for a transaction to abort, undo all of the heap updates and retry.

Consistency Models Cerone et al. [7] define a general framework for transactional consistency models within the context of distributed systems, in particular of geo-replicated databases. In this scenario, guaranteeing that the database behaves as if transactions get executed serially is often too slow, or unfeasible due to network failures. This is the main reason why weaker consistency guarantees are taken into account, which allows the occurrence of behaviours prohibited by serializability. The framework itself enables to uniformly specify a variety of such models by taking an axiomatic approach, with the goal of reasoning about databases at a high level of abstraction. What is key in finding common ground among a number of consistency models is the property of atomic visibility. This states that it is guaranteed that all or none of the effects of a transaction are visible to others. Replicated database systems typically enforce this behaviour. The crucial difference is then to understand and analyse when the effects become visible.

As part of the framework, reasoning happens at the level of abstract database executions, which are structures of events with particular relations established on them. Each introduced consistency model is specified through consistency axioms that restrict the possible abstract executions. We will go through the high-level details of two of such models to understand how they are generally developed.

The baseline model is **Read Atomic**, which is defined through two axioms. The internal consistency axiom, Int, ensures that, as part of the body of a transaction, the value read from a database item x is equivalent to the last value written to x or the last value read from it. On the other hand, the external consistency axiom, Ext, enforces that any time a transaction T reads a value from item x before it writes to it, it will obtain a value which was written to x by transactions preceding T. In the case where no transaction wrote to x, the default value of 0 will be returned.

The *precedence* between transactions is given by the visibility relation, VIS. We write $T_1 \xrightarrow{\mathsf{VIS}} T_2$ in order to express that T_2's internal operations can be influenced by T_1's effect on the database. Another fundamental relation of abstract executions is arbitration, AR, which is defined on transactions. The meaning of $T_1 \xrightarrow{\mathsf{AR}} T_2$ is that transaction T_2's version of database items supersedes the one written to by T_1.

The adoption of the **Read Atomic** model leads to a variety of anomalies including the one of causality violation, an example of which is graphically shown in Figure 2.10. In order to forbid this kind of behaviour we are required to strengthen the consistency model with a new axiom, TransVis. Causal consistency will in fact be obtained by enforcing the VIS relation to be transitive. It follows that the effects of transactions ordered by VIS are observed by others in this same order.

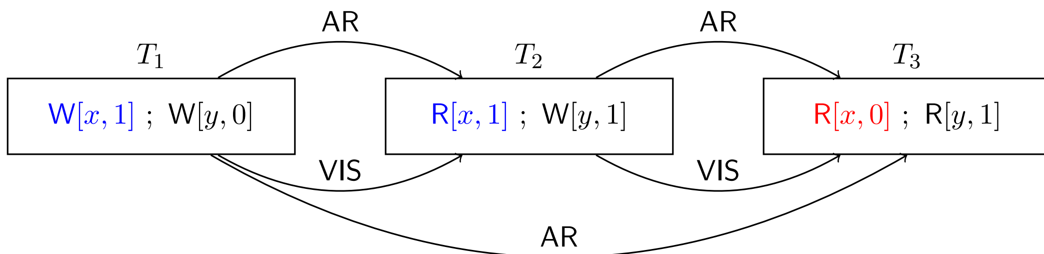


Figure 2.10: An example of the causality violation anomaly.

3. Motivating example

We present an introductory example to give a taste of the work done from a high-level point of view. All of the notation and details that follow will be formally introduced and explained in later sections of this report.

Let's assume the existence of a retail bank which keeps all records of its customers' accounts in a database. The latter is a simple collection of indexed numerical items. We can reach a particular item in the database using its index key, also known as address. A bank account is implemented as a single item a stored in the database, where a's address is the customer's unique identifier and a's content is the amount of money currently in the account.

The bank has a particular client, Bob, who pays his flat's rent of \$500 the first day of every month through direct debit. Coincidentally, the bank also pays interest to its customers on the first day of every month. In a fictional period of high interest rates, it gives \$600 per month to all of its 5 customers, including Bob. The modern bank's backend software system is setup in a way that the two described operations are scheduled to run in transactions that execute at the same time, concurrently. The executed code is displayed in Figure 3.1. We assume that Bob's unique identifier, and therefore account address, is 2. This explains why the first transaction declaration assigns value 2 to variable `bob` to later dereference it and read the amount of money in the account. In the case where there is more than \$500 available, the same quantity is subtracted and the account's database item is updated. At the same time, as part of the other transaction, Bob's identifier is retrieved and used to increment by \$600 the amount in his account.

```
begin                          ||
  bob := 2;                    ||  begin
  m := [bob];                  ||    id := 2;
  if (m ≥ 500)                 ||    x := [id];
    [bob] := m − 500           ||    [id] := x + 600;
  else                         ||  end
    skip                       ||
end                            ||
```

Figure 3.1: The bank's database program ran every month in parallel to execute Bob's direct debit and credit interest to all of its customers.

If we now were to assume the atomicity of operations at the level of single commands, we would need to consider all possible interleavings of the two blocks of code. Some of the possible executions that arise in this context are incorrect and can lead to serious issues. Table 3.1 shows every possible end state of the database, assuming that it started as $\{0 \mapsto n_0, 1 \mapsto n_1, 2 \mapsto n_2, 3 \mapsto n_3, 4 \mapsto n_4\}$. The final results that we would expect, and want, are the ones described by options (i) and (ii) in the table, where Bob's account either contains n_2+600 dollars or $n_2 + 100$, based on whether its initial allowance was less or more than \$500. Nevertheless,

(iii) might still happen if the direct debit transaction reads Bob's money into `m` first, but writes to it it after the other transaction has added \$600 to the total.

Table 3.1: All the possible final database outcomes from the program in Figure 3.1.

#	**0**	**1**	**2**	**3**	**4**
i	n_0	n_1	$n_2 + 600$	n_3	n_4
ii	n_0	n_1	$n_2 + 100$	n_3	n_4
iii	n_0	n_1	$n_2 - 500$	n_3	n_4

This clearly demonstrates that such a system is not safe for usage and requires a *concurrency control* mechanism in place. We can for instance consider a setting where each transaction locks access to the entire database, executes, and once done, gives its priviledge to another transaction which is running in parallel. Such a primitive technique would clearly not provide good performance but would guarantee the absence of the issues we encountered previously and would be less hard to reason about. This is because, given two programs p and p' running concurrently, we only have to consider the case where p goes first followed by p' and where p' runs before p. On the other hand, *Two-Phase-Locking* works at the level of single database entries and allows concurrent transactions to only acquire locks for the items they read or write. The crucial constraint enforced by the mechanism is that, once a transaction releases a lock on an item, it cannot later lock any other one. This requirement is what will allows us to show in this report the equivalence of any execution to one where transactions run one before the other, with no interleaving. We will therefore proceed with a formal proof of the program assuming it runs in the environment described at the beginning of the paragraph.

In order to reason about the concurrent program in Figure 3.1, we introduce an abstraction to describe bank accounts. One of these is represented by a logic predicate of the shape $\mathsf{Account}(x, m)$. The latter describes an account that resides in the database at address x and currently contains m dollars. As the bank allows no overdraft, we add a constraint to all accounts in order to enforce m to be a non-negative number. On top of this, the same predicate gives capabilities to perform the two fundamental actions:

- Pay *rent* by removing \$500 from the account when there is enough allowance.
- Credit *interest* by adding \$600 to the total.
- *Restart* the whole process when a new month starts.

Formally, we represent all of this information as:

$$\mathsf{Account}(x, m) \equiv \exists r. \boxed{x \mapsto m \land m \geq 0}^{r}_{I(r,x)} * [\textsc{Rent}]^r * [\textsc{Interest}]^r * [\textsc{Restart}]^r$$

Everything that is in the graphical rectangle, also known as a *box*, represents the content of a part, or *region*, of the database which is shared among multiple transactions able to access it. The region is uniquely identified by r and transactions can modify it by following a precise set of rules that is listed in the following table, which we call the region *interpretation*. Intuitively, each action of the shape $a \rightsquigarrow b$ in the interpretation of a region, shows the required state that needs to be present in the region, a, followed by what that same chunk will be replaced with, b.

$$I(r, x) \triangleq \left(\begin{array}{rl} \textsc{Rent}: & \exists m.\, x \mapsto m \land m \geq 500 \rightsquigarrow x \mapsto m - 500 * [\textsc{Rent}]^r \\ \textsc{Interest}: & \exists m.\, x \mapsto m \rightsquigarrow x \mapsto m + 600 * [\textsc{Interest}]^r \\ \textsc{Restart}: & \exists m.\, x \mapsto m * [\textsc{Rent}]^r * [\textsc{Interest}]^r \rightsquigarrow x \mapsto m \end{array} \right)$$

The previously introduced capabilities are mapped to concrete mutations of the shared region's contents. For instance, if we look at the INTEREST case, we notice how a transaction is able to transform $x \mapsto m$ into $x \mapsto m + 600$ given that it holds the capability in its local state, and moves the latter back in the region. Since all region updates have to be modelled as part of its interpretation, it follows that the latter serves the crucial purpose of declaring to all transactions how might the shared state change under the effect of a concurrent program. This enables users of the program logic to write *stable* assertions in their programs' specifications. If they do so, it is then ensured that a particular assertion will remain valid, no matter how concurrent transactions modify a region's content according to the interpretation's actions. Such a constraint automatically gives us the *composionality* of proofs, meaning that two transactions can be proven independently and then combined in parallel.

We can now build a sketch proof for the two transactions in isolation in Figure 3.2 and Figure 3.3. Notice how in both cases, the preconditions are stable with respect to the interference coming from the other transaction. In fact, we can see a disjunction as part of the shared region content which takes into account the possibility of a concurrent action occurring before the transaction starts, and for which we don't possess a full capability. For example, in Figure 3.2 we can se that the precondition includes $2 \mapsto m \vee 2 \mapsto m + 600$ which, by only considering the intereference table, reflects both the situation where the considered transaction is the first one to execute, and the one where the other transaction precedes it.

$$
\left\{\exists r. \boxed{(2 \mapsto m \vee 2 \mapsto m + 600) \wedge m \geq 0}^{r}_{I(r,2)} * [\textsc{Rent}]^{r}\right\}
$$

begin

$\quad \{2 \mapsto m \vee 2 \mapsto m + 600 \wedge m \geq 0\}$

$\quad \texttt{bob} := 2;$

$\quad \{\texttt{bob} \mapsto m \vee \texttt{bob} \mapsto m + 600 \wedge m \geq 0\}$

$\quad \texttt{m} := [\texttt{bob}];$

$\quad \{\texttt{bob} \mapsto \texttt{m} \wedge (\texttt{m} = m \vee \texttt{m} = m + 600) \wedge m \geq 0\}$

$\quad \texttt{if}\ (\texttt{m} \geq 500)$

$\qquad \{\texttt{bob} \mapsto \texttt{m} \wedge (\texttt{m} = m \vee \texttt{m} = m + 600) \wedge \texttt{m} \geq 500\}$

$\qquad [\texttt{bob}] := \texttt{m} - 500$

$\qquad \{\texttt{bob} \mapsto \texttt{m} - 500 \wedge (\texttt{m} = m \vee \texttt{m} = m + 600) \wedge \texttt{m} \geq 500\}$

$\quad \texttt{else}$

$\qquad \{\texttt{bob} \mapsto m \wedge 0 \leq m < 500\}$

$\qquad \texttt{skip}$

$\qquad \{\texttt{bob} \mapsto m \wedge 0 \leq m < 500\}$

end

$$
\left\{\begin{array}{l}
\left(\exists r. \boxed{2 \mapsto m - 500 \wedge m \geq 500 * [\textsc{Rent}]^{r}}^{r}_{I(r,2)}\right) \vee \\
\left(\exists r. \boxed{2 \mapsto m + 100 \wedge m \geq 0 * [\textsc{Rent}]^{r}}^{r}_{I(r,2)}\right) \vee \\
\left(\exists r. \boxed{2 \mapsto m \wedge 0 \leq m < 500}^{r}_{I(r,2)} * [\textsc{Rent}]^{r}\right)
\end{array}\right\}
$$

Figure 3.2: Sketch proof of the rent direct debit transaction.

$$\left\{\exists r. \boxed{(2 \mapsto m \wedge m \geq 0) \vee (2 \mapsto m - 500 \wedge m \geq 500)}^{r}_{I(r,2)} * [\textsc{Increment}]^{r}\right\}$$

```
begin
  {(2 ↦ m ∧ m ≥ 0) ∨ (2 ↦ m − 500 ∧ m ≥ 500)}
  id := 2;
  {(id ↦ m ∧ m ≥ 0) ∨ (id ↦ m − 500 ∧ m ≥ 500)}
  x := [id];
  {id ↦ x ∧ ((x = m ∧ m ≥ 0) ∨ (x = m − 500 ∧ m ≥ 500))}
  [id] := x + 600;
  {id ↦ x + 600 ∧ ((x = m ∧ m ≥ 0) ∨ (x = m − 500 ∧ m ≥ 500))}
end
```

$$\left\{\exists r. \boxed{(2 \mapsto m + 600 \wedge m \geq 0) \vee (2 \mapsto m + 100 \wedge m \geq 500) * [\textsc{Increment}]^{r}}^{r}_{I(r,2)}\right\}$$

Figure 3.3: Sketch proof of the credit interest transaction.

Once the proofs are done, we can combine them for our original example in order to obtain the verified result we were hoping for: the end effect of the program is either to add \$100 to Bob's account when he has allowance to pay his rent, or to add \$600 if he does not and later flag him with an alert. For this reason, we formally introduce two other predicates that allow to express this.

$$\mathsf{Alerted}(x, m) \equiv \exists r. \boxed{x \mapsto m \wedge m \geq 0 * [\textsc{Increment}]^{r}}^{r}_{I(r,x)} * [\textsc{Rent}]^{r} * [\textsc{Restart}]^{r}$$

$$\mathsf{Processed}(x, m) \equiv \exists r. \boxed{x \mapsto m \wedge m \geq 0 * [\textsc{Increment}]^{r} * [\textsc{Rent}]^{r}}^{r}_{I(r,x)} * [\textsc{Restart}]^{r}$$

As we can see, in the case of Alerted, the RENT capability is still in the local state and has not been moved to the shared region, meaning that the first transaction could not succeed in its direct debit operation. The final sketch proof appears in Figure 3.4. When combining the post conditions of the two transactions, we elide the conflicting assertions that result in `false`, and only keep the valid ones.

$$\{\mathsf{Account}(2, n_2)\}$$

```
begin                      ||  begin
  bob := 2;                ||    id := 2;
  m := [bob];              ||    x := [id];
  if (m ≥ 500)             ||    [id] := x + 600;
     [bob] := m − 500      ||  end
end                        ||
```

$$\{\mathsf{Alerted}(2, n_2 + 600) \vee \mathsf{Processed}(2, n_2 + 100)\}$$

Figure 3.4: High-level sketch proof of the parallel composition of the *rent* and *interest* transactions.

4. The mCAP Program Logic

The path to the definition of a program logic for serializable transactions starts in this section, where we lay its foundations by taking elements from existing logics for concurrency, and then we expand it to a general framework that can be instantiated for one's needs. As a consequence, we provide the first formalisation of the mCAP logic model (where the m stands for *modern*) and later of its semantics. This work builds upon the CAP program logic, presented in Section 2.2 and first introduced in [12], and borrows some of the definitions and proof strategies from CoLoSL [23] [22]. The main novelties introduced in mCAP, as compared to regular CAP, are concerned with freeing its usage as a logic from some hard constraints and create the necessary space for transactional reasoning. mCAP is in fact parametric with respect to the region capabilities (or *guards*) used, it allows multiple region updates at once through the repartitioning operation, and does not enforce the entirety of a region capability to be held in order to construct the region itself.

The program logic that we later provide in Section 5 will be a concrete instantiation of mCAP and will enable us to reason about systems adopting real transactional atomicity and later about ones using 2PL to handle their concurrency control.

4.1 Partial Commutative Monoid

The concept of a partial commutative monoid (pcm) was first used to model program resources in [6] through a separation algebra, in order to abstract from the standard separation logic model for heaps equipped with an operator to compose them. We explain such concept here and list some of its core properties that will later be used. The definition we give follows the one presented as part of the Views framework [11].

Definition 4.1. (Partial commutative monoid). A *partial commutative monoid* with multiple units is a triple $(\mathcal{M}, \bullet, \mathbf{0})$ where $\mathcal{M}$ is a set equipped with a partial operator $\bullet : \mathcal{M} \times \mathcal{M} \rightharpoonup \mathcal{M}$ and a unit set $\mathbf{0} \subseteq \mathcal{M}$ satisfying:

- Commutativity: $m_1 \bullet m_2 = m_2 \bullet m_1$ when either is defined.
- Associativity: $m_1 \bullet (m_2 \bullet m_3) = (m_1 \bullet m_2) \bullet m_3$ when either is defined.
- Existence of unit: for all $m \in \mathcal{M}$ there exists $u \in \mathbf{0}$ such that $u \bullet m = m$.
- Minimality of unit: for all $m \in \mathcal{M}$ and $u \in \mathbf{0}$, if $u \bullet m$ is defined then $u \bullet m = m$.

There is often the need to compare elements of a partial commutative monoid. For this reason, we introduce the ordering relation between pairs of such elements, which is defined when there is some element that composed with the first one results in the second one.

Definition 4.2. (Ordering) Given a pcm $(\mathcal{M}, \bullet, \mathbf{0})$, the *ordering* relation $\leq : \mathcal{M} \times \mathcal{M}$ is defined as:

$$\leq \triangleq \{(m_1, m_2) \mid \exists m.\, m_1 \bullet m = m_2\}$$

We write $m_1 \leq m_2$ for $(m_1, m_2) \in \leq$. Next, we introduce a relation between elements of a pcm that allows to retrieve all of the compatible element-pairs. This means that if m_1 and m_2 are compatible, then their composition is defined.

Definition 4.3. (Compatibility). Given a pcm $(\mathcal{M}, \bullet, \mathbf{0})$, the *compatibility* relation $\sharp : \mathcal{M} \times \mathcal{M}$ is defined as:

$$\sharp \triangleq \{(m_1, m_2) \mid \exists m.\, m_1 \bullet m_2 = m\}$$

4.2 Worlds

A *world* is the structure that represents how a thread sees all of the existing resources and their states, together with a way to describe how these can be modified. We now informally present all of a world's components and properties which are later formalised. A world is in fact a *well-formed* triple $(l, g, \mathcal{J})$, where:

- The *logical local state* l represents the resources which are locally owned by the thread and are not externally accessible, i.e. no other thread can see nor update them.
- The *shared state* g represents all of the globally shared resources which are divided into regions and accessible to all threads.
- The *action model* $\mathcal{J}$ describes how, for every region in the shared state, a thread holding a particular *capability* can update the region. Action models include partial functions, one per region, which associate capabilities to *actions*. An action is a pair (s, s') of logical states where s is the *pre-state*, the state of the region before the action is applied, and s' is the *post-state* which is the state of the region after the action takes place.

Worlds can be composed whenever they have the same shared state, action model and a disjoint local logical state. We now proceed to define the latter as tuples whose first component is a machine state, the heap, while the second one is a mapping from regions to capabilities held by the thread for that particular region. Given that mCAP is parametric with respect to the partial commutative monoid representing machine states and capabilities, we allow users of the framework to chose a suitable instantiation to tackle a particular program verification.

Parameter 4.4. (Machine states pcm). Assume a partial, commutative monoid with multiple units which represents *machine states*, $(\mathbb{M}, \bullet_{\mathbb{M}}, \mathbf{0}_{\mathbb{M}})$. Elements of $\mathbb{M}$ are ranged over by $m, m_1, \ldots, m_n$.

Parameter 4.5. (Primitive capability pcm). Assume a partial, commutative monoid with multiple units which represents *primitive capability resources*, $(\mathbb{K}, \bullet_{\mathbb{K}}, \mathbf{0}_{\mathbb{K}})$. Elements of $\mathbb{K}$ are ranged over by $\kappa, \kappa_1, \ldots, \kappa_n$.

Given that the shared state in a mCAP world is organized into regions, we require a way to uniquely characterize each of those. We therefore define distinct region identifiers.

Definition 4.6. (Region identifiers). Assume a set of *region identifiers* Rid, ranged over by $r, r_1, \ldots, r_n$.

Definition 4.7. (Capability). Given a pcm for primitive capabilities, $(\mathbb{K}, \bullet_{\mathbb{K}}, \mathbf{0}_{\mathbb{K}})$, the set of *capabilities*, RKap, is defined as the set of partial functions with a finite domain from region identifiers to primitive capabilities.

$$\mathsf{RKap} \triangleq \mathsf{Rid} \stackrel{\text{fin}}{\rightharpoonup} \mathbb{K}$$

The RKap set is ranged over by $\rho, \rho_1, \ldots, \rho_n$. Composition on capabilities, $\circ : \mathsf{RKap} \times \mathsf{RKap} \rightharpoonup \mathsf{RKap}$, is defined as follows:

$$(\rho \circ \rho')(r) \triangleq \begin{cases} \rho(r), & \text{if } r \notin \mathsf{dom}(\rho') \\ \rho'(r), & \text{if } r \notin \mathsf{dom}(\rho) \\ \rho(r) \bullet_{\mathbb{K}} \rho'(r), & \text{otherwise} \end{cases}$$

The capabilities pcm is defined as $(\mathsf{RKap}, \circ, \mathbf{0}_{\mathsf{RK}})$, where the capability unit set, $\mathbf{0}_{\mathsf{RK}} : \mathsf{Rid} \overset{\text{fin}}{\rightharpoonup} \mathbb{K}$, is the function with an empty domain.

One can see region capabilities as a way to record what capabilities are held for regions present in the world's shared state. Such capabilities can be owned by a thread in its local state or be included inside of a region.

A logical state is therefore a pair (m, ρ) where $m \in \mathbb{M}$ is a machine state and $\rho \in \mathsf{RKap}$ is a region capability, describing what resources are owned and which capabilities are held.

Definition 4.8. (Logical states). Given a pcm for machine states $(\mathbb{M}, \bullet_{\mathbb{M}}, \mathbf{0}_{\mathbb{M}})$, and one for capabilities $(\mathbb{K}, \bullet_{\mathbb{K}}, \mathbf{0}_{\mathbb{K}})$, the set of *logical states*, $\mathsf{LStates}$, ranged over by $l, l_1, \ldots, l_n$ is defined as:

$$\mathsf{LState} \triangleq \mathbb{M} \times \mathsf{RKap}$$

Composition on logical states, $\circ : \mathsf{LState} \times \mathsf{LState} \rightharpoonup \mathsf{LState}$, is defined as:

$$(m, \rho) \circ (m', \rho') \triangleq (m \bullet_{\mathbb{M}} m', \rho \circ \rho')$$

The logical states unit set is $\mathbf{0}_{\mathsf{L}} = \{(m, \rho) \mid m \in \mathbf{0}_{\mathbb{M}} \wedge \rho \in \mathbf{0}_{\mathsf{RK}}\}$ and the pcm of logical states is defined as $(\mathsf{LState}, \circ, \mathbf{0}_{\mathsf{L}})$.

Given a logical state l we use l_{M} and l_{K} to refer to its first and second projections respectively. The entirety of the globally shared state is divided in regions, each of which carries a logical state to describe its contents both in terms of machine state and capabilities.

Definition 4.9. (Shared states). The set of *shared states* GState, ranged over by $g, g_1, \ldots g_n$, is defined as the set of partial functions with a finite domain, mapping region identifiers to logical states:

$$\mathsf{GState} \triangleq \mathsf{Rid} \overset{\text{fin}}{\rightharpoonup} \mathsf{LState}$$

The *combination function*, $\llbracket - \rrbracket : \mathsf{GState} \to \mathsf{LState}$ is defined as:

$$\llbracket g \rrbracket \triangleq \prod_{r \in \mathsf{dom}(g)}^{\circ} g(r)$$

The *cross-composition* function between logical states and shared states, $\oplus : \mathsf{LState} \times \mathsf{GState} \rightharpoonup \mathsf{LState}$, is defined as:

$$l \oplus g \triangleq l \circ \llbracket g \rrbracket$$

We can now move onto analysing how the shared state can be modified by threads. This interaction is modelled in an abstract way through the use of actions that represent a transformation from the current logical state to a new, updated one.

Definition 4.10. (Action models). The set of *actions* Action, is defined as the set of tuples of logical states.

$$\mathsf{Action} \triangleq \mathsf{LState} \times \mathsf{LState}$$

Actions are used as part of of action models. The set of *action models*, AMod, ranged over by $\mathcal{J}, \mathcal{J}_1, \ldots, \mathcal{J}_n$, is defined as follows.

$$\mathsf{AMod} \triangleq \mathsf{Rid} \overset{\text{fin}}{\rightharpoonup} \mathbb{K} \overset{\text{fin}}{\rightharpoonup} \mathcal{P}(\mathsf{Action})$$

An action model with an empty domain is simply denoted by $\emptyset$.

We refer to the first component of an action (l, l') as the *pre-state* while the second one as the *post-state*.

The property of well-formedness applies to tuples of the shape $(l, g, \mathcal{J})$, when the cross-composition of the local and shared state is defined, the resulting region capability is contained in the action model and the regions in the shared state are the same as the ones described by the action model.

Definition 4.11. (Well-formedness). A given triple $(l, g, \mathcal{J}) : \mathsf{LState} \times \mathsf{GState} \times \mathsf{AMod}$ is *well-formed*, written $\mathsf{wf}((l, g, \mathcal{J}))$, when the following holds:

$$\mathsf{wf}((l, g, \mathcal{J})) \iff \exists l'.\, l \oplus g = l' \land \mathsf{dom}(l'_{\mathsf{K}}) \subseteq \mathsf{dom}(\mathcal{J}) \land \mathsf{dom}(g) = \mathsf{dom}(\mathcal{J})$$

Definition 4.12. (World). The set of all *worlds*, World, ranged over by $w, w_1, \dots w_n$, is defined as the set of well-formed triples containing a local state, a global one and an action model.

$$\mathsf{World} \triangleq \{w \in \mathsf{LState} \times \mathsf{GState} \times \mathsf{AMod} \mid \mathsf{wf}(w)\}$$

Composition on worlds, $\bullet : \mathsf{World} \times \mathsf{World} \rightharpoonup \mathsf{World}$, is defined by composing local states and requiring that shared states and action models be identical.

$$(l, g, \mathcal{J}) \bullet (l', g', \mathcal{J}') \triangleq \begin{cases} (l \circ l', g, \mathcal{J}), & \text{if } g = g' \text{ and } \mathcal{J} = \mathcal{J}' \\ & \text{and } \mathsf{wf}((l \circ l', g, \mathcal{J})) \\ \mathsf{undef} & \text{otherwise} \end{cases}$$

The worlds pcm is defined as $(\mathsf{World}, \bullet, \mathbf{0}_{\mathsf{W}})$, where the worlds unit set is defined as any well-defined world whose local state is part of the logical state unit set, $\mathbf{0}_{\mathsf{L}}$:

$$\mathbf{0}_{\mathsf{W}} = \{(l, g, \mathcal{J}) \mid (l, g, \mathcal{J}) \in \mathsf{World} \land l \in \mathbf{0}_{\mathsf{L}}\}$$

Given a world w, we write $w_{\mathsf{L}}, w_{\mathsf{S}}$ and w_{A} for its first, second and third projections.

4.3 Assertions

Assertions in mCAP follow the ones in standard CAP with the only difference of being parametric with respect to the machine state assertions and capability assertions that describe elements of the given sets $\mathbb{M}$ and $\mathbb{K}$ respectively. Judgements in mCAP have the shape $\Delta \vdash \{P\}\ \mathbb{P}\ \{Q\}$ where P, Q are *assertions*, $\mathbb{P}$ is a program and Δ is an *axiom definition*, all defined in this section.

We assume the presence of an infinite set of logical variables, $x \in \mathsf{LVar}$ and logical environments, LEnv, such that $e \in \mathsf{LEnv} \triangleq \mathsf{LVar} \rightarrow \mathsf{Val}$. Logical environments associate logical variables with their values. Also, since mCAP is an extension of CAP, we provide support for abstract predicates used to express concrete properties. We assume to be given a set of predicate environments, PEnv that associate each predicate name (coming from the set PName) and its arguments, to a set of worlds that satisfy them. Formally $\delta \in \mathsf{PEnv} \triangleq \mathsf{PName} \times \mathsf{Val}^* \rightarrow \mathcal{P}(\mathsf{World})$.

Parameter 4.13. (Machine state assertions). Assume a set of *machine state assertions* MAssn, ranged over by $\mathcal{M}, \mathcal{M}_1, \dots, \mathcal{M}_n$. Given the machine states pcm $(\mathbb{M}, \bullet_{\mathbb{M}}, \mathbf{0}_{\mathbb{M}})$, assume an assertion semantics function, that maps machine state assertions to the elements of $\mathbb{M}$ they describe:

$$[\![-]\!]^{\mathbb{M}}_{-} : \mathsf{MAssn} \rightarrow \mathsf{LEnv} \rightarrow \mathcal{P}(\mathbb{M})$$

Parameter 4.14. (Capability assertions). Assume a set of *capability assertions* KAssn ranged over by $\mathcal{K}, \mathcal{K}_1, \dots, \mathcal{K}_n$ and an associated semantics function that maps such assertions to elements of the capability separation algebra given as $(\mathbb{K}, \bullet_{\mathbb{K}}, \mathbf{0}_{\mathbb{K}})$.

$$[\![-]\!]^{\mathbb{K}}_{-} : \mathsf{KAssn} \rightarrow \mathsf{LEnv} \rightarrow \mathcal{P}(\mathbb{K})$$

Definition 4.15. (Assertion syntax). Assertions are elements of the Assn set defined by the following grammar, where $x \in \mathsf{LVar}, r \in \mathsf{Rid}$ and $\alpha \in \mathsf{PName}$.

$$
\begin{aligned}
A &::= \mathtt{false} \mid \mathtt{emp} \mid \mathcal{M} \mid [\mathcal{K}]^r \\
p, q \in \mathsf{LAssn} &::= A \mid \neg p \mid \exists x.\, p \mid p \vee q \mid p * q \mid p \mathrel{-\!\!*} q \\
P, Q \in \mathsf{Assn} &::= p \mid P \vee Q \mid \exists x.\, P \mid P * Q \mid \boxed{P}_I^r \mid \alpha(\mathbb{E}_1, \ldots, \mathbb{E}_n) \\
I \in \mathsf{IAssn} &::= \emptyset \mid \{\mathcal{K} : \exists \vec{y}.\, P \rightsquigarrow Q\} \cup I \\
\Delta \in \mathsf{Ax} &::= \emptyset \mid \forall \vec{x}.\, P \implies Q \mid \forall \vec{x}.\, \alpha(\vec{x}) \equiv P \mid \Delta_1, \Delta_2
\end{aligned}
$$

The structure of assertions follows from the separation logic one, with the addition of the given capability assertions $[\mathcal{K}]^r$, shared region (or *boxed*) assertions $\boxed{P}_I^r$ and *interference* assertions I. Machine state assertions, $\mathcal{M}$ are interpreted over a world's local logical state, i.e. $\mathcal{M}$ holds true when it is satisfied by m, as part of (m, ρ), for a $\rho \in \mathbf{0}_{\mathsf{RK}}$. In a similar way, a capability assertion $[\mathcal{K}]^r$ is true in a logical state $(m, [r \mapsto \kappa])$ for $m \in \mathbf{0}_{\mathbb{M}}$, a shared region r and capability κ that satisfies $\mathcal{K}$. The assertion $\mathtt{emp}$ is only satisfied by the unit of logical states, formally any $l \in \mathbf{0}_{\mathsf{L}}$. The separating assertion $P * Q$ is true of worlds that can be split in two parts such that the first one satisfies P while the second one Q. A predicate assertion holds in all worlds belonging to its semantic interpretation as given by the environment δ. The predicate's arguments are evaluated with respect to the logical environment e.

An interference assertion I, describes actions that are enabled by a particular capability in the form of pre-condition and post-condition. I is checked against the action model $\mathcal{J}$ to verify that, for the region r it describes, every pre-condition P and post-condition Q are satisfied by $\mathcal{J}$'s actions pre- and post-state. The variables that are existentially bound as part of I's action ($\vec{y}$) are set in the logical environment e to be the same when checking for satisfaction of P and Q. This is practically done by building (in Definition 4.16) a *tiny* action model out of the assertions in I and then checking that it is contained inside of $\mathcal{J}$.

Boxed assertions of the shape $\boxed{P}_I^r$ are true of worlds $(l, g, \mathcal{J})$ where the local state is empty, $l \in \mathbf{0}_{\mathsf{L}}$, and the logical state associated to r inside of g satisfies P. We also need to make sure that the interference assertion I, attached to region r, is satisfied by $\mathcal{J}(r)$, as described in the previous paragraph.

Finally, a syntactic predicate environment Δ, allows to use assertions in order to build a parametric predicate and store its meaning throughout a proof. In Section 4.3 we will study their semantic interpretation and see how they are used.

Definition 4.16. (Interference assertion semantics). The semantics of *interference assertions*, $[\![-]\!]^{\mathrm{I}}_{r,e,\delta,\mathcal{J}} : \mathsf{IAssn} \times \mathsf{Rid} \times \mathsf{LEnv} \times \mathsf{PEnv} \times \mathsf{AMod} \to \mathsf{Rid} \overset{\text{fin}}{\rightharpoonup} \mathbb{K} \overset{\text{fin}}{\rightharpoonup} \mathcal{P}(\mathsf{Action})$, are defined as:

$$
[\![\emptyset]\!]^{\mathrm{I}}_{r,e,\delta,\mathcal{J}}(r')(\kappa) \triangleq \emptyset
$$

$$
[\![\mathcal{K} : \exists \vec{y}.\, P \rightsquigarrow Q \cup I]\!]^{\mathrm{I}}_{r,e,\delta,\mathcal{J}}(r')(\kappa) \triangleq [\![I]\!]^{\mathrm{I}}_{r,e,\delta,\mathcal{J}}(r')(\kappa) \cup
\begin{cases}
\emptyset, & \text{if } r' \neq r \\
(l_p, l_q), & \text{if } \exists g_p, g_q, e', \vec{v}.\, \kappa \in [\![\mathcal{K}]\!]^{\mathrm{K}}_e \\
& \wedge g_p(r) = l_p \wedge g_q(r) = l_q \\
& \wedge \forall \dot{r} \neq r.\, g_p(\dot{r}) = g_q(\dot{r}) \\
& \wedge (l_p, g_p, \mathcal{J}), e', \delta \vDash P \\
& \wedge (l_q, g_q, \mathcal{J}), e', \delta \vDash Q \\
& \wedge e' = e[\vec{y} \mapsto \vec{v}]
\end{cases}
$$

The way we determine semantics for interference assertions requires a precise analysis. We are effectively building a function that, once given an interference assertion I, region identifier r, logical environment, predicate environment and action model returns a new action model, $\mathcal{J}_I$. The latter only describes all of the actions modelled by I and referring to region r. In

fact, when we intepret a single action's intereference assertion $\mathcal{K} : \exists \vec{y}.\, P \rightsquigarrow Q$, we build a function that, once queried for actions in region r associated to a given capability κ, will return a tuple of logical states (l_p, l_q). Here, l_p is the local state of the world built as $(l_p, g_p, \mathcal{J})$ that is used, together with a logical environment e' where all of the $\vec{y}$ in P and Q are bound to values, in order to satisfy assertion P. In a similary way, $(l_q, g_q, \mathcal{J})$ is constructed to satisfy Q with the same environment e'. The shared state components of these *artificial* worlds are such that they are equivalent for all regions but r. On top of this, $g_p(r) = l_p$ and $g_q(r) = l_q$. For interference assertions describing a number of actions, we recursively union the result of a single interpretation with the rest of the actions in I.

Definition 4.17. (Assertion semantics). Assertion semantics are given with respect to a world $w \in \mathsf{World}$, a logical environment $e \in \mathsf{LEnv}$ and a predicate environment $\delta \in \mathsf{PEnv}$. They tell us whether a configuration w, e, δ satisfies a particular assertion.

$$
\begin{array}{rcl}
(l, g, \mathcal{J}), e, \delta \vDash p & \iff & l, e \vDash_{\mathsf{SL}} p \\
(l, g, \mathcal{J}), e, \delta \vDash \boxed{P}_I^r & \iff & l \in \mathbf{0}_{\mathsf{L}} \text{ and } \exists l'.\, (l', g, \mathcal{J}), e, \delta \vDash P \\
 & & \text{and } \exists r'.\, r' = e(r) \wedge g(r') = l' \wedge [\![I]\!]^{\mathrm{I}}_{r', e, \delta, \mathcal{J}} \subseteq \mathcal{J} \\
w, e, \delta \vDash \alpha(\mathbb{E}_1, \ldots, \mathbb{E}_n) & \iff & w \in \delta(\alpha, [\![\mathbb{E}_1]\!]^{\mathrm{E}}_e, \ldots, [\![\mathbb{E}_n]\!]^{\mathrm{E}}_e) \\
w, e, \delta \vDash \exists x.\, P & \iff & \exists v.\, w, e[x \mapsto v], \delta \vDash P \\
w, e, \delta \vDash P \vee Q & \iff & w, e, \delta \vDash P \text{ or } w, e, \delta \vDash Q \\
w, e, \delta \vDash P * Q & \iff & \exists w_1, w_2.\, w = w_1 \bullet w_2 \text{ and} \\
 & & w_1, e \vDash P \text{ and } w_2, e \vDash Q \\
l, e \vDash_{\mathsf{SL}} \texttt{false} & & \text{never} \\
l, e \vDash_{\mathsf{SL}} \texttt{emp} & \iff & l \in \mathbf{0}_{\mathsf{L}} \\
l, e \vDash_{\mathsf{SL}} \mathcal{M} & \iff & \exists m, \rho.\, l = (m, \rho) \text{ and } m \in [\![\mathcal{M}]\!]^{\mathrm{M}}_e \text{ and } \rho \in \mathbf{0}_{\mathsf{RK}} \\
l, e \vDash_{\mathsf{SL}} [\mathcal{K}]^r & \iff & \exists m, \kappa.\, l = (m, [r \mapsto \kappa]) \text{ and } \kappa \in [\![\mathcal{K}]\!]^{\mathrm{K}}_e \text{ and } m \in \mathbf{0}_{\mathbb{M}} \\
l, e \vDash_{\mathsf{SL}} \neg p & \iff & l, e \nvDash_{\mathsf{SL}} p \\
l, e \vDash_{\mathsf{SL}} p \mathrel{-\!\!*} q & \iff & \forall l'.\, l', e \vDash_{\mathsf{SL}} p \text{ implies } l \circ l', e \vDash_{\mathsf{SL}} q \\
l, e \vDash_{\mathsf{SL}} p * q & \iff & \exists l_1, l_2.\, l = l_1 \circ l_2 \text{ and} \\
 & & l_1, e \vDash_{\mathsf{SL}} p \text{ and } l_2, e \vDash_{\mathsf{SL}} q \\
l, e \vDash_{\mathsf{SL}} p \vee q & \iff & l, e \vDash_{\mathsf{SL}} p \text{ or } l, e \vDash_{\mathsf{SL}} q \\
l, e \vDash_{\mathsf{SL}} \exists x.\, p & \iff & \exists v.\, l, e[x \mapsto v] \vDash_{\mathsf{SL}} p
\end{array}
$$

Now, given a logical environment $e \in \mathsf{LEnv}$ and a predicate environment $\delta \in \mathsf{PEnv}$, we write $[\![P]\!]_{e,\delta}$ for the set of all worlds satisfying assertion P under the environments e and δ.

$$[\![P]\!]_{e,\delta} \triangleq \{w \mid w, e, \delta \vDash P\}$$

Similarly to CAP, the separating conjunction of shared state assertions on the same region is interpreted as a regular non-separating conjunction, $\wedge$, formally:

$$\boxed{P}_I^r * \boxed{Q}_I^r \iff \boxed{P \wedge Q}_I^r$$

We also syntactically allow nested regions, but they can always be separated. Their semantic meaning is therefore expressed as follows:

$$\boxed{\boxed{P}_I^r * Q}_{I'}^{r'} \iff \boxed{P}_I^r * \boxed{Q}_{I'}^{r'}$$

4.4 Environment Semantics

This section is dedicated to the description of how we shape concurrent behaviours at the model level. We start by formally defining what the rely and guarantee relations are for each thread. This allows us to describe the effects of a thread and of the environment on the current world. Such relations allow us to further state *stability* for assertions and for predicate environments, a fundamental property to later on build our proofs. All of these components are then used to formalise the concept of *repartitioning* that logically represents a thread's atomic actions included in its guarantee relation [12][23].

The rely relation models how the environment, i.e. other running threads, can reorganize or modify a world. All these behaviours are depicted as *interference* from the environment and they can be divided into two kinds: region construction and region update. Before we move on, it is important to state what it means for a region identifier to be fresh inside of a world, as such definition will be used throughout the section.

Definition 4.18. (Region freshness). A region identifier r is *fresh* with respect to a world $w = (l, g, \mathcal{J})$, when r does not appear as part of w's shared state, g.

$$\mathsf{fresh}(r, (l, g, \mathcal{J})) \iff r \notin \mathsf{dom}(g)$$

We model how the environment can create a new region out of existing resources in its own logical local state through the construction rely relation between worlds.

Definition 4.19. (Construction rely). The *construction rely* relation, $R^c : \mathsf{World} \times \mathsf{World}$, is defined as:

$$R^c \triangleq \left\{(w, w') \mid \exists r, l, a.\, \mathsf{fresh}(r, w) \land w'_{\mathsf{L}} = w_{\mathsf{L}} \land w'_{\mathsf{S}} = w_{\mathsf{S}}[r \mapsto l] \land w'_{\mathsf{A}} = w_{\mathsf{A}}[r \mapsto a]\right\}$$

The created region must have a fresh identifier r, used to map the new shared resources inside of the global state and the allowed actions in the action model. The updated world's local state w'_{L}, as seen by a thread, will obviously be unmodified given that the environment cannot touch it. The fact that R^c is defined over worlds, which are well-defined by definition, ensures that $w'_{\mathsf{L}} \circ w'_{\mathsf{S}}$ is always defined. On top of this, it will also be the case that the local state will not contain any capability for the region that has just been constructed.

The environment can also update an already existing region inside the global state, as long as it holds a capability that entitles it to perform the change.

Definition 4.20. (Update rely). The *update rely* relation, $R^u : \mathsf{World} \times \mathsf{World}$, is defined as:

$$R^u \triangleq \left\{ \big((l, g, \mathcal{J}), (l, g', \mathcal{J})\big) \mid \exists r, \kappa.\, \kappa \sharp (l \oplus g)_{\mathsf{K}}(r) \land (g, g') \in \lceil \mathcal{J}(r) \rceil(\kappa)\right\}$$

An update done by the environment is only allowed to modify the world's shared state g, while the local state l and action model $\mathcal{J}$ stay the same. The actual region update is only allowed when the environment makes use of a capability κ that is compatible with the capabilities that are currently in the local and shared states. Moreover, κ needs to enable the shared state transition as described by $\mathcal{J}(r)$. Since we allow actions to only describe a part of the region they are modifying, and not the entirety of it, we introduce and use the *action framing* function, $\lceil - \rceil : (\mathbb{K} \to \mathcal{P}(\mathsf{Action})) \times \mathbb{K} \to \mathcal{P}(\mathsf{Action})$.

$$\lceil a \rceil(\kappa) \triangleq \{(p \circ f, q \circ f) \mid (p, q) \in a(\kappa) \land f \in \mathsf{LState}\}$$

This way, all of the region resources that are not affected by the update, i.e. f, will stay the same in the resulting world.

We now have all of the ingredients to define the overall rely relation between worlds, which describes any behaviour of the environment on the global state and action model.

Definition 4.21. (Rely relation). The *rely* relation, $R : \mathsf{World} \times \mathsf{World}$, is defined as follows.

$$R \triangleq (R^c \cup R^u)^*$$

This definition is noticeably different from CAP's [12] original one, since the latter only allowed one single update and multiple constructions. In fact, we generalize it and employ the transitive closure in order to allow the environment to perform multiple updates and region constructions in one single atomic step, as seen by a thread.

Definition 4.22. (Stability). An assertion P is said to be stable with respect to a predicate environment $\delta \in \mathsf{PEnv}$, written $\mathsf{stable}_\delta(P)$, if and only if, for all $e \in \mathsf{LEnv}$ and $w, w' \in \mathsf{World}$, if $w, e, \delta \vDash P$ and $(w, w') \in R$, then $w', e, \delta \vDash P$.

Intuitively, we require the stability property as we need to be sure that any given assertion P which is currently satisfiable, is not going to be invalidated by a rely step done by the environment and modelled through R.

Definition 4.23. (Predicate environment stability). A predicate environment δ is said to be stable, written $\mathsf{pstable}(\delta)$, if and only if, for all $W \in \mathsf{range}(\delta)$, for all $w, w' \in \mathsf{World}$, if $w \in W$ and $(w, w') \in R$, then $w' \in W$. The semantics of a syntactic predicate environment $\Delta \in \mathsf{Ax}$ are defined as a set of stable predicate environments that satisfy them:

$$\begin{aligned}
[\![\emptyset]\!]^{\mathrm{P}} &\triangleq \{\delta \mid \mathsf{pstable}(\delta)\} \\
[\![\forall \vec{x}.\, P \implies Q]\!]^{\mathrm{P}} &\triangleq \{\delta \mid \mathsf{pstable}(\delta) \wedge \forall \vec{v}.\, [\![P]\!]_{\emptyset[\vec{x}\mapsto\vec{v}],\delta} \subseteq [\![Q]\!]_{\emptyset[\vec{x}\mapsto\vec{v}],\delta}\} \\
[\![\forall \vec{x}.\, \alpha(\vec{x}) \equiv P]\!]^{\mathrm{P}} &\triangleq \{\delta \mid \mathsf{pstable}(\delta) \wedge \forall \vec{v}.\, \delta(\alpha, \vec{v}) = [\![P]\!]_{\emptyset[\vec{x}\mapsto\vec{v}],\delta}\} \\
[\![\Delta_1, \Delta_2]\!]^{\mathrm{P}} &\triangleq [\![\Delta_1]\!]^{\mathrm{P}} \cap [\![\Delta_2]\!]^{\mathrm{P}}
\end{aligned}$$

The effect of a particular thread's actions is modelled through the guarantee relation. A thread can create a new region using currently owned resources in its local state and such behaviour is formally described through the G^c between worlds. On the other hand, we leave region destruction as implicit, i.e. the user can encode it as a special action in the action model associated to the region, which gets rid of everything inside of it.

Definition 4.24. (Construction guarantee). The *construction guarantee* relation, $G^c : \mathsf{World} \times \mathsf{World}$, is defined as:

$$G^c \triangleq \left\{ (w, w') \;\middle|\; \begin{array}{c} \exists r, m, l, l', a, \rho.\, \mathsf{fresh}(r, w) \wedge \mathsf{dom}(\rho) = \{r\} \wedge \\ w_{\mathsf{L}} = l \circ l' \wedge w'_{\mathsf{L}} = l \circ (m, \rho) \wedge m \in \mathbf{0}_{\mathbb{M}} \wedge \\ w'_{\mathsf{S}} = w_{\mathsf{S}}[r \mapsto l'] \wedge w'_{\mathsf{A}} = w_{\mathsf{A}}[r \mapsto a] \end{array} \right\}$$

As we can see from the definition, a thread is allowed to create a region as long as its identifier r is entirely fresh and the new shared state includes a part of its local state that is moved out of it. In exchange, w'_{L} will include capabilities for the new region. The action model is accordingly updated to describe all of the actions allowed on r.

We instead model how a thread can modify an existing region in the shared state, for which it holds the appropriate capabilities, through the guarantee update relation between worlds. As part of the relation definition, we will make use of the orthogonal of the machine state and capability pcms which is generally defined in the following statement.

Definition 4.25. (Orthogonal). Given a pcm $(\mathcal{M}, \bullet, \mathbf{0})$ and an element $m \in \mathcal{M}$, its *orthogonal* $(-)^{\perp}_{\mathcal{M}} : \mathcal{M} \to \mathcal{P}(\mathcal{M})$ is the set of all elements in $\mathcal{M}$ which are compatible with it.

$$(m)^{\perp}_{\mathcal{M}} \triangleq \{m' \mid m \sharp m'\}$$

Definition 4.26. (Update guarantee). The *update guarantee* relation, G^u : $\mathsf{World} \times \mathsf{World}$, is defined as:

$$G^u \triangleq \left\{ ((l, g, \mathcal{J}), (l', g', \mathcal{J})) \;\middle|\; \begin{array}{c} ((l \oplus g)_{\mathsf{K}})^{\perp}_{\mathbb{K}} = ((l' \oplus g')_{\mathsf{K}})^{\perp}_{\mathbb{K}} \wedge \\ (g = g' \vee \exists r, \kappa \leq l_{\mathsf{K}}(r).\, (g, g') \in \lceil \mathcal{J}(r) \rceil(\kappa) \\ \wedge ((l \oplus g)_{\mathsf{M}})^{\perp}_{\mathbb{M}} = ((l' \oplus g')_{\mathsf{M}})^{\perp}_{\mathbb{M}}) \end{array} \right\}$$

A guarantee update enables a thread to change l and g (while not introducing new regions), leaving the action model $\mathcal{J}$ unmodified. During the update, a thread cannot introduce new capabilities as part of its local state or the global shared one. This requirement is enforced by checking that the orthogonal (Definition 4.25) of all capabilities stays the same as part of the transition from l, g to l', g'. The update can further leave the global state unmodified (thus only performing a local change) or it is entitled to modify it through an action, as long as the running thread holds an appropriate capability in its local state. As part of an update, there can be a transfer of resources from the local to the shared state together with their mutation. Since we do not want to enable *fictitious* resource creation, we require the guarantee relation to preserve the orthogonal of the local and global machine state's composition as well.

We can finally define the overall guarantee relation between worlds, expressed through G.

Definition 4.27. (Guarantee). The *guarantee* relation, G : $\mathsf{World} \times \mathsf{World}$, is defined as:

$$G \triangleq (G^c \cup G^u)^*$$

Similar to the rely relation, the definition uses a transitive closure to allow multiple region creations and updates as part of a single atomic step performed by a thread. This gives the main reason behind the *orthogonal* constraint as part of the update guarantee definition. A practical explanation of why this condition is required is illustrated in the following example. Assume a situation where we instantiate machine states as standard variable stores and we leave capabilities as abstract or not explicitly specified (through the use of $-$). Let's consider what happens when the action model $\mathcal{J}$ has the following action mapping for shared region r and the orthogonal preservation is not required.

$$\mathcal{J}(r) = \left\{ \begin{array}{ll} \kappa_1 : & ([x \mapsto 1], \kappa_2) \rightsquigarrow ([x \mapsto 2, y \mapsto 1], \emptyset) \\ \kappa_2 : & ([x \mapsto 2, y \mapsto 1], \emptyset) \rightsquigarrow ([x \mapsto 3], \kappa_2) \end{array} \right\}$$

A thread holding capability κ_1 can use it when r contains $x \mapsto 1$ to update it to $x \mapsto 2$ and introduce out of nowhere, a new resource $y \mapsto 1$, which was not previously held in the thread's local logical state. Moreover, if the obtained capability κ_2 is used by the same thread as part of the same atomic update to bring the state of the region r directly from $x \mapsto 1$ to $x \mapsto 3$ without one noticing the introduction of the *fictitious* resource y. We therefore have the strong constraint on the orthogonal preservation in order to cope with these kind of scenarios.

At this point, we are enabled to explain the fundamental notion of repartitioning with respect to an update from p to q, written $P \Rrightarrow^{\{p\}\{q\}} Q$ [12]. The latter indicates that any world w_1 satisfying the assertion P has a machine-state component $(w_1)_{\mathsf{M}}$ that includes a part, m_1, which satisfies the separation logic assertion p. Moreover, when m_1 is replaced by m_2 that satisfies assertion q, we are able to reconstruct a world w_2 that satisfies Q and for which the transition from w_1 to w_2 is allowed by the guarantee relation G.

Definition 4.28. (Repartitioning). The *repartioning* of worlds, written $P \Rrightarrow^{\{p\}\{q\}} Q$, holds if and only if, for every logical environment $e \in \mathsf{LEnv}$, predicate environment $\delta \in \mathsf{PEnv}$, and world $w_1 = (l_1, g_1, \mathcal{J}_1) \in \mathsf{World}$ such that $w_1, e, \delta \vDash P$, there exist states $m_1, m' \in \mathbb{M}$, such that for a $\rho_1 \in \mathbf{0}_{\mathsf{RK}}$:

- $(m_1, \rho_1), e \vDash_{\mathsf{SL}} p$ and

- $m_1 \bullet_{\mathbb{M}} m' = (l_1 \oplus g_1)_{\mathsf{M}}$ and
- for every $m_2 \in \mathbb{M}$ and $\rho_2 \in \mathbf{0}_{\mathsf{RK}}$ where $(m_2, \rho_2), e \vDash_{\mathsf{SL}} q$, there exists a world $w_2 = (l_2, g_2, \mathcal{J}_2) \in \mathsf{World}$ such that $w_2, e, \delta \vDash Q$ and
  - $m_2 \bullet_{\mathbb{M}} m' = (l_2 \oplus g_2)_{\mathsf{M}}$
  - $(w_1, w_2) \in G$

We write $P \Rrightarrow Q$ in order to express $P \Rrightarrow^{\{\texttt{emp}\}\{\texttt{emp}\}} Q$. The latter allows the shared state to be reorganized around but not to be effectively mutated.

4.5 Programming Language

We present here a simple concurrent imperative programming language that supports transactions and is parametric with respect to its low-level commands. It is modelled after the WHILE language and it is an instantiation of the language presented in the Views framework [11].

Definition 4.29. (Variables). The set of *variable* names is Var and it is ranged over by $\texttt{x}$, $\texttt{y}$, $\texttt{a}$, $\texttt{b}$, ...

Definition 4.30. (Numerical Expressions). The set of *numerical expressions*, Expr is ranged over by $\mathbb{E}, \mathbb{E}_1, \ldots, \mathbb{E}_n$. Numerical expressions are built from the following grammar, where $n \in \mathbb{Z}, \texttt{x} \in \mathsf{Var}$ and $x \in \mathsf{LVar}$.

$$\mathbb{E} ::= n \mid \texttt{x} \mid x \mid \mathbb{E}_1 + \mathbb{E}_2 \mid \mathbb{E}_1 - \mathbb{E}_2 \mid \mathbb{E}_1 \times \mathbb{E}_2 \mid \mathbb{E}_1 \div \mathbb{E}_2$$

Definition 4.31. (Boolean Expressions). The set of primitive boolean values is defined as $\mathsf{Bool} \triangleq \{\bot, \top\}$. The set of *boolean expressions* is BExpr and it is ranged over by $\mathbb{B}, \mathbb{B}_1, \ldots, \mathbb{B}_n$. Boolean expressions are generated from the following grammar:

$$\mathbb{B} ::= \texttt{true} \mid \texttt{false} \mid \mathbb{B}_1 \wedge \mathbb{B}_2 \mid \mathbb{B}_1 \vee \mathbb{B}_2 \mid \neg\mathbb{B} \mid \mathbb{E}_1 = \mathbb{E}_2 \mid \mathbb{E}_1 > \mathbb{E}_2$$

User instantiations of mCAP can specify the elementary commands of their language. These are the lowest-level building blocks on top of which we build sequential commands, transactions and later programs. One can see these as the standard variable assignment, memory dereference, etc in the separation logic world, but in reality, mCAP does not constraint them to be of a particular shape. It follows that a lot of flexibility is given in order to cover more context-specific language requirements.

Parameter 4.32. (Elementary commands). Assume a set of elementary commands, ECmd, ranged over by $\hat{\mathbb{C}}, \hat{\mathbb{C}}_1, \ldots, \hat{\mathbb{C}}_n$.

Definition 4.33. (Sequential commands). Cmd is the set of *sequential commands* which is ranged over by $\mathbb{C}, \mathbb{C}_1, \ldots, \mathbb{C}_n$. Sequential commands are defined by the following grammar:

$$\mathbb{C} ::= \hat{\mathbb{C}} \mid \texttt{skip} \mid \mathbb{C}_1; \mathbb{C}_2 \mid \texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2 \mid \texttt{while}\,(\mathbb{B})\ \mathbb{C}$$

Commands include the plain elementary commands provided by the framework user, the no-op skip action, sequential composition of commands, branching on a boolean condition and looping. Note that no paralellism is allowed as part of sequential commands. This follows from the fact that commands must appear as part of transactions and concurrent composition is only allowed at the level of programs.

Every transaction that is executed as part of a program is associated with a unique identifier (e.g. a number or a timestamp).

Parameter 4.34. (Transaction identifiers). Assume a set of *transaction identifiers*, Tid, ranged over by $\iota, i, j \ldots$ with an associated strict total order relation $<$, used to compare them.

We write a system transaction with identifier ι as $\mathbb{T}_\iota$. It is now time to distinguish between client and system transactions. Given that our setup will require to identify a particular transaction in order to associate to it a variety of information, system transactions are associated to a particular identifier. This burden is not passed on to the user of the framework, since identifiers are only added at runtime and not while writing programs. For this reason, we first give a grammar for user transactions that allows clients not to explicitly specify the unique identifier associated with each of them. Next, we specify another grammar for the system ones that carry an identifier.

Definition 4.35. (Transactions). The set of user transactions UTrans is ranged over by $\hat{\mathbb{T}}$ and represents how transactions are written by clients of the language. They are defined as follows:

$$\hat{\mathbb{T}} ::= \texttt{begin}\ \mathbb{C}\ \texttt{end}$$

The set of system transactions Trans is ranged over by $\mathbb{T}$ and defined using the following grammar, where $\iota \in \mathsf{Tid}$ is a transaction identifier.

$$\mathbb{T} ::= \hat{\mathbb{T}} \mid \texttt{begin}\ \mathbb{C}\ \texttt{end}_\iota$$

A transaction's body is defined as the command $\mathbb{C}$ inside of the transaction $\texttt{begin}\ \mathbb{C}\ \texttt{end}$ or $\texttt{begin}\ \mathbb{C}\ \texttt{end}_\iota$.

Programs are the top-level construct of our programming language and are allowed to be instatiated as a single system transaction $\mathbb{T}$, the sequential composition (;) of programs, their parallel composition through the use of the $\|$ operator, a non-deterministic choice $+$ between two programs and a nondeterministic loop of a program with *.

Definition 4.36. (Programs). Programs come from the set Prog which is ranged over by $\mathbb{P}, \mathbb{P}_1, \ldots, \mathbb{P}_n$ and it is defined using the following grammar:

$$\mathbb{P} ::= \texttt{skip} \mid \mathbb{T} \mid \mathbb{P}^* \mid \mathbb{P}_1 + \mathbb{P}_2 \mid \mathbb{P}_1; \mathbb{P}_2 \mid \mathbb{P}_1 \| \mathbb{P}_2$$

It is assumed that the parallel composition operator, $\|$, is both commutative and associative. This peculiarity is justified by the way such operation is treated in the Views operational semantics for programs. The parallel composition of programs, $\mathbb{P}_1 \| \mathbb{P}_2$, is in fact seen as a nondeterministic one-step reduction in one of $\mathbb{P}_1$ or $\mathbb{P}_2$. Therefore, the nondeterministic aspect enables us to determine the commutativity and associativity properties of the operator.

4.6 Proof System

The proof system that we illustrate here is for the programming language described in Section 4.5 and it is also an instantiation of the Views framework [11]. We start by parametrizing axioms for elementary commands, and later define the ones for sequential commands and transactions.

As a consequence of leaving elementary commands as a parameter to the user, we also require the provision of axioms for them. These build a relationship between sets of machine states, once affected by a particular command.

Parameter 4.37. (Elementary command axioms). Given the pcm for machine states $(\mathbb{M}, \bullet_\mathbb{M}, \mathbf{0}_\mathbb{M})$, assume a set of *elementary command axioms*, $\mathrm{Ax}_{\hat{\mathsf{C}}} : \mathcal{P}(\mathbb{M}) \times \mathsf{ECmd} \times \mathcal{P}(\mathbb{M})$.

A function for the semantic evaluation of boolean expressions under a given machine state is necessary to later build axioms for sequential commands. We must remember that the programming language defined in Section 4.5 allows boolean conditions to appear both as part of if-statements and of while loops, and the corresponding axioms will depend on how such conditions are evaluated.

Parameter 4.38. (Boolean machine state semantics). Given the pcm for machine states $(\mathbb{M}, \bullet_{\mathbb{M}}, \mathbf{0}_{\mathbb{M}})$, assume a *boolean semantics* function for machine states:

$$[\![-]\!]^{\text{B}}_{-} : \mathbb{B} \times \mathbb{M} \to \mathsf{Bool}$$

Definition 4.39. (Sequential command axioms). Given the axiomatisation of elementary commands $\text{Ax}_{\hat{\mathsf{C}}}$, the *sequential command axioms*, $\text{Ax}_{\mathsf{C}} : \mathcal{P}(\mathbb{M}) \times \mathsf{Cmd} \times \mathcal{P}(\mathbb{M})$, are defined in the following way:

$$\text{Ax}_{\mathsf{C}} \triangleq \text{Ax}_{\hat{\mathsf{C}}} \cup \text{Ax}_{\text{SKIP}} \cup \text{Ax}_{\text{SEQ}} \cup \text{Ax}_{\text{COND}} \cup \text{Ax}_{\text{LOOP}}$$

$$\text{Ax}_{\text{SKIP}} \triangleq \{(M, \texttt{skip}, M) \mid M \in \mathcal{P}(\mathbb{M})\}$$

$$\text{Ax}_{\text{SEQ}} \triangleq \{(M, \mathbb{C}_1; \mathbb{C}_2, M') \mid (M, \mathbb{C}_1, M'') \in \text{Ax}_{\mathsf{C}} \wedge (M'', \mathbb{C}_2, M') \in \text{Ax}_{\mathsf{C}}\}$$

$$\text{Ax}_{\text{COND}} \triangleq \left\{ (M, \texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2, M') \;\middle|\; \begin{array}{c} \forall m_t \in M.\, [\![\mathbb{B}]\!]^{\text{B}}_{m_t} = \top \wedge (M, \mathbb{C}_1, M') \in \text{Ax}_{\mathsf{C}} \\ \vee \\ \forall m_f \in M.\, [\![\mathbb{B}]\!]^{\text{B}}_{m_f} = \bot \wedge (M, \mathbb{C}_2, M') \in \text{Ax}_{\mathsf{C}} \end{array} \right\}$$

$$\text{Ax}_{\text{LOOP}} \triangleq \left\{ (M, \texttt{while}\,(\mathbb{B})\ \mathbb{C}, M) \;\middle|\; \begin{array}{c} \forall m \in M.\, [\![\mathbb{B}]\!]^{\text{B}}_{m} = \bot \\ \vee \\ \exists M'.\, M' \subseteq M \wedge \forall m \in M'.\, [\![\mathbb{B}]\!]^{\text{B}}_{m} = \top \\ \wedge\ (M', \mathbb{C}, M) \in \text{Ax}_{\mathsf{C}} \end{array} \right\}$$

where M, M' and M'' are used to quantify over elements of $\mathcal{P}(\mathbb{M})$.

Axioms for sequential commands relate machine states sets and are obtained by combining together axioms for every possible type of commands. In the case of an elementary command, we use the set provided as part of Parameter 4.37, while for `skip` any state in $\mathbb{M}$ is accepted and left untouched. As expected, axioms for sequential composition of two commands, $\mathbb{C}_1; \mathbb{C}_2$, are defined as the pair of sets of states M, M' where M, M'' and M'', M' are in the axioms for commands $\mathbb{C}_1$ and $\mathbb{C}_2$ respectively. On the other hand, in the occurrence of an if-statement with a condition $\mathbb{B}$, we find the sets M and M' by considering the case where either all the states in M evaluate $\mathbb{B}$ as $\top$, i.e. true, and therefore M' will be part of the axiom for $\mathbb{C}_1$, or they evaluate the condition as $\bot$ and M' is such that $(M, \mathbb{C}_2, M') \in \text{Ax}_{\mathsf{C}}$. Finally, axioms for the while loop command are of the form $(M, \texttt{while}\,(\mathbb{B})\ \mathbb{C}, M)$ where either $\mathbb{B}$ is semantically evaluated to false by all states in M, thus modelling an end of loop situation, or there is a subset of machine states M' which makes $\mathbb{B}$ true and the same states are in an axiom for $\mathbb{C}$ of the shape $(M', \mathbb{C}, M)$.

Definition 4.40. (Transaction axioms). Given the axiomatisation of sequential commands Ax_{C}, the set of *transaction axioms*, $\text{Ax}_{\mathsf{T}} : \mathcal{P}(\mathsf{World}) \times \mathsf{Trans} \times \mathcal{P}(\mathsf{World})$, is defined as:

$$\text{Ax}_{\mathsf{T}} \triangleq \left\{(W, \texttt{begin}\ \mathbb{C}\ \texttt{end}, W') \mid (M_1, \mathbb{C}, M_2) \in \text{Ax}_{\mathsf{C}} \wedge W \Rrightarrow^{\{M_1\}\{M_2\}} W'\right\}$$

where W and W' are used to quantify over elements of $\mathcal{P}(\mathsf{World})$.

Notice how the axioms for transactions make use of the axioms for their body $\mathbb{C}$ to obtain the necessary sets of machine state enabled by $\mathbb{C}$ which are then used as arguments to perform a repartitioning from worlds in W to the ones in W'. This particular relation enables us to cover all possible axioms for transactions.

At this point, once we have succesfully defined axioms for transactions, we can list all of the proof rules that support programs.

Definition 4.41. (Proof rules). All proof rules that follow, carry the implicit assumption that the preconditions and postconditions of their judgements are stable with respect to the environment.

$$\frac{}{\Delta \vdash \{P\}\ \texttt{skip}\ \{P\}}\ \textsc{Skip} \qquad \frac{\vdash_{\mathsf{SL}} \{p\}\ \mathbb{C}\ \{q\} \quad P \Rrightarrow^{\{p\}\{q\}} Q}{\Delta \vdash \{P\}\ \texttt{begin}\ \mathbb{C}\ \texttt{end}\ \{Q\}}\ \textsc{Trans}$$

$$\frac{\Delta \vdash \{P\}\ \mathbb{P}_1\ \{R\} \quad \Delta \vdash \{R\}\ \mathbb{P}_2\ \{Q\}}{\Delta \vdash \{P\}\ \mathbb{P}_1; \mathbb{P}_2\ \{Q\}}\ \textsc{Seq} \qquad \frac{\Delta \vdash \{P\}\ \mathbb{P}_1\ \{Q\} \quad \Delta \vdash \{P\}\ \mathbb{P}_2\ \{Q\}}{\Delta \vdash \{P\}\ \mathbb{P}_1 + \mathbb{P}_2\ \{Q\}}\ \textsc{Choice}$$

$$\frac{\Delta \vdash \{P\}\ \mathbb{P}\ \{Q\}}{\Delta \vdash \{P * R\}\ \mathbb{P}\ \{Q * R\}}\ \textsc{Frame} \qquad \frac{\Delta \vdash P \Rrightarrow P' \quad \Delta \vdash \{P'\}\ \mathbb{P}\ \{Q'\} \quad \Delta \vdash Q' \Rrightarrow Q}{\Delta \vdash \{P\}\ \mathbb{P}\ \{Q\}}\ \textsc{Conseq}$$

$$\frac{\Delta \vdash \{P_1\}\ \mathbb{P}_1\ \{Q_1\} \quad \Delta \vdash \{P_2\}\ \mathbb{P}_2\ \{Q_2\}}{\Delta \vdash \{P_1 * P_2\}\ \mathbb{P}_1 \| \mathbb{P}_2\ \{Q_1 * Q_2\}}\ \textsc{Par} \qquad \frac{\Delta \vdash \{P\}\ \mathbb{P}\ \{P\}}{\Delta \vdash \{P\}\ \mathbb{P}^*\ \{P\}}\ \textsc{Loop}$$

$$\frac{[\![\Delta]\!]^{\mathrm{P}} \subseteq [\![\Delta']\!]^{\mathrm{P}} \quad \Delta' \vdash \{P\}\ \mathbb{P}\ \{Q\}}{\Delta \vdash \{P\}\ \mathbb{P}\ \{Q\}}\ \textsc{Pred-I}$$

$$\frac{\forall \delta \in [\![\Delta]\!]^{\mathrm{P}}.\, \mathsf{stable}_\delta(R) \quad \alpha \notin \Delta, P, Q \quad \Delta, (\forall \vec{x}.\, \alpha(\vec{x}) \equiv R) \vdash \{P\}\ \mathbb{P}\ \{Q\}}{\Delta \vdash \{P\}\ \mathbb{P}\ \{Q\}}\ \textsc{Pred-E}$$

Most of the rules are standard from disjoint concurrent separation logic [19], with the exception of rules Pred-I and Pred-E which are instead derived from CAP [12]. The first one is used to weaken the assumptions about the predicate definitions, since if we are able to prove a triple with assumptions Δ' then it must be the case that we can prove the same triple under stronger assumptions Δ. On the other hand, the Pred-E proof rule allows the introduction of a new predicate definition. In order to do so, the new predicate must be stable with respect to the environment and its name must not be in the existing definitions or the triple's assertions.

As part of the Conseq rule, we use the repartitioning operator as presented in Definition 4.28 instead of the classic implication arrow $\implies$. The Trans rule occurs at the level of a single transaction $\mathbb{T}$, and makes sure that its command body $\mathbb{C}$ satisfies some separation logic level assertions that are able to repartition the world from $\mathbb{T}$'s pre-condition to its post-condition. The Frame rule works in the standard way, by enabling to add resources to both the pre-condition and the post-condition of a program, given that the latter does not modify them. This works since we must remember that all pre- and post-conditions are implicitly stable. Finally, the elegance of the Par rule in disjoint CSL is preserved, in line with CAP, given that we can run programs in parallel by separating the starting state and joining it at the end. Note that the composition of worlds is only defined when their shared state is equal, meaning that the actual separation that appears in the rule is with respect to the local state.

Given that the proof rules related to predicates, namely Pred-I and Pred-E, are not part of the ones included in the views framework, we show that they are sound in Lemma A.6 and Lemma A.9 respectively.

4.7 Semantics

The operational semantics of the programming language are defined in terms of the effects that elementary commands, sequential commands and transactions have on concrete program

states. These differ from the machine states that we introduced in Parameter 4.4, since they should describe *real* system states. As a matter of fact, concurrent program logics often enhance their states with abstract information that enables particular aspects of their reasoning. mCAP accepts any kind of artificial construct as part of its machine states under the condition that they can be mapped to concrete states on which commands can apply their effects. In Section 4.8 we will see how the user is able to specify such conversion.

Given the fact that we are defining an instantiation of the Views framework, the operational semantics described here follow the structure defined in [11] provided with the set of concrete states and the interpretation of transactions (i.e. atomic commands).

Parameter 4.42. (Concrete states). Assume a set of *concrete states* $\mathcal{S}$ ranged over by $\sigma, \sigma_1, \ldots, \sigma_n$.

We require instantiations of mCAP to additionally provide a function that interprets the semantics of boolean expressions at the level of concrete states. This is necessary since sequential commands support `if-then-else` and `while` and they both have a boolean expression as condition which needs to be evaluated in order to decide how to branch.

Parameter 4.43. (Boolean concrete state semantics). Given the the set of concrete states $\mathcal{S}$, assume a *boolean semantics* function:

$$[\![-]\!]^{\mathbb{B}}_{-} : \mathbb{B} \times \mathcal{S} \to \mathsf{Bool}$$

As elementary commands are given as a parameter to the mCAP framework, specifically Parameter 4.32, we also expect the user to provide a function that gives the operational semantics intepretation for each of them. The latter returns a potentially nondeterministic state transformer function which, given a concrete state σ, returns a set of those that arise as a consequence of applying the elementary command to σ.

Parameter 4.44. (Elementary command interpretation). Given the set of concrete states $\mathcal{S}$, assume an *elementary command interpretation* function associating elementary commands to a state-transformer function:

$$[\![\hat{\mathbb{C}}]\!]_{\hat{\mathsf{C}}} : \mathsf{ECmd} \to \mathcal{S} \to \mathcal{P}(\mathcal{S})$$

The interpretation function is also lifted to a set of concrete states, such that for $S \in \mathcal{P}(\mathcal{S})$:

$$[\![\hat{\mathbb{C}}]\!]_{\hat{\mathsf{C}}}(S) \triangleq \bigcup_{\sigma \in S} [\![\hat{\mathbb{C}}]\!]_{\hat{\mathsf{C}}}(\sigma)$$

Definition 4.45. (Commands interpretation). The interpretation function for elementary commands, formally $[\![-]\!]^{s}_{\mathsf{C}} : \mathsf{Cmd} \to \mathcal{S} \to \mathcal{P}(\mathcal{S})$, is defined as:

$$\begin{aligned}
[\![\hat{\mathbb{C}}]\!]_{\mathsf{C}}(\sigma) &\triangleq [\![\hat{\mathbb{C}}]\!]_{\hat{\mathsf{C}}}(\sigma) \\
[\![\texttt{skip}]\!]_{\mathsf{C}}(\sigma) &\triangleq \{\sigma\} \\
[\![\mathbb{C}_1; \mathbb{C}_2]\!]_{\mathsf{C}}(\sigma) &\triangleq \{\sigma' \mid S = [\![\mathbb{C}_1]\!]_{\mathsf{C}}(\sigma) \wedge \sigma' \in [\![\mathbb{C}_2]\!]_{\mathsf{C}}(S)\} \\
[\![\texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2]\!]_{\mathsf{C}}(\sigma) &\triangleq \textbf{if } [\![\mathbb{B}]\!]^{\mathbb{B}}_{\sigma} \textbf{ then } [\![\mathbb{C}_1]\!]_{\mathsf{C}}(\sigma) \textbf{ else } [\![\mathbb{C}_2]\!]_{\mathsf{C}}(\sigma) \\
[\![\texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}(\sigma) &\triangleq [\![\texttt{if}\,(\mathbb{B})\ \ (\mathbb{C}; \texttt{while}\,(\mathbb{B})\ \mathbb{C})\ \ \texttt{else skip}]\!]_{\mathsf{C}}(\sigma)
\end{aligned}$$

The interpretation function is lifted to a set of concrete states such that for $S \in \mathcal{P}(\mathcal{S})$:

$$[\![\mathbb{C}]\!]_{\mathsf{C}}(S) \triangleq \bigcup_{\sigma \in S} [\![\mathbb{C}]\!]_{\mathsf{C}}(\sigma)$$

In the case of an elementary command, we simply use Definition 4.44 while the interpretation of `skip` is the identity function on the input state σ. The effect of a sequential composition

of two commands is defined as the set of all states arising from the execution of the first command, followed by the second command applied to the resulting states. Loops are rewritten as conditionals, where the if body contains the loop body composed with the same loop. If-statements are instead applied to a state by branching on the boolean condition, semantically evaluated under the concrete state σ.

Definition 4.46. (Transactions interpretation). Finally, we can provide the interpretation function for transactions, which is formally defined as $[\![-]\!]_{\mathsf{T}} : \mathsf{Trans} \to \mathcal{S} \to \mathcal{P}(\mathcal{S})$,

$$[\![\texttt{begin } \mathbb{C} \texttt{ end}]\!]_{\mathsf{T}}(\sigma) \triangleq [\![\mathbb{C}]\!]_{\mathsf{C}}(\sigma)$$

The interpretation function is lifted to a set of concrete states such that for $S \in \mathcal{P}(\mathcal{S})$:

$$[\![\mathbb{T}]\!]_{\mathsf{T}}(S) \triangleq \bigcup_{\sigma \in S} [\![\mathbb{T}]\!]_{\mathsf{T}}(\sigma)$$

Therefore, the semantic meaning of running an atomic transaction on the concrete state σ, is any resulting state that is the outcome from executing the transaction's body $\mathbb{C}$, starting with the same state σ. On top of this, we also give a definition for the interpretation of system transactions. Given that these only differ from user transactions in terms of their identifier, and the operational semantics we define do not make use of such identifiers, we can simply drop the latter.

$$[\![\texttt{begin } \mathbb{C} \texttt{ end}_\iota]\!]_{\mathsf{T}}(\sigma) \triangleq [\![\texttt{begin } \mathbb{C} \texttt{ end}]\!]_{\mathsf{T}}(\sigma)$$

4.8 Soundness

The soundness of mCAP is established by relating its proof judgements to the operational semantics. The link is created through a *reification* function that transforms worlds, as defined in Section 4.2, to concrete machine states. As we constructed mCAP as an instantiation of the Views framework [11], its soundness follows from the soundness of Views, given that transactions (atomic commands) are shown to be sound with respect to their operational semantics.

Parameter 4.47. (Machine state reification). Given the partial commutative monoid for machine states $(\mathbb{M}, \bullet_{\mathbb{M}}, \mathbf{0}_{\mathbb{M}})$, assume a *reification function* $\lfloor - \rfloor_{\mathbb{M}} : \mathbb{M} \to \mathcal{P}(\mathcal{S})$ which associates machine states to sets of concrete ones.

The views framework guarantees soundness of an instantiated logic once we are able to prove its *Axiom Soundness* property, specifically **Property L** in [11]. This is formulated and proven in Theorem 4.51. Given the structure of the operational semantics for transactions, the proof relies on two other results, namely the axiom soundness of sequential commands and elementary commands. The latter is provided to the framework as a further parameter since elementary commands are themselves parametric.

Parameter 4.48. (Elementary command soundness). Given the pcm for machine states $(\mathbb{M}, \bullet_{\mathbb{M}}, \mathbf{0}_{\mathbb{M}})$ and its associated reification function $\lfloor - \rfloor_{\mathbb{M}}$, assume that for every elementary command $\hat{\mathbb{C}} \in \mathsf{ECmd}$, the corresponding axiom $(M_1, \hat{\mathbb{C}}, M_2) \in \mathrm{Ax}_{\hat{\mathsf{C}}}$ and any given machine state $m \in \mathbb{M}$ the following *soundness* property holds:

$$[\![\hat{\mathbb{C}}]\!](\lfloor M_1 \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}) \subseteq \lfloor M_2 \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}$$

Intuitively, the property establishes that, for any given axiom of elementary commands $(M_1, \hat{\mathbb{C}}, M_2)$, the operational semantics of $\hat{\mathbb{C}}$ are able to transform elements of the reificated machine states set M_1 into ones that belong to the reification of M_2. Moreover, the property also enforces frame preservation. In fact, it suggests that if we apply the command to the composition of the elements in M_1 with an arbitrary machine state m and then reify the result through $\lfloor - \rfloor_{\mathbb{M}}$, then the result will be a subset of the elements in $\lfloor M_2 \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}$, where the set M_2 is composed with the same m and the result is reified.

Definition 4.49. (Reification). The *reification of worlds*, $\lfloor - \rfloor_W : \mathsf{World} \to \mathcal{P}(\mathcal{S})$, is defined as follows.

$$\lfloor (l, g, \mathcal{J}) \rfloor_W \triangleq \lfloor (l \circ g)_{\mathsf{M}} \rfloor_{\mathbb{M}}$$

We transform worlds into machine states by dropping the action model, composing the local and the shared state together, and removing all of the region capabilities from it in order to end up only with the machine component of the resulting logical state.

Definition 4.50. (Judgements). The syntactic triple $\Delta \vdash \{P\}\ \mathbb{P}\ \{Q\}$ is defined in terms of the semantic triple as stated in Views [11] and it holds if and only if

$$\forall e, \delta.\, \delta \in [\![\Delta]\!]^{\mathrm{P}} \implies \vDash \left\{[\![P]\!]_{e,\delta}\right\}\ \mathbb{P}\ \left\{[\![Q]\!]_{e,\delta}\right\}$$

In a comparable way to elementary commands, the soundness of transactions is proven by considering an arbitrary transactions' axiom $(W_1, \mathbb{T}, W_2)$, built according to Definition 4.40, together with a world w. It is required to show that the operational semantics of $\mathbb{T}$, when applied to elements of the reification of $W_1 \bullet_{\mathbb{M}} \{w\}$, output a concrete state which is a subset of $\lfloor W_1 \bullet_{\mathbb{M}} R(\{w\}) \rfloor_{\mathbb{M}}$. Not only we require the semantics to correctly work in terms of what the axioms impose, but also to preserve frames even under the effects of the rely relation. The property in fact ensures that any additional world component w, not needed by $\mathbb{T}$ (the axiom does not describe it), will remain unchanged upto its rely relation evolutions.

Theorem 4.51. (Transaction soundness). For all $\mathbb{T} \in \mathsf{Trans}, (W_1, \mathbb{T}, W_2) \in \mathrm{Ax}_{\mathsf{T}}$ and $w \in \mathsf{World}$:

$$[\![\mathbb{T}]\!]_{\mathsf{T}}(\lfloor W_1 \bullet_{\mathbb{M}} \{w\} \rfloor_W) \subseteq \lfloor W_2 \bullet_{\mathbb{M}} R(\{w\}) \rfloor_W$$

Proof. By induction on the structure of $\mathbb{T}$.

Case: `begin` $\mathbb{C}$ `end`
Let's pick an arbitrary $\mathbb{C} \in \mathsf{Cmd}, w \in \mathsf{World}$ and $W_1, W_2 \in \mathcal{P}(\mathsf{World})$ such that the following holds:

$$(W_1, \texttt{begin}\ \mathbb{C}\ \texttt{end}, W_2) \in \mathrm{Ax}_{\mathsf{T}}$$

From the definition of Ax_{T} we know that there exists $M_1, M_2 \in \mathcal{P}(\mathbb{M})$ such that:

$$(M_1, \mathbb{C}, M_2) \in \mathrm{Ax}_{\mathsf{C}} \wedge W_1 \Rrightarrow^{\{M_1\}\{M_2\}} W_2 \tag{1}$$

To show:

$$[\![\texttt{begin}\ \mathbb{C}\ \texttt{end}]\!]_{\mathsf{T}}(\lfloor W_1 \bullet \{w\} \rfloor_W) \subseteq \lfloor W_2 \bullet R(\{w\}) \rfloor_W$$

Let's pick an arbitrary $w_1 = (l_1, g_1, \mathcal{J}_1) \in W_1$. We are now left to show that there exists a $w_2 \in \mathsf{World}$ and $w' \in R(w)$ such that:

$$[\![\texttt{begin}\ \mathbb{C}\ \texttt{end}]\!]_{\mathsf{T}}(\lfloor w_1 \bullet w \rfloor_W) = \lfloor w_2 \bullet w' \rfloor_W$$

From the definition of $[\![-]\!]_{\mathsf{T}}, \lfloor - \rfloor_W$, and the properties of $\bullet$ and $\bullet_{\mathbb{M}}$ we have:

$$\begin{aligned} [\![\texttt{begin}\ \mathbb{C}\ \texttt{end}]\!]_{\mathsf{T}}(\lfloor w_1 \bullet w \rfloor_W) &= [\![\mathbb{C}]\!]_{\mathsf{C}}(\lfloor w_1 \bullet w \rfloor_W) & (2)\\ &= [\![\mathbb{C}]\!]_{\mathsf{C}}\left(\lfloor (l_1 \oplus g_1)_{\mathsf{M}} \bullet_{\mathbb{M}} (w_{\mathsf{L}})_{\mathsf{M}} \rfloor_{\mathbb{M}}\right) & (3) \end{aligned}$$

From (1) and the definition of $\Rrightarrow$ we know there exists $m_1 \in M_1$ and $m' \in \mathbb{M}$ such that:

$$\begin{aligned} m_1 \bullet_{\mathbb{M}} m' = (l_1 \oplus g_1)_{\mathsf{M}} \wedge \quad & (4)\\ \forall m_2 \in M_2.\, \exists w_2 = (l_2, g_2, \mathcal{J}_2) \in W_2.\, m_2 \bullet_{\mathbb{M}} m' = (l_2 \oplus g_2)_{\mathsf{M}} \wedge (w_1, w_2) \in G \quad & (5) \end{aligned}$$

From (3) and (4) we get:

$$[\![\texttt{begin}\ \mathbb{C}\ \texttt{end}]\!]_{\mathsf{T}}(\lfloor w_1 \bullet w \rfloor_W) = [\![\mathbb{C}]\!]_{\mathsf{C}}(\lfloor m_1 \bullet_{\mathbb{M}} m' \bullet_{\mathbb{M}} (w_{\mathsf{L}})_{\mathsf{M}} \rfloor_{\mathbb{M}}) \tag{6}$$

From (1) and the soundness of commands shown in Theorem A.1, we can rewrite (6) as:

$$[\![\texttt{begin}\ \mathbb{C}\ \texttt{end}]\!]_{\mathsf{T}}(\lfloor w_1 \bullet w \rfloor_W) \subseteq \lfloor M_2 \bullet_{\mathbb{M}} \{m' \bullet_{\mathbb{M}} (w_{\mathsf{L}})_{\mathsf{M}}\} \rfloor_{\mathbb{M}}$$

Which means that there exists a $m_2 \in M_2$ such that:

$$[\![\texttt{begin}\ \mathbb{C}\ \texttt{end}]\!]_{\mathsf{T}}(\lfloor w_1 \bullet w \rfloor_W) = \lfloor m_2 \bullet_{\mathbb{M}} m' \bullet_{\mathbb{M}} (w_{\mathsf{L}})_{\mathsf{M}} \rfloor_{\mathbb{M}} \tag{7}$$

From (5) we know that there exists $w_2 \in \mathsf{World}$ such that:

$$w_2 = (l_2, g_2, \mathcal{J}_2) \in W_2 \wedge m_2 \bullet_{\mathbb{M}} m' = (l_2 \oplus g_2)_{\mathsf{M}} \tag{8}$$

$$\wedge\ (w_1, w_2) \in G \tag{9}$$

From the definition of $\lfloor - \rfloor_W$ and the properties of $\bullet_{\mathbb{M}}$ and $\bullet$ we can rewrite (7) as:

$$[\![\texttt{begin}\ \mathbb{C}\ \texttt{end}]\!]_{\mathsf{T}}(\lfloor w_1 \bullet w \rfloor_W) = \lfloor (l_2 \circ w_{\mathsf{L}}, g_2, \mathcal{J}_2) \rfloor_W \tag{10}$$

From (9) and Lemma A.2 we know that there exists a $w' \in \mathsf{World}$ such that:

$$w' = (w_{\mathsf{L}}, g_2, \mathcal{J}_2) \wedge w' \in R(w) \tag{11}$$

From (8), (10) and (11) we know that there exists a $w_2 \in W_2$ and $w' \in R(w)$ such that:

$$[\![\texttt{begin}\ \mathbb{C}\ \texttt{end}]\!]_{\mathsf{T}}(\lfloor w_1 \bullet w \rfloor_W) = \lfloor w_2 \bullet R(w) \rfloor_W$$

□

Theorem 4.52. (Soundness). For all predicate axioms $\Delta \in \mathsf{Ax}$, assertions $P, Q \in \mathsf{Assn}$ and program $\mathbb{P} \in \mathsf{Prog}$, if $\Delta \vdash \{P\}\ \mathbb{P}\ \{Q\}$ then $\Delta \vDash \{P\}\ \mathbb{P}\ \{Q\}$.

5. A Logic for Serializable Transactions

The path towards a general logic for serializable transactions reaches its final stage. At this point, we can make effective use of the mCAP framework in order to formalise what we need. Given that the mCAP's model and program logic, described in Section 4 and Section 4.3 respectively, contain several parameters, we proceed to their instantiation for our particular requirements. In fact, transactions need to be able to access a global storage through read and write operations as well as keeping a local stack for their private variables.

By the end of this section, users will be able to verify programs containing transactions that are executed in an truly atomic fashion and in isolation. Later on, we take the logic one step further, by specifying the first application to programs that run under the two-phase locking protocol.

5.1 Model

The abstract model that describes the ecosystem under which transactions run as part of programs is illustrated in this section. At the core of the model resides a concrete, low-level storage, which can be thought of as a contiguous space of memory cells or as a collection of database items. Each of the latter contains a single numerical value and is associated with a unique address, or key, used to index it. Transactions access the storage to read from and permanently write to it.

Definition 5.1. (Storage values). The set of *storage values*, Val, is defined to be equivalent to the set of integers $\mathbb{Z}$. It is ranged over by $v, v_1, \dots v_n$.

Definition 5.2. (Storage keys). The set of *keys* used to index elements in the storage is defined as Key, ranged over by $k, k_1, \dots k_n$ and it is equivalent to the set of natural numbers $\mathbb{N}$.

Definition 5.3. (Storage). A *storage* is defined to be the set of partial functions with a finite domain from storage keys to storage values:

$$\mathsf{Storage} \triangleq \mathsf{Key} \xrightarrow{\text{fin}} \mathsf{Val}$$

The $\mathsf{Storage}$ set is ranged over by $h, h_1, \dots, h_n$. Let $\mathsf{State} \triangleq \mathsf{Storage} \uplus \{\lightning\}$ be the set containing all of the possible members of $\mathsf{Storage}$ together with a special, *faulting*, state $\lightning$ which indicates a program state where a failure happened.

Local variables are privately used by transactions inside of their bodies. They can be arbitrarily initialized, assigned and read and are recorded as part of a transaction's stack.

Definition 5.4. (Transaction stack). The set of *transaction stacks* is defined as the set of partial functions with finite domain, mapping variable names to their respective value:

$$\mathsf{Stack} \triangleq \mathsf{Var} \xrightarrow{\text{fin}} \mathsf{Val}$$

The Stack set is ranged over by $s, s_1, \dots, s_n$.

The first and fundamental parameter to provide to mCAP is the one related to machine states. This describes the machine component of every logical state in the model. Considering that we treat transactions as atomic blocks of code that carry a local variable store and can access a global storage, we model the machine state monoid as tuples of elements coming from $\mathsf{Storage}$ and Stack. It follows that every transaction's view on the world is based on those two components which fully describe what it can interact with.

Definition 5.5. (Machine states pcm). The partial commutative monoid with multiple units for machine states is instantiated to:

$$(\mathsf{Storage} \times \mathsf{Stack}, \uplus_2, \{(\emptyset, \emptyset)\})$$

where the union of tuples of partial functions with disjoint domains, $\uplus_2$, is defined as:

$$(h, s) \uplus_2 (h', s') \triangleq (h \uplus h', s \uplus s')$$

The unit set of the machine state pcm we provide contains a single element, which is a tuple of two functions with an empty domain, therefore not mapping any storage locations or local variable names to values.

Even within our instantiation of mCAP, we leave the partial commutative monoid for capabilities as a parameter, in order not to constrain users of the latter and ease context-specific reasoning.

We provide semantics for all the numerical expressions that appear as part of our programs in a standard way. The logical environment e and variable stack s components, are used to evaluate logical and program variables respectively. In the case where an expression continas a variable (either logical or program) which is not in the domain of the appropriate mapping, we adopt the convention of evaluating it to 0. A similar convention is established by returning the constant value 1 in the occurrence of an arithmetic division expression ($\div$) with a zero denominator.

Definition 5.6. (Numerical expression semantics). The semantics of numerical expressions are expressed through a function, $[\![-]\!]^{\mathrm{E}}_{-,-} : \mathsf{Expr} \times \mathsf{LEnv} \times \mathsf{Stack} \to \mathsf{Val}$, defined as:

$$
\begin{aligned}
[\![v]\!]^{\mathrm{E}}_{e,s} &\triangleq v \\
[\![\mathtt{x}]\!]^{\mathrm{E}}_{e,s} &\triangleq \begin{cases} s(\mathtt{x}), & \text{if } \mathtt{x} \in \mathsf{dom}(s) \\ 0, & \text{otherwise} \end{cases} \\
[\![x]\!]^{\mathrm{E}}_{e,s} &\triangleq \begin{cases} e(x), & \text{if } x \in \mathsf{dom}(e) \\ 0, & \text{otherwise} \end{cases} \\
[\![\mathbb{E}_1 + \mathbb{E}_2]\!]^{\mathrm{E}}_{e,s} &\triangleq [\![\mathbb{E}_1]\!]^{\mathrm{E}}_{e,s} + [\![\mathbb{E}_2]\!]^{\mathrm{E}}_{e,s} \\
[\![\mathbb{E}_1 - \mathbb{E}_2]\!]^{\mathrm{E}}_{e,s} &\triangleq [\![\mathbb{E}_1]\!]^{\mathrm{E}}_{e,s} - [\![\mathbb{E}_2]\!]^{\mathrm{E}}_{e,s} \\
[\![\mathbb{E}_1 \times \mathbb{E}_2]\!]^{\mathrm{E}}_{e,s} &\triangleq [\![\mathbb{E}_1]\!]^{\mathrm{E}}_{e,s} \times [\![\mathbb{E}_2]\!]^{\mathrm{E}}_{e,s} \\
[\![\mathbb{E}_1 \div \mathbb{E}_2]\!]^{\mathrm{E}}_{e,s} &\triangleq \begin{cases} 1, & \text{if } [\![\mathbb{E}_2]\!]^{\mathrm{E}}_{e,s} = 0 \\ [\![\mathbb{E}_1]\!]^{\mathrm{E}}_{e,s} \div [\![\mathbb{E}_2]\!]^{\mathrm{E}}_{e,s}, & \text{otherwise} \end{cases}
\end{aligned}
$$

Whenever we encounter a use of the semantics expression that only provides a Stack component without the logical environment one, i.e. $[\![\mathbb{E}]\!]^{\mathrm{E}}_{s}$, we implicitly refer to $[\![\mathbb{E}]\!]^{\mathrm{E}}_{\emptyset,s}$.

Boolean expressions that appear in programs are also evaluated in a classical way, by recursively evaluating the sub-expressions until we reach terminal ones, i.e. `true` or `false`, and then logically combining all of the intermediate steps.

Definition 5.7. (Boolean expression semantics). The semantics of boolean expressions are expressed through a function, $[\![-]\!]^{\mathrm{B}}_{-} : \mathsf{BExpr} \to (\mathsf{Storage} \times \mathsf{Stack}) \to \mathsf{Bool}$, defined as:

$$
\begin{aligned}
[\![\texttt{true}]\!]^{\mathrm{B}}_{(h,s)} &\triangleq \top \\
[\![\texttt{false}]\!]^{\mathrm{B}}_{(h,s)} &\triangleq \bot \\
[\![\mathbb{B}_1 \wedge \mathbb{B}_2]\!]^{\mathrm{B}}_{(h,s)} &\triangleq [\![\mathbb{B}_1]\!]^{\mathrm{B}}_{(h,s)} \wedge [\![\mathbb{B}_2]\!]^{\mathrm{B}}_{(h,s)} \\
[\![\mathbb{B}_1 \vee \mathbb{B}_2]\!]^{\mathrm{B}}_{(h,s)} &\triangleq [\![\mathbb{B}_1]\!]^{\mathrm{B}}_{(h,s)} \vee [\![\mathbb{B}_2]\!]^{\mathrm{B}}_{(h,s)} \\
[\![\neg\mathbb{B}]\!]^{\mathrm{B}}_{(h,s)} &\triangleq \neg[\![\mathbb{B}]\!]^{\mathrm{B}}_{(h,s)} \\
[\![\mathbb{E}_1 = \mathbb{E}_2]\!]^{\mathrm{B}}_{(h,s)} &\triangleq [\![\mathbb{E}_1]\!]^{\mathrm{E}}_{s} = [\![\mathbb{E}_2]\!]^{\mathrm{E}}_{s} \\
[\![\mathbb{E}_1 > \mathbb{E}_2]\!]^{\mathrm{B}}_{(h,s)} &\triangleq [\![\mathbb{E}_1]\!]^{\mathrm{E}}_{s} > [\![\mathbb{E}_2]\!]^{\mathrm{E}}_{s} \\
[\![\mathbb{E}_1 < \mathbb{E}_2]\!]^{\mathrm{B}}_{(h,s)} &\triangleq [\![\mathbb{E}_1]\!]^{\mathrm{E}}_{s} < [\![\mathbb{E}_2]\!]^{\mathrm{E}}_{s}
\end{aligned}
$$

Note that we only provide one function to cover the semantics of boolean expressions appearing inside machine states (Parameter 4.38) and concrete states (Parameter 4.43) because we instantiate the two to the same set, with the only difference of the $\lightning$ state.

5.2 Assertions

The instantiated machine states are described through assertions equivalent to the standard separation logic ones [24]. We leave a grammar for capability assertions as a parameter, following our choice of not instantiating a capability pcm in Section 5.1.

Definition 5.8. (Machine state assertions). Machine state assertions are instantiated to be able to describe a logical or program variable in the local stack and a cell in the storage.

$$\mathcal{M} \in \mathsf{MAssn} ::= x = \mathbb{E} \mid \mathtt{x} = \mathbb{E} \mid \mathbb{E}_1 \mapsto \mathbb{E}_2$$

The associated semantics function, which maps such assertions to elements of the machine state pcm, $[\![-]\!]^{\mathrm{M}}_{-} : \mathsf{MAssn} \to \mathsf{LEnv} \to \mathcal{P}(\mathbb{M})$, is defined as:

$$
\begin{aligned}
[\![x = \mathbb{E}]\!]^{\mathrm{M}}_{e} &\triangleq \big\{(h, s) \mid h \in \mathsf{Storage} \wedge s \in \mathsf{Stack} \wedge e(x) = [\![\mathbb{E}]\!]^{\mathrm{E}}_{e,s}\big\} \\
[\![\mathtt{x} = \mathbb{E}]\!]^{\mathrm{M}}_{e} &\triangleq \big\{(h, s) \mid h \in \mathsf{Storage} \wedge s(\mathtt{x}) = [\![\mathbb{E}]\!]^{\mathrm{E}}_{e,s}\big\} \\
[\![\mathbb{E}_1 \mapsto \mathbb{E}_2]\!]^{\mathrm{M}}_{e} &\triangleq \big\{(h, s) \mid \mathsf{dom}(h) = \{[\![\mathbb{E}_1]\!]^{\mathrm{E}}_{e,s}\} \wedge h([\![\mathbb{E}_1]\!]^{\mathrm{E}}_{e,s}) = [\![\mathbb{E}_2]\!]^{\mathrm{E}}_{e,s}\big\}
\end{aligned}
$$

Next, we formalise the elementary commands that are used as part of system transactions' bodies. These are *truly* atomic commands in the system. The commands allow a transaction to make a variable assignment in order to associate a numerical value to a local program variable, read a value from the global storage into a local variable, write a numerical expression to a storage cell and allocate an arbitrary amount of cells to extend the domain of the storage. The syntactic grammar used is the standard one from the WHILE language.

Definition 5.9. (Elementary commands). The set of *elementary commands*, ECmd, is instantiated through the following grammar, where $\mathtt{x} \in \mathsf{Var}$.

$$\hat{\mathbb{C}} ::= \mathtt{x} := \mathbb{E} \mid \mathtt{x} := [\mathbb{E}] \mid \mathtt{x} := \mathtt{alloc}(\mathbb{E}) \mid [\mathbb{E}_1] := \mathbb{E}_2$$

Definition 5.10. (Elementary command axioms). The set of *elementary command axioms* $\mathrm{Ax}_{\hat{\mathsf{C}}} : \mathcal{P}(\mathsf{Storage} \times \mathsf{Stack}) \times \mathsf{Cmd} \times \mathcal{P}(\mathsf{Storage} \times \mathsf{Stack})$ are defined to be the standard separation logic ones, describing the heap (our storage) and the variable store, i.e. a transaction's stack.

5.3 Semantics

The operational semantics component of our mCAP instantiation are defined in terms of effects to a *concrete* state of the system. This represents the true contents of the machine state, without any artificial construct introduced to aid reasoning. Given that we did not make use of any extra information as part of the machine states defined in Section 5.1, we will instantiate concrete states to be equivalent to the machine ones. On top of this, we add a special state element, the faulting state. This is a wildcard with the precise purpose of representing all states where something went *wrong*. We write $\lightning$ to refer to it.

Definition 5.11. (Concrete states). The set of *concrete states* $\mathcal{S}$ is instantiated to $(\mathsf{Storage} \times \mathsf{Stack}) \uplus \{\lightning\}$, where $\lightning$ represents a faulting state..

At this point we can provide a semantics interpretation function that formally describes how elements of ECmd affect the concrete state both in terms of its storage and variable stack component.

Definition 5.12. (Elementary commands interpretation). The *interpretation function* for elementary commands, $[\![-]\!]_{\hat{\mathsf{C}}} : \mathsf{ECmd} \to \mathcal{S} \to \mathcal{P}(\mathcal{S})$, is defined in the following way:

$$[\![\mathtt{x} := \mathbb{E}]\!]_{\hat{\mathsf{C}}}(h, s) \triangleq \{(h, s[\mathtt{x} \mapsto [\![\mathbb{E}]\!]^{\mathrm{E}}_s])\}$$

$$[\![\mathtt{x} := \mathtt{alloc}(\mathbb{E})]\!]_{\hat{\mathsf{C}}}(h, s) \triangleq \begin{cases} \{(h[\vec{a} \mapsto 0], s[\mathtt{x} \mapsto l])\} & \text{if } n = [\![\mathbb{E}]\!]^{\mathrm{E}}_s \text{ and } n > 0 \text{ and } l \in \mathsf{Key} \\ & \text{and } \vec{a} = (l, \ldots, l + n - 1) \text{ and} \\ & \{l, \ldots, l + n - 1\} \cap \mathsf{dom}(h) \equiv \emptyset \\ \{\lightning\}, & \text{otherwise} \end{cases}$$

$$[\![\mathtt{x} := [\mathbb{E}]]\!]_{\hat{\mathsf{C}}}(h, s) \triangleq \begin{cases} \{(h, s[\mathtt{x} \mapsto v])\}, & \text{if } k = [\![\mathbb{E}]\!]^{\mathrm{E}}_s \text{ and } k \in \mathsf{dom}(h) \\ & \text{and } h(k) = v \\ \{\lightning\}, & \text{otherwise} \end{cases}$$

$$[\![[\mathbb{E}_1] := \mathbb{E}_2]\!]_{\hat{\mathsf{C}}}(h, s) \triangleq \begin{cases} \{(h[k \mapsto v], s)\}, & \text{if } k = [\![\mathbb{E}_1]\!]^{\mathrm{E}}_s \text{ and } v = [\![\mathbb{E}_2]\!]^{\mathrm{E}}_s \\ & \text{and } k \in \mathsf{dom}(h) \\ \{\lightning\}, & \text{otherwise} \end{cases}$$

A variable assignment to `x` only updates the local variable stack to map variable `x` to the evaluated expression $\mathbb{E}$ under s, leaving the heap component unmodified. A memory allocation succeeds when it is possible to expand the current heap h to hold a sequence of n cells that are currently not in its domain. All fresh storage cells will be mapped to the default value of 0, and the address of the first allocated cell is assigned to variable `x` as part of the variable stack. In all other cases, the command fails. A memory dereference succeeds in reading a value v associated to key k only when k is in the heap's domain and the value mapped to it is effectively v. Last, we can mutate memory at location k only when it is part of the domain of the heap h. For both commands, in the event of a storage access whose corresponding key does not exist, we end up in $\lightning$, the faulting state.

We move to the final parameter required by mCAP for its instantiations, namely the reification function that transforms machine states into concrete ones.

Definition 5.13. (Machine state reification). The machine state *reification function* $\lfloor - \rfloor_{\mathbb{M}} : \mathbb{M} \to \mathcal{P}(\mathcal{S})$ is defined as:

$$\lfloor m \rfloor_{\mathbb{M}} \triangleq \{m\}$$

Given that the concrete states set, $\mathcal{S}$, is defined to be $(\mathsf{Storage} \times \mathsf{Stack}) \uplus \{\lightning\}$, and $\mathbb{M}$ is instantiated to $\mathsf{Storage} \times \mathsf{Stack}$ we conclude that $\mathbb{M} \subset \mathcal{S}$. It follows that establishing a relationship between the two sets resolves to just using the identity function.

6. Two-Phase Locking Semantics

The logic for serializable transactions is now applied to the specific context of two-phase locking, given that the latter guarantees the serializability of its executions. This step allows to reason about environments where concurrent transactions do not run in full isolation, but are effectively interleaved and therefore require a mechanism to coordinate their actions. Our plan is to introduce all of the necessary components that are able to model the protocol.

In fact, in order to formally describe the behaviours that arise from adopting 2PL as a concurrency control protocol for transactions, we extend the model presented in Section 5.1, by adding two new global constructs together with a way to keep track of the locking phase each transaction is currently in. These changes are necessary, since we need to enable a real interleaving of concurrent transactions at the level of single operations, and to introduce the concept of locking storage items with the goal of synchronizing transactions.

Later on, we give meaning to program executions in this new environment by specifying a set of small-step operational semantics rules, which are defined as a labelled transition system. They are formulated in a way which enforces the two-phase rule dictated by the protocol.

6.1 Model

Under the 2PL protocol, all accesses to the shared storage cells must be protected by a corresponding lock. This means that any transaction that needs to read or write a particular cell must be granted an appropriate lock for that same cell. The abstract locks we use, behave in a standard way: depending on the mode, they can have zero, one or more *owners*, i.e. threads that have been granted access to the protected cells.

Definition 6.1. (Lock modes). The set of *lock modes*, Lock, is ranged over by $\kappa, \kappa_1, \ldots, \kappa_n$ and defined as:

$$\mathsf{Lock} \triangleq \{\textsc{u}, \textsc{s}, \textsc{x}\}$$

The associated strict total order, $>$, is defined in the following way:

$$> \triangleq \{(\textsc{x}, \textsc{u}), (\textsc{x}, \textsc{s}), (\textsc{s}, \textsc{u})\}$$

The total order $\geq$ on the set Lock is equivalent to $>$ unioned with all of the reflexive pairs.

We informally refer to each of the lock mode entries as *unlocked*, *shared* and *exclusive* respectively. This reflects the fact that either no transaction is accessing a cell, one or more transactions are allowed to read the cell's content or a single transaction has been given the permission to write to the cell.

The 2PL protocol sets a precise constraint on the pattern of acquisition and release of locks. A transaction $\mathbb{T}$'s lifecycle is clearly distinguished between two phases. It initially starts executing in the *growing* phase ($\curlywedge$), where it is free to sequentially acquire locks for any cell it needs. Once it releases one of the locks it is holding, the transaction enters the *shrinking* phase ($\curlyvee$). Here $\mathbb{T}$ is denied any new lock acquisitions while it is allowed to gradually release the locks that are still being held.

Definition 6.2. (Locking phase). The set of *locking phases*, ranged over by p, is defined as:

$$\mathsf{Phase} \triangleq \{\curlywedge, \curlyvee\}$$

We sometimes refer to the growing and shirinking phase using *acquiring* and *releasing* phase respectively.

At this point we have all of the ingredients to introduce the other main component of our model, the lock manager. This global structure records the status of all locks on storage cells.

Definition 6.3. (Lock manager). A *lock manager* is defined as a partial function with a finite domain from storage keys to pairs of transaction identifiers and lock modes.

$$\mathsf{LMan} \triangleq \mathsf{Key} \overset{\text{fin}}{\rightharpoonup} \mathcal{P}(\mathsf{Tid}) \times \mathsf{Lock}$$

The LMan set is ranged over by $\Phi, \Phi_1, \ldots, \Phi_n$. In order to cope with the default state of locks associated to keys initially absent from the domain of a lock manager, a total function, $\hat{-}(-) : \mathsf{LMan} \times \mathsf{Key} \rightarrow \mathcal{P}(\mathsf{Tid}) \times \mathsf{Lock}$, is defined as:

$$\hat{\Phi}(k) \triangleq \begin{cases} \Phi(k), & \text{if } k \in \mathsf{dom}(\Phi) \\ (\emptyset, \text{U}), & \text{otherwise} \end{cases}$$

In fact, every cell starts in the unlocked state with no transaction owning it. We say that a lock manager is *empty* and write it as $\Phi = \emptyset$, if and only if all of the keys in its domain are mapped to locks with no owners. Formally:

$$\Phi = \emptyset \iff \forall k \in \mathsf{dom}(\Phi).\, (\emptyset, \text{U}) = \Phi(k)$$

We keep track of each transaction's stack and locking phase inside of another global structure which we call the transactions state.

Definition 6.4. (Transactions state). The *state* of all transactions running as part of a program is defined as a partial function with a finite domain:

$$\mathsf{TState} \triangleq \mathsf{Tid} \overset{\text{fin}}{\rightharpoonup} \mathsf{Stack} \times \mathsf{Phase}$$

Elements of TState are ranged over by $S, S_1, \ldots, S_n$. Another total function is also defined, $\hat{-}(-) : \mathsf{TState} \times \mathsf{Tid} \rightarrow \mathsf{Stack} \times \mathsf{Phase}$. Such function overrides the original function lookup in order to cope with newly created transactions with an empty stack starting in the growing phase.

$$\hat{S}(\iota) \triangleq \begin{cases} S(\iota), & \text{if } \iota \in \mathsf{dom}(S) \\ (\emptyset, \curlywedge), & \text{otherwise} \end{cases}$$

6.2 Operational Semantics

The formal behaviour of transactional programs' executions that are controlled by the 2PL protocol, is expressed through a set of operational semantics rules which are able to reduce a given program under a particular program state. Each program transition is tagged with an action label that describes its effects in terms of the global state components, i.e. the storage and the lock manager.

Definition 6.5. (Action labels). Atomic actions performed by transactions are represented using transition *action labels* from the set Act, ranged over by $\alpha, \alpha_1, \ldots, \alpha_n$ and defined with the following grammar:

$$\begin{aligned} \alpha \in \mathsf{Act} ::= {} & \mathsf{sys} \mid \mathsf{id}(\iota) \mid \mathsf{alloc}(\iota, n, l) \mid \mathsf{read}(\iota, k, v) \mid \mathsf{write}(\iota, k, v) \\ & \mid \mathsf{lock}(\iota, k, \kappa) \mid \mathsf{unlock}(\iota, k) \end{aligned}$$

where $k, l \in \mathsf{Key}, n \in \mathbb{N}, v \in \mathsf{Val}, \kappa \in \mathsf{Lock}, \iota \in \mathsf{Tid}$.

System transitions are labelled with sys and describe program-level execution. For this reason they are not part of any transaction. Control flow commands, that happen at the level of transactions (e.g. conditionals, loops), are labelled through $\mathsf{id}(\iota)$, where ι is the identifier of the transaction performing the action. A transaction ι reads a particular value v from cell indexed with k via a transition whose label is $\mathsf{read}(\iota, k, v)$ while similarly it can write a value v to k through $\mathsf{write}(\iota, k, v)$. Cells are allocated with transitions labelled with $\mathsf{alloc}(\iota, n, l)$ where l is the storage address from which a sequence of n fresh consecutive cells starts. Last, lock and unlock actions do not arise explicitly from syntactic commands as per Definition 5.9, but instead have their respective labels: $\mathsf{lock}(\iota, k, \kappa)$ for transaction ι to lock cell k under mode κ and $\mathsf{unlock}(\iota, k)$ for the same transaction to release any kind of lock held on k.

Formal semantics for action labels are provided through an interpretation function that, given an action, returns a global state transformer function that maps a storage and a lock manager to an updated version resulting from the label execution.

Definition 6.6. (Label interpretation). The *label interpretation* is given as a function from action labels to a global state transformer.

$$[\![-]\!] : \mathsf{Act} \to \mathsf{Storage} \times \mathsf{LMan} \to \mathsf{State} \times \mathsf{LMan}$$

$$[\![\mathsf{sys}]\!] \triangleq \lambda h, \Phi.\, (h, \Phi)$$

$$[\![\mathsf{id}(\iota)]\!] \triangleq \lambda h, \Phi.\, (h, \Phi)$$

$$[\![\mathsf{alloc}(\iota, n, l)]\!] \triangleq \lambda h, \Phi. \begin{cases} (h[\vec{a} \mapsto 0], \Phi[\vec{a} \mapsto (\{\iota\}, \mathrm{X})]), & \text{if } n > 0 \text{ and } l \in \mathsf{Key} \text{ and} \\ & \vec{a} = (l, \dots, l+n-1) \text{ and} \\ & \{l, \dots, l+n-1\} \cap \mathsf{dom}(h) \equiv \emptyset \\ (\lightning, \emptyset), & \text{otherwise} \end{cases}$$

$$[\![\mathsf{read}(\iota, k, v)]\!] \triangleq \lambda h, \Phi. \begin{cases} (h, \Phi), & \text{if } k \in \mathsf{dom}(h) \text{ and } h(k) = v \\ & \text{and } (\{\iota\} \uplus I, \kappa) = \hat{\Phi}(k) \text{ and } \kappa \geq \mathrm{S} \\ (\lightning, \emptyset), & \text{otherwise} \end{cases}$$

$$[\![\mathsf{write}(\iota, k, v)]\!] \triangleq \lambda h, \Phi. \begin{cases} (h[k \mapsto v], \Phi), & \text{if } k \in \mathsf{dom}(h) \text{ and } (\{\iota\}, \mathrm{X}) = \hat{\Phi}(k) \\ (\lightning, \emptyset), & \text{otherwise} \end{cases}$$

$$[\![\mathsf{lock}(\iota, k, \kappa)]\!] \triangleq \lambda h, \Phi. \begin{cases} (h, \Phi[k \mapsto (\{\iota\}, \kappa)]) & \text{if } k \in \mathsf{dom}(h) \text{ and} \\ & (\kappa \geq \mathrm{S} \text{ and } (\emptyset, \mathrm{U}) = \hat{\Phi}(k) \\ & \text{or } \kappa = \mathrm{X} \text{ and } (\{\iota\}, \mathrm{S}) = \hat{\Phi}(k)) \\ (h, \Phi[k \mapsto (\{\iota\} \uplus I, \mathrm{S})]) & \text{if } k \in \mathsf{dom}(h) \text{ and } \kappa = \mathrm{S} \text{ and} \\ & (I, \mathrm{S}) = \hat{\Phi}(k) \text{ and } I \not\equiv \emptyset \\ (\lightning, \emptyset), & \text{otherwise} \end{cases}$$

$$[\![\mathsf{unlock}(\iota, k)]\!] \triangleq \lambda h, \Phi. \begin{cases} (h, \Phi[k \mapsto (\emptyset, \mathrm{U})]) & \text{if } k \in \mathsf{dom}(h) \text{ and } (\{\iota\}, \kappa) = \hat{\Phi}(k) \\ (h, \Phi[k \mapsto (I, \mathrm{S})]) & \text{if } k \in \mathsf{dom}(h) \text{ and } (\{\iota\} \uplus I, \mathrm{S}) = \hat{\Phi}(k) \\ & \text{and } I \not\equiv \emptyset \\ (\lightning, \emptyset), & \text{otherwise} \end{cases}$$

Both the sys and $\mathsf{id}(\iota)$ actions have no effect on the global storage and on the lock manager structures, therefore the state transformer associated to them will simply be identity. The $\mathsf{alloc}(\iota, n, l)$ label will instead give a transformer that succeeds in creating new cells in the range of addresses from l to $l + n - 1$ (since we precisely want n consecutive cells) only when all of

such addresses are not currently in the heap domain and n is a positive number. The initial value for those storage cells is set to be 0. In all other cases, the operation will fail, bringing the resulting state to $\lightning$, i.e. the faulting state. The keys which have been allocated will also be set in the output lock manager, by locking them in exclusive mode and giving their sole ownership to transaction ι.

The intepretation of a read action label, $\mathsf{read}(\iota, k, v)$ is identity in the case where v is the value associated to address k as part of the input storage h and ι is an owner of k's lock, currently under a mode which is either shared or exclusive (expressed through the total order $\geq$). In all other scenarios the execution fails, resulting in the $\lightning$ state. The state transformer for write operations, namely $\mathsf{write}(\iota, k, v)$, works in a similar way, by requiring transaction ι to be the unique owner of cell k locked in exclusive mode. On top of this, k must obviously be part of the input storage h. If these requirements are met, then ι can update k's numerical value to the given v, otherwise the system fails and we reach $\lightning$ again.

A $\mathsf{lock}(\iota, k, \kappa)$ action label transforms the input lock manager according to a set of requirements. First, cell k must be a valid key in the input storage h. Next, we have three cases to consider. Either the lock action is requesting a shared lock on k, i.e. $\kappa = \textsc{s}$, and the current lock mode on k is already $\textsc{s}$, in which case ι is simply added to the set of lock owners. Otherwise, if the lock associated to k inside of Φ is in the unlocked mode, and there are no owners, then ι is free to lock it in any mode out of $\textsc{s}$ or $\textsc{x}$. The last scenario is referred to as *lock upgrade*, since, if ι is currently the sole owner of k under shared mode, it can issue an exclusive lock request and be granted access. All other cases are covered by reaching the faulting state. The final label to consider is the unlock one, namely $\mathsf{unlock}(\iota, k)$, which releases a lock currently owned by ι. In the case where ι was the only transaction holding the lock, then the lock manager is transformed so that k's lock turns into unlocked mode with an empty owner set. If instead there are other threads holding the same lock in shared mode, then the latter is preserved and ι is removed from the corresponding owners set.

6.2.1 Commands

The operational semantics of commands are defined as a labelled transition relation through the following rules, where $\iota \in \mathsf{Tid}$ is the identifier of the transaction as part of which the command is executed, and the labels are the actions illustrated in Definition 6.6. The effect of transaction commands is reflected locally on $\mathbb{T}_\iota$'s variable stack and locking phase, while globally through the interpretation of the action label.

$$(-,-,-) \xrightarrow{-}_{-} (-,-,-) :$$
$$(\mathsf{Stack} \times \mathsf{Phase} \times \mathsf{Cmd}) \times \mathsf{Act} \times \mathsf{Tid} \times (\mathsf{Stack} \times \mathsf{Phase} \times \mathsf{Cmd})$$

The Assign rule works in a standard way, by updating the value associated to a variable in the transaction's stack with the assigned expression evaluated under the same stack s. It then resolves into a `skip` command without modifying the transaction's locking phase.

$$\frac{v = [\![\mathbb{E}]\!]^{\text{E}}_s}{(s, p, \mathtt{x} := \mathbb{E}) \xrightarrow{\mathsf{id}(\iota)}_\iota (s[\mathtt{x} \mapsto v], p, \mathtt{skip})} \textsc{Assign}$$

The conditional statement is coped with using rules CondT and CondF that cover the cases where the boolean condition $\mathbb{B}$, evaluated under the current stack s, resolves to $\top$ or to $\bot$. In the first case, the execution will proceed to the if body, $\mathbb{C}_1$, while in the second one to the $\mathbb{C}_2$ command. Given that such control flow operations have no concrete effect on the global state of the program, conditional transitions are labelled with $\mathsf{id}(\iota)$.

$$\frac{[\![\mathbb{B}]\!]^{\text{B}}_s = \top}{(s, p, \mathtt{if}\ (\mathbb{B})\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2) \xrightarrow{\mathsf{id}(\iota)}_\iota (s, p, \mathbb{C}_1)} \textsc{CondT}$$

$$\frac{[\![\mathbb{B}]\!]_s^{\mathrm{B}} = \bot}{(s, p, \texttt{if}\ (\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2) \xrightarrow{\mathsf{id}(\iota)}_\iota (s, p, \mathbb{C}_2)}\ \textsc{CondF}$$

Loops are supported in the operational semantics in a similar way to the if statement, with two rules: LoopT and LoopF. In the first case, the condition $\mathbb{B}$ must be evaluated to $\top$ and the loop is unrolled for one iteration at a time. In the second case, when $\mathbb{B}$ is semantically evaluated to $\bot$ under the stack s, the program execution exits the loop, resolving in a `skip` command. Again, $\mathsf{id}(\iota)$ is used to label such transitions since there is no action involving the global state (i.e. storage and lock manager).

$$\frac{[\![\mathbb{B}]\!]_s^{\mathrm{B}} = \top}{(s, p, \texttt{while}\ (\mathbb{B})\ \mathbb{C}) \xrightarrow{\mathsf{id}(\iota)}_\iota (s, p, \mathbb{C}; \texttt{while}\ (\mathbb{B})\ \mathbb{C})}\ \textsc{LoopT}$$

$$\frac{[\![\mathbb{B}]\!]_s^{\mathrm{B}} = \bot}{(s, p, \texttt{while}\ (\mathbb{B})\ \mathbb{C}) \xrightarrow{\mathsf{id}(\iota)}_\iota (s, p, \texttt{skip})}\ \textsc{LoopF}$$

The SeqSkip and Seq rules, cope with the sequential composition of commands inside of a transaction's body. The first command in a composition is ran one step at a time through rule Seq, until it is eventually resolved to `skip` and at that point, the execution proceeds to the next command $\mathbb{C}$ via the SeqSkip rule. Any action label α, which is executed by $\mathbb{C}_1$, is then replicated in the reduction of $\mathbb{C}_1; \mathbb{C}_2$.

$$\frac{}{(s, p, \texttt{skip}; \mathbb{C}) \xrightarrow{\mathsf{id}(\iota)}_\iota (s, p, \mathbb{C})}\ \textsc{SeqSkip} \qquad \frac{(s, p, \mathbb{C}_1) \xrightarrow{\alpha}_\iota (s', p', \mathbb{C}_1')}{(s, p, \mathbb{C}_1; \mathbb{C}_2) \xrightarrow{\alpha}_\iota (s', p', \mathbb{C}_1'; \mathbb{C}_2)}\ \textsc{Seq}$$

A transaction can allocate fresh storage cells through the use of the $\mathtt{x} := \mathtt{alloc}(\mathbb{E})$ command, by specifying the amount of necessary space through the numerical expression $\mathbb{E}$. The Alloc rule's premiss requires n to be the result of the semantic evaluation of $\mathbb{E}$ under the stack s. The rest of the requirements are imposed by the interpretation of the $\mathsf{alloc}(\iota, n, l)$ label. The Alloc rule reduces the command to `skip` and updates the transaction stack, so that variable `x` points to the start address of the newly allocated storage cells sequence, l.

$$\frac{n = [\![\mathbb{E}]\!]_s^{\mathrm{E}}}{(s, p, \mathtt{x} := \mathtt{alloc}(\mathbb{E})) \xrightarrow{\mathsf{alloc}(\iota, n, l)}_\iota (s[\mathtt{x} \mapsto l], p, \texttt{skip})}\ \textsc{Alloc}$$

The Read rule enables the reduction of transactions reading items in the shared storage. Transitions generated this way are labelled through $\mathsf{read}(\iota, k, v)$ and carry all of the requirements imposed by the semantic interpretation of actions. The accessed key k is the result of semantically evaluating expression $\mathbb{E}$ under the current stack s. Finally, the rule reduces the command into `skip` and updates the value associated to variable `x` in s to the value read from the storage, namely v.

$$\frac{k = [\![\mathbb{E}]\!]_s^{\mathrm{E}}}{(s, p, \mathtt{x} := [\mathbb{E}]) \xrightarrow{\mathsf{read}(\iota, k, v)}_\iota (s[\mathtt{x} \mapsto v], p, \texttt{skip})}\ \textsc{Read}$$

In order to reduce a write command issued by a transaction, the Write rule is used. The target storage key k and the value v written to it are the results of semantically evaluating expressions $\mathbb{E}_1$ and $\mathbb{E}_2$ respectively, both under the transaction's stack s. The transition is labelled with the $\mathsf{write}(\iota, k, v)$ action which adds the set of requirements on the current storage and lock manager. The rule also reduces the command to `skip`.

$$\frac{k = [\![\mathbb{E}_1]\!]_s^{\mathrm{E}} \quad v = [\![\mathbb{E}_2]\!]_s^{\mathrm{E}}}{(s, p, [\mathbb{E}_1] := \mathbb{E}_2) \xrightarrow{\mathsf{write}(\iota,k,v)}_{\iota} (s, p, \mathtt{skip})} \text{ WRITE}$$

At any point during the execution of a command as part of a transaction $\mathbb{T}_\iota$, the latter can acquire a lock on a cell in the storage in a nondeterministic fashion through the LOCK rule, as long as $\mathbb{T}_\iota$ is strictly in the growing phase. The current transaction's stack, locking phase and command $\mathbb{C}$ will not be modified as part of this transition. We model locking and unlocking of cells this way, in order to cover all possible flavours of 2PL. In fact, any particular pattern of lock acquisition and release that complies with two-phase locking will be a subset of this operational semantics.

$$\frac{}{(s, \curlywedge, \mathbb{C}) \xrightarrow{\mathsf{lock}(\iota,k,\kappa)}_{\iota} (s, \curlywedge, \mathbb{C})} \text{ LOCK}$$

Opposite to the LOCK rule we present the UNLOCK one, which copes with transactions nondeterministically releasing locks on cells they own. No constraint is set on which locking phase $\mathbb{T}_\iota$ is currently in, but the successful reduction labelled via $\mathsf{unlock}(\iota, k)$ sets the transaction's phase to shrinking. This way, the same rule covers the case where the current unlock is the first one determining the start of the shrinking phase or the case where ι has already started releasing locks. As part of the reduction described by UNLOCK, the transaction's stack and command are unmodified.

$$\frac{}{(s, p, \mathbb{C}) \xrightarrow{\mathsf{unlock}(\iota,k)}_{\iota} (s, \curlyvee, \mathbb{C})} \text{ UNLOCK}$$

6.2.2 Transactions

The operational semantics of transactions are defined as a labelled transition relation through the following rule, where $\iota \in \mathsf{Tid}$ is the identifier of the transaction which is being executed and the labels are the actions performed as part of commands in the transaction's body.

$$(-,-,-) \xrightarrow{-} (-,-,-) :$$
$$(\mathsf{Stack} \times \mathsf{Phase} \times \mathsf{Trans}) \times \mathsf{Act} \times (\mathsf{Stack} \times \mathsf{Phase} \times \mathsf{Trans})$$

The only rule which allows the reduction of a transaction's body is STEP. The latter indicates that as long as we can run a single reduction step on the body of $\mathbb{T}_\iota$, namely $\mathbb{C}$, to bring the transaction's state to $s', p', \mathbb{C}'$, then we can replicate that same step on the level of the transaction to bring it to the same $s', p', \mathtt{begin}\ \mathbb{C}'\ \mathtt{end}$. Note how the transaction identifier ι is used to annotate the command reduction on $\mathbb{C}$.

$$\frac{(s, p, \mathbb{C}) \xrightarrow{\alpha}_{\iota} (s', p', \mathbb{C}')}{(s, p, \mathtt{begin}\ \mathbb{C}\ \mathtt{end}_\iota) \xrightarrow{\alpha} (s', p', \mathtt{begin}\ \mathbb{C}'\ \mathtt{end}_\iota)} \text{ STEP}$$

6.2.3 Programs

The operational semantics of programs are defined as a labelled transition relation through the following rules. Labels either come from system transitions, or from the execution of transactions and moreover of the commands within their bodies. Their effects are propagated all the way up, to the level of programs. Note how in the relation, the first component is a Storage, which is transformed into a State. This is because specific operations might lead to the faulting state, $\lightning$, from which it is not safe to proceed execution and we therefore leave the program in an idle state where it cannot further reduce.

$$(-,-,-,-) \xrightarrow{-} (-,-,-,-)$$
$$: (\mathsf{Storage} \times \mathsf{LMan} \times \mathsf{TState} \times \mathsf{Prog}) \times \mathsf{Act} \times (\mathsf{State} \times \mathsf{LMan} \times \mathsf{TState} \times \mathsf{Prog})$$

Whenever a program execution encounters a new user transaction definition that does not carry a transaction identifier, it will associate a fresh one to it, ι, and turn it into a system transaction. The identifier ι must be distinct from all others currently running in the system (and traceable via the transactions state S). Moreover ι needs to be the maximum element with regards to all other present identifiers. For this purpose, we leverage the given strict total order $<$, defined on Tid.

$$\frac{\iota \in \{i \mid \forall j \in \mathsf{dom}(S).\, j < i\}}{(h, \Phi, S, \texttt{begin}\ \mathbb{C}\ \texttt{end}) \xrightarrow{\mathsf{id}(\iota)} (h, \Phi, S[\iota \mapsto (\emptyset, \curlywedge)], \texttt{begin}\ \mathbb{C}\ \texttt{end}_\iota)}\ \textsc{Start}$$

On the opposite end, when a transaction's body is the `skip` command, the program reduces to `skip` through the PSkip rule which determines the end of the transaction. This can only happen as long as $\mathbb{T}_\iota$ (i.e. the terminating transaction) is not holding any locks as part of the lock manager. This is concretely checked by making sure the identifier ι is not part of any owners' set in Φ.

$$\frac{\forall k.\, k \in \mathsf{dom}(\Phi) \implies \iota \notin (\Phi(k) \downarrow_1)}{(h, \Phi, S, \texttt{begin skip end}_\iota) \xrightarrow{\mathsf{id}(\iota)} (h, \Phi, S, \texttt{skip})}\ \textsc{PSkip}$$

The main rule that deals with the reduction of an active transaction is Exec. This applies to a system transaction identified with ι. As long as such transaction can be reduced for one single step, labelled with action α, in the same way we can reduce the program in which $\mathbb{T}_\iota$ resides. The state of $\mathbb{T}_\iota$ is retrieved from the overall transactions state S, which gives both the local stack and locking phase for ι itself. As part of the reduction, the updated storage component h' and lock manager one Φ', will be the output of the state trasformer function obtained from the interpretation of α, and applied to the current global structures h, Φ. The new local state of $\mathbb{T}_\iota$, i.e. s' and p', is recorded by updating the mapping for ι as part of S. This way, the next time transaction ι is selected for a one-step reduction, we can pick up its local state from where it was left previously.

$$\frac{(s, p) = \hat{S}(\iota) \quad (s, p, \texttt{begin}\ \mathbb{C}\ \texttt{end}_\iota) \xrightarrow{\alpha} (s', p', \texttt{begin}\ \mathbb{C}'\ \texttt{end}_\iota) \quad (h', \Phi') = [\![\alpha]\!](h, \Phi)}{(h, \Phi, S, \texttt{begin}\ \mathbb{C}\ \texttt{end}_\iota) \xrightarrow{\alpha} (h', \Phi', S[\iota \mapsto (s', p')], \texttt{begin}\ \mathbb{C}'\ \texttt{end}_\iota)}\ \textsc{Exec}$$

Sequential composition of programs is reduced through the PSeq and PSeqSkip operational semantics rules. Similarly to the commands one, $\mathbb{P}_1; \mathbb{P}_2$ is overall reduced by first taking steps in $\mathbb{P}_1$ until the program execution hits `skip`, at which point the reduction follows with $\mathbb{P}_2$ using a system transition labelled with sys.

$$\frac{(h, \Phi, S, \mathbb{P}_1) \xrightarrow{\alpha} (h', \Phi', S', \mathbb{P}_1')}{(h, \Phi, S, \mathbb{P}_1; \mathbb{P}_2) \xrightarrow{\alpha} (h', \Phi', S', \mathbb{P}_1'; \mathbb{P}_2)}\ \textsc{PSeq} \qquad \frac{}{(h, \Phi, S, \texttt{skip}; \mathbb{P}) \xrightarrow{\mathsf{sys}} (h, \Phi, S, \mathbb{P})}\ \textsc{PSeqSkip}$$

On the other hand, the parallel composition of two programs is carried over by choosing one of the programs to reduce for one single step in a nondeterministic way. It follows that the interleaving of reductions coming from different programs happens as part of this process. Also, for this reason, we have two rules to cope with the parallel composition $\mathbb{P}_1 \| \mathbb{P}_2$: ParL and ParR, which, as the names suggest, take one step in the program which is syntactically on the left ($\mathbb{P}_1$) or on the right ($\mathbb{P}_2$).

$$\frac{(h, \Phi, S, \mathbb{P}_1) \xrightarrow{\alpha} (h', \Phi', S', \mathbb{P}_1')}{(h, \Phi, S, \mathbb{P}_1 \| \mathbb{P}_2) \xrightarrow{\alpha} (h', \Phi', S', \mathbb{P}_1' \| \mathbb{P}_2)}\ \textsc{ParL} \qquad \frac{(h, \Phi, S, \mathbb{P}_2) \xrightarrow{\alpha} (h', \Phi', S', \mathbb{P}_2')}{(h, \Phi, S, \mathbb{P}_1 \| \mathbb{P}_2) \xrightarrow{\alpha} (h', \Phi', S', \mathbb{P}_1 \| \mathbb{P}_2')}\ \textsc{ParR}$$

When both programs that are being composed in parallel reach `skip`, we can use the ParEnd rule to reduce the composition itself to `skip` through a system transition sys, which logically does not modify any components in the global state.

$$\frac{}{(h, \Phi, S, \texttt{skip} \| \texttt{skip}) \xrightarrow{\textsf{sys}} (h, \Phi, S, \texttt{skip})} \textsc{ParEnd}$$

Looping is supported by the operational semantics through the Loop rule which converts $\mathbb{P}^*$ into $\texttt{skip} + (\mathbb{P}; \mathbb{P}^*)$ via a system transition labelled with sys. The syntactic transformation turns the loop into a nondeterministic choice between terminating the program or executing the loop body ($\mathbb{P}$) and repeating the same loop choice.

$$\frac{}{(h, \Phi, S, \mathbb{P}^*) \xrightarrow{\textsf{sys}} (h, \Phi, S, \texttt{skip} + (\mathbb{P}; \mathbb{P}^*))} \textsc{Loop}$$

The ChoiceL and ChoiceR rules cover the reduction of a nondeterministic choice between two programs, $\mathbb{P}_1 + \mathbb{P}_2$. The choice happens as part of a system transition labelled with sys which does not modify the global structures h, Φ, S but only reduces the program component.

$$\frac{}{(h, \Phi, S, \mathbb{P}_1 + \mathbb{P}_2) \xrightarrow{\textsf{sys}} (h, \Phi, S, \mathbb{P}_1)} \textsc{ChoiceL} \qquad \frac{}{(h, \Phi, S, \mathbb{P}_1 + \mathbb{P}_2) \xrightarrow{\textsf{sys}} (h, \Phi, S, \mathbb{P}_2)} \textsc{ChoiceR}$$

7. Two-Phase Locking is Serializable

The idea of enabling client reasoning around 2PL using the style proposed by mCAP is only feasible once we are able to prove the logic's soundness with respect to the operational semantics for two-phase locking. This objective will guide us through the entirety of this section that gradually defines and proves all of the necessary components. We take a multi-step approach in proving soundness, by first showing the equivalence between the 2PL semantics and the ATOM semantics, where transactions are ran in one single step by reducing their body to termination. This result is achieved using a strategy that requires the serializability of two-phase locking, together with a set of properties that will be analysed in detail. Next, it is also shown that mCAP's original operational semantics, directly inherited from the Views framework [11], are able to replicate any execution that can happen through the ATOM ones. We can now see how the entire strategy fits together and will be utilized to finally complete the soundness proof.

7.1 Serializability

The operational semantics for 2PL defined in Section 6.2 claim that, given the adherence to the two-phase protocol, they guarantee the serializability of the histories of operations that arise from program executions. This section is dedicated to proving such statement by analyzing traces produced by programs. We begin with some preliminary definitions followed by auxiliary lemmata and finally a formal proof of serializability.

7.1.1 Traces

Since serializability is a property of program histories, our proof will use them and we therefore here introduce and formalize the concept of a trace.

Definition 7.1. (Trace). A *trace* is an ordered sequence of operations that are generated by the reduction of a generic program under the 2PL semantics. It comes from the set $[\mathsf{Act} \times \mathbb{N}]$ which is ranged over by $\tau, \tau_1, \ldots, \tau_n$. Every element of a trace is a tuple with an action label and an ordinal number to represent the numerical index of the operation.

We refer to elements of a trace as operations, often using op, x, y to range over them. We only study the particular kind of traces that arises from a program which succesfully terminates and reduces to `skip`. Such traces can be retrieved from program executions using the trace function, that given a storage, transactions stack and program, returns one of the possible traces that are produced by the operational semantics starting with the input state.

$$
\begin{aligned}
\mathsf{trace}(h, \Phi, S, \mathbb{P}) &\triangleq \mathsf{trace'}(h, \Phi, S, \mathbb{P}, 0) \\
\mathsf{trace'}(h, \emptyset, S, \mathtt{skip}, n) &\triangleq [] \\
\mathsf{trace'}(h, \Phi, S, \mathbb{P}, n) &\triangleq (\alpha, n) : \mathsf{trace'}(h', \Phi', S', \mathbb{P}', n+1) \\
&\quad \text{where } (h, \Phi, S, \mathbb{P}) \xrightarrow{\alpha} (h', \Phi', S', \mathbb{P}') \rightarrow^* (h'', \emptyset, S'', \mathtt{skip})
\end{aligned}
$$

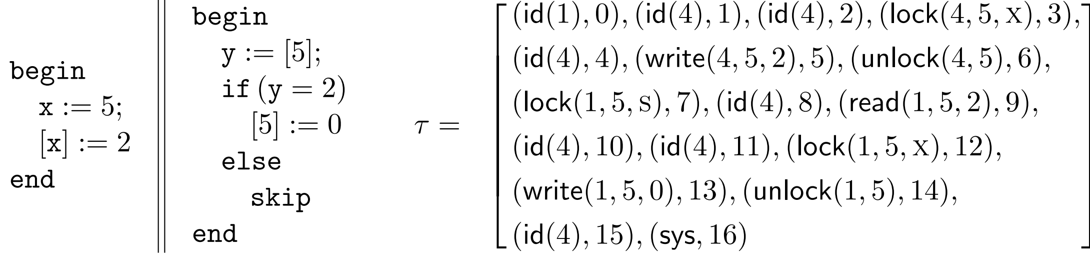


Figure 7.1: A program with parallel composition of two transaction and a possible trace generated by the execution under 2PL semantics.

To better illustrate concrete traces we provide an example program with its corresponding formally generated trace. Consider the program and trace in Figure 7.1. The first transaction assigns a value of 5 to variable `x` and later mutates the same cell to write value 2. The other one checks whether cell indexed with 5 has a value of 2 and if so, it updates it back to 0. In the execution preliminaries, the two user transactions get converted into system one and are assigned with identifiers, 4 and 1 respectively. Later, transaction 4 acquires the lock on 5 in exclusive mode, meaning it will complete the reduction of its commands before 1 starts with its own ones. Since transaction 1 needs a lock on the same cell in shared mode first and then in exclusive mode, it will acquire it with S and then upgrade it to X in order to complete the update.

Going back to formal definitions, we say that an operation $op \in \mathsf{Act} \times \mathbb{N}$ belongs to a trace τ, written $op \in \tau$ if and only if $op \in \tau = \top$ where the $\in$ predicate is defined as follows:

$$
\begin{aligned}
(\alpha, n) \in [] &\overset{\text{def}}{\iff} \bot \\
(\alpha, n) \in (\alpha', n') : \tau &\overset{\text{def}}{\iff} \big(\alpha = \alpha' \land n = n'\big) \lor (\alpha, n) \in \tau
\end{aligned}
$$

For conciseness, we often overload the $\in$ predicate to cope with actions only, without the numerical component, and write $\alpha \in \tau$ to actually mean $\exists n.\, (\alpha, n) \in \tau$. It is often useful to express the order of operations as part of a trace. For this reason, we introduce the precedence of operations which asserts that two operations belong to the trace and their respective indexes are correctly ordered.

Definition 7.2. (Operation precedence). The *operation precedence* as part of a trace τ, written $\tau \vDash x < y$, is defined in the following way:

$$
\tau \vDash (\alpha, n) < (\alpha', n') \overset{\text{def}}{\iff} (\alpha, n) \in \tau \land (\alpha', n') \in \tau \land n < n'
$$

For conciseness, we also overload the definition of $<$ to cope with the ordering of a concrete operation and a generic action (i.e. the first component of an operation) and of two generic actions without an associated index.

$$
\begin{aligned}
\tau \vDash op < \alpha &\overset{\text{def}}{\iff} \exists n.\, \tau \vDash op < (\alpha, n) \\
\tau \vDash \alpha < op &\overset{\text{def}}{\iff} \exists n.\, \tau \vDash (\alpha, n) < op \\
\tau \vDash \alpha < \alpha' &\overset{\text{def}}{\iff} \exists n, n'.\, \tau \vDash (\alpha, n) < (\alpha', n')
\end{aligned}
$$

This allows to assert on the existence of some particular action appearing before or after another one as part of a trace which will prove very useful in later definitions. Two different actions which are part of the same trace are in conflict if they are performed by different

transactions and one of them is a write or alloc, while the other one is a read, write or alloc on the same key. These are treated specially since the order in which they appear might affect the end result of the program execution.

Definition 7.3. (Conflicting actions). Actions α and α' are *conflicting* if and only if they both belong to the same trace τ and $\mathsf{conflict}(\alpha, \alpha') = \top$, where the $\mathsf{conflict}$ predicate is defined as:

$$\begin{aligned}
\mathsf{conflict}(\mathsf{write}(i,k,v), \mathsf{write}(j,k,v')) &\overset{\text{def}}{\iff} i \neq j \\
\mathsf{conflict}(\mathsf{write}(i,k,v), \mathsf{read}(j,k,v')) &\overset{\text{def}}{\iff} i \neq j \\
\mathsf{conflict}(\mathsf{read}(i,k,v), \mathsf{write}(j,k,v')) &\overset{\text{def}}{\iff} i \neq j \\
\mathsf{conflict}(\mathsf{alloc}(i,n,l), \mathsf{read}(j,k,v')) &\overset{\text{def}}{\iff} i \neq j \wedge l \leq k < l+n \\
\mathsf{conflict}(\mathsf{alloc}(i,n,l), \mathsf{write}(j,k,v')) &\overset{\text{def}}{\iff} i \neq j \wedge l \leq k < l+n \\
\mathsf{conflict}(\alpha, \alpha') &\overset{\text{def}}{\iff} \bot
\end{aligned}$$

We overload Definition 7.3 to also cope with trace operations, by simply unwrapping them and extracting the corresponding action part of the tuple.

$$\mathsf{conflict}((\alpha, n), (\alpha', n')) \triangleq \mathsf{conflict}(\alpha, \alpha')$$

Finally, in order to aid the clarity of property definitions regarding traces, we define two predicates describing particular operations. The first one, $op(\iota)$, refers to any action performed by transaction $\mathbb{T}_\iota$.

$$\begin{aligned}
op(\iota) \triangleq (\alpha, n) \text{ s.t. } \alpha \in \{&\mathsf{id}(\iota), \mathsf{write}(\iota,k,v), \mathsf{lock}(\iota,k,\kappa), \mathsf{unlock}(\iota,k), \mathsf{read}(\iota,k,v), \\
&\mathsf{alloc}(\iota,m,l) \mid k,l \in \mathsf{Key}, v \in \mathsf{Val}, \kappa \in \mathsf{Lock}, m \in \mathbb{N}\} \wedge n \in \mathbb{N}\}
\end{aligned}$$

Similarly, the $op(\iota, k)$ predicate describes any operation done by the transaction identified with ι, which accesses item k, i.e. either reading, writing or allocating it.

$$\begin{aligned}
op(\iota, k) \triangleq (\alpha, n) \text{ s.t. } \alpha \in \{&\mathsf{read}(\iota,k,v), \mathsf{write}(\iota,k,v) \mid v \in \mathsf{Val}\} \cup \\
\{&\mathsf{alloc}(\iota,m,l) \mid m \in \mathbb{N}, l \in \mathsf{Key}, l \leq k < l+m\} \wedge n \in \mathbb{N}
\end{aligned}$$

The two following definitions empower us to formally define the serializability of traces. We start by describing what it means for a trace τ to be serial: intuitively, τ is the result of running a program where transactions are executed in a sequential fashion, with no interleaving of operations. Next, we define the concept of equivalence between traces. This is expressed by imposing the equality on their operations and transactions, but not strictly on the order in which they appear. In fact, only conflicting operations are required to be ordered in the same way.

Definition 7.4. (Trace equivalence). Two traces $\tau, \tau' \in [\mathsf{Act} \times \mathbb{N}]$ are defined to be *equivalent*, written $\tau \equiv \tau'$, if and only if they contain the same transactions and operations and if they order conflicting operations in the same way.

$$\begin{gathered}
\tau \equiv \tau' \iff \\
(\forall \iota, x.\, x = op(\iota) \vee x = \mathsf{sys} \implies (x \in \tau \iff x \in \tau')) \wedge \\
(\forall x, x', i, j, k.\, x = op(i,k) \wedge x' = op(j,k) \wedge \mathsf{conflict}(x, x') \implies (\tau \vDash x < x' \iff \tau' \vDash x < x'))
\end{gathered}$$

Definition 7.5. (Serial trace). A trace τ is defined as *serial* if and only if for any two transactions i and j inside of it, either all of i's operations appear before j's ones or all of j's operations

appear before i's ones.

$$\begin{gathered}\mathsf{serial}(\tau) \iff \\ \forall i, j.\, i \neq j \land op(i) \in \tau \land op(j) \in \tau \implies \\ (\forall x, x'.\, x = op(i) \in \tau \land x' = op(j) \in \tau \implies \tau \vDash x < x') \\ \lor \\ (\forall x, x'.\, x = op(i) \in \tau \land x' = op(j) \in \tau \implies \tau \vDash x' < x)\end{gathered}$$

Definition 7.6. (Serializable). A trace τ is *serializable* if and only if it is equivalent to some serial trace τ'.

$$\mathsf{serializable}(\tau) \iff \exists \tau'.\, \mathsf{serial}(\tau') \land \tau \equiv \tau'$$

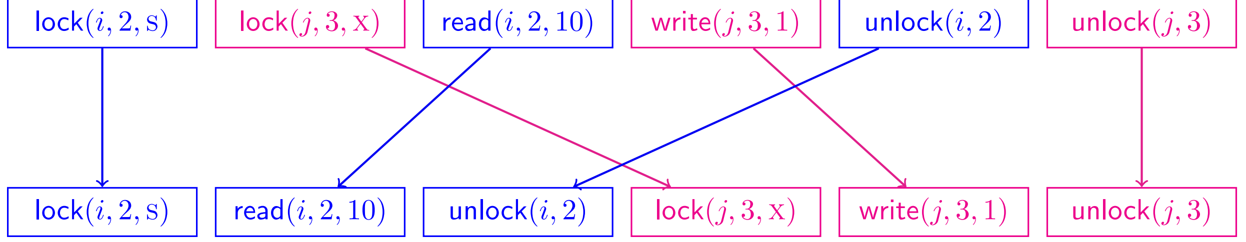


Figure 7.2: A serializable trace at the top with one of the corresponding equivalent and serial traces.

7.1.2 Graphs

The proof of serializability will be carried over a particular representation of traces that we introduce in this section. We utilize serialization graphs in order to describe relationships between transactions participating in a program execution.

Definition 7.7. (Serialization graph). The *serialization graph* of a given trace τ is a directed graph whose nodes are the identifiers of all the transactions that perform an action as part of τ and whose edges are between transactions that have ordered conflicting operations according to τ.

$$\mathsf{SG}(\tau) \triangleq (N, E)$$

$$\text{where } N \triangleq \{\iota \mid op(\iota) \in \tau\}$$

$$E \triangleq \left\{ (i, j) \;\middle|\; \begin{array}{c} x = op(i, k) \land x' = op(j, k) \\ \land\, \mathsf{conflict}(x, x') \land \tau \vDash x < x' \end{array} \right\}$$

Edge connections and paths within a graph $G = (N, E)$ are described by introducing an arrow notation. We write $i \to j \in G$ to indicate that there is an edge from node i to node j in E. In the following list of statements, we make an abuse of notation to allow more expressive formulas and add a transitive closure arrow $(\to^*)$ that is able to describe a path in the graph.

$$\begin{array}{ll} i \to j \in E \triangleq (i, j) \in E & i \to j \in G \triangleq i \to j \in G\downarrow_2 \\ i \to^1 j \in E \triangleq i \to j \in E & i \to^n j \in E \triangleq \exists t.\, i \to t \in E \land t \to^{n-1} \in E \\ i \to^* j \in E \triangleq \exists n.\, i \to^n j \in E & i \to^* j \in G \triangleq i \to^* j \in G\downarrow_2 \end{array}$$

Our goal is now to prove that all traces whose serialization graph contains no cyclic paths are serializable. We do this by first formally introducing the acyclic property of graphs and later the concept of a topological order of a graph.

Definition 7.8. (Acyclic graph). A graph $G = (N, E)$ is *acyclic* if and only if there is no path in G that connects a node in N to itself.

$$\mathsf{acyclic}(G) \iff \forall a.\, a \in G \downarrow_1 \implies \neg a \to^* a \in G$$

Definition 7.9. (Topological sort). A *topological sort* of a graph $G = (N, E)$ is an ordered sequence of all nodes in N such that if node a appears before node b in the sequence, then there is no path from b to a in G.

$$t = \mathsf{topo}((N, E)) \iff$$
$$(\forall n.\, n \in t \iff n \in N) \wedge (\forall a, b.\, t \vDash a < b \implies \neg b \to^* a \notin E)$$

The following theorem establishes a key property we later need in order to prove serializability of the operational semantics introduced in Section 6.2. The process will in fact start by building the serialization graph of an arbitrary trace arising from 2PL semantics, showing that under any circumstance it is acyclic and therefore serializable. This fundamental relationship is established in Theorem 7.10 where we obtain the needed result.

Theorem 7.10. (Acyclic means serializable). Every trace that has a serialization graph with no cycles is serializable [3].

$$\forall \tau.\, \mathsf{acyclic}(\mathsf{SG}(\tau)) \implies \mathsf{serializable}(\tau)$$

Proof. Let's assume that $\tau \in [\mathsf{Act} \times \mathbb{N}]$ is a trace which includes operations coming from transactions identified with $N = \{\iota_1, \ldots, \iota_m\}$ for a finite m. It follows that N is also the set of nodes of $\mathsf{SG}(\tau)$. By our original assumption we know that $\mathsf{SG}(\tau)$ is acyclic. For this reason we can always find a topological sort $t = \mathsf{topo}(\mathsf{SG}(\tau)) = [t_{\iota_1}, \ldots, t_{\iota_m}]$. Let τ' be the serial trace that includes transactions (in the presented order) identified with $t_{\iota_1}, \ldots, t_{\iota_m}$ and has all of the same operations as τ. Let $x = op(i, k)$ and $x' = op(j, k)$ such that $\mathsf{conflict}(x, x')$ holds and $\tau \vDash x < x'$. By definition of serialization graph, $i \to j \in \mathsf{SG}(\tau)$. Therefore, in any topological sort of $\mathsf{SG}(\tau)$, i must appear before j. As a consequence, all of i's operations appear before j's ones in τ' and in particular $\tau' \vDash x < x'$. By construction, τ' is serial and it contains all of τ's operations and we showed that any two conflicting operations are ordered in the same way. We can conclude that $\tau \equiv \tau'$ which implies that τ is serializable. □

7.1.3 Proof

The list of lemmata that follows, describes concrete properties on arbitrary traces τ generated from the 2PL operational semantics rules. The proof process follows a similar structure as the one defined in [21] and [3].

We first introduce and prove all the required properties that serve as building blocks for the final proof. The initial step is to determine that all read, write or allocation actions performed by a transaction, are always followed by an unlock action on the same key done by the same transaction.

Lemma 7.11. (Proof in B.1).

$$\forall \tau, \iota, k, \kappa, x.\, x = op(\iota, k) \wedge x \in \tau \implies (\tau \vDash x < \mathsf{unlock}(\iota, k))$$

Conversely, we are also required to show that anytime a particular transaction accesses a storage item through a read or write operation, it locks the corresponding entry in the lock manager using the required lock mode. Thus, we first show that all reads are preceded by the appropriate shared lock acquisition.

Lemma 7.12. (Proof in B.2).

$$\forall \tau, \iota, k, v, \kappa, x, n.$$
$$x = (\mathsf{read}(\iota, k, v), n) \land x \in \tau \implies (\tau \vDash \mathsf{lock}(\iota, k, \kappa) < x \land \kappa \geq \mathrm{S})$$

Then, for the write case, we also show that all writes to a cell are preceded by the appropriate exclusive lock acquisition.

Lemma 7.13. (Proof in B.3).

$$\forall \tau, x, i, k, v, n.\, x = (\mathsf{write}(i, k, v), n) \land x \in \tau \implies \tau \vDash \mathsf{lock}(i, k, \mathrm{X}) < x$$

The lock manager is not only modified with lock and unlock operations, but also with the allocation of new storage cells, which enlarges the domain of the storage and locks the new cells into exclusive mode. In relation to this, we prove that a read or write operation accessing a cell allocated as part of the trace, must appear after the corresponding alloc action. On top of this, we also show that the transaction responsible for allocating the cell will always be the first one to release the lock it holds on it, before any other transaction is able to lock it.

Lemma 7.14. (Proof in B.4).

$$\forall \tau, i, j, x, x', n, n', l, m, k, v, \kappa.$$
$$x = (\mathsf{alloc}(i, m, l), n) \land x' \in \{(\mathsf{read}(j, k, v), n'), (\mathsf{write}(j, k, v), n')\} \land l \leq k < l + m$$
$$\land\ x \in \tau \land x' \in \tau \implies \big(\tau \vDash x < x' \land \tau \vDash \mathsf{unlock}(i, k) < \mathsf{lock}(j, k, \kappa)\big)$$

Next, we shift the focus to a fundamental property of a single transaction within a trace, namely the two-phase property, from which the entire protocol gets its name from. It is therefore necessary to show that any full trace generated by our operational semantics is such that no lock is acquired by a transaction after one gets released by the same transaction.

Lemma 7.15. (Proof in B.5).

$$\forall \tau, \iota, k, k', n, n', x, x', \kappa.$$
$$\big(x = (\mathsf{lock}(\iota, k, \kappa), n) \land x' = (\mathsf{unlock}(\iota, k'), n') \land x \in \tau \land x' \in \tau\big)$$
$$\implies \big(\tau \vDash x < x'\big)$$

At this point, we can start looking at properties involving relationships between transactions that have conflicting operations within a trace. In the following lemma, we show that it must be the case that if two transactions run conflicting operations on the same item, either one releases its lock before the other acquires it or vice versa.

Lemma 7.16. (Proof in B.6).

$$\forall \tau, i, j, k, \kappa, \kappa', x, x'.$$
$$x = op(i, k) \in \tau \land x' = op(j, k) \in \tau \land \mathsf{conflict}(x, x') \implies$$
$$(\tau \vDash \mathsf{unlock}(i, k) < \mathsf{lock}(j, k, \kappa)) \lor \big(\tau \vDash \mathsf{unlock}(j, k) < \mathsf{lock}(i, k, \kappa')\big)$$

We can now start combining the lemmata demonstrated so far to prove properties about the serialization graph of traces. In fact, we show that every edge (i, j) in the serialization graph of a trace τ, corresponds to two conflicting operations in τ done by i and j respectively, and such that transaction i releases the corresponding lock before j acquires it.

Lemma 7.17.

$$\forall \tau, i, j.\, i \to j \in \mathsf{SG}(\tau) \implies$$
$$\exists k, \kappa, x, x'.\, x = op(i, k) \in \tau \land x' = op(j, k) \in \tau$$
$$\land\ \mathsf{conflict}(x, x') \land \tau \vDash \mathsf{unlock}(i, k) < \mathsf{lock}(j, k, \kappa)$$

Proof. Let's pick an arbitrary trace $\tau \in [\mathsf{Act} \times \mathbb{N}]$, such that $\tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P})$ for some $h \in \mathsf{Storage}, S \in \mathsf{TState}, \mathbb{P} \in \mathsf{Prog}$ and transaction identifiers $i, j \in \mathsf{Tid}$. Now we assume that $i \to j \in \mathsf{SG}(\tau)$. From the definition of a serialization graph built through the SG predicate, we directly obtain that there must be two operations $x = op(i,k)$ and $x' = op(j,k)$ in τ which are conflicting, so the following holds:

$$\mathsf{conflict}(x, x') \tag{12}$$

$$\wedge\ \tau \vDash x < x' \tag{13}$$

In the case where $x = (\mathsf{alloc}(i, n, l), n')$ for some $n, n' \in \mathbb{N}$, then we obtain the needed result from Lemma 7.14, since we directly know that $\tau \vDash \mathsf{unlock}(i,k) < \mathsf{lock}(j, k, \kappa)$.

In all other cases, x and x' are conflicting read/write operations. By Lemma 7.12, Lemma 7.13 and Lemma 7.11 we obtain that both x and x' must be preceded and followed by the appropriate lock acquisitions and releases. Formally, the following must hold for $x_i^l = (\mathsf{lock}(i, k, \kappa), n_1)$, $x_i^u = (\mathsf{unlock}(i,k), n_2)$, $x_j^l = (\mathsf{lock}(j, k, \kappa'), n_3)$, $x_j^u = (\mathsf{unlock}(j,k), n_4)$ and $n_1, n_2, n_3, n_4 \in \mathbb{N}, \kappa, \kappa' \in \mathsf{Lock}$.

1. $\tau \vDash x_i^l < x < x_i^u$
2. $\tau \vDash x_j^l < x' < x_j^u$

By Lemma 7.16 we know that, given the operations are conflicting from (12), either $\tau \vDash x_i^u < x_j^l$ or $\tau \vDash x_j^u < x_i^l$ holds. In the case where $\tau \vDash x_j^u < x_i^l$ holds, by points 1 and 2 we would get:

$$\begin{aligned}
&\tau \vDash x_i^l < x < x_i^u \wedge \tau \vDash x_j^l < x' < x_j^u \wedge \tau \vDash x_j^u < x_i^l \\
&\implies \tau \vDash x_j^l < x' < x_j^u < x_i^l < x < x_i^u \\
&\implies \tau \vDash x' < x
\end{aligned}$$

The last result, i.e. $\tau \vDash x' < x$, contradicts (13). Therefore it must be the case that the other option, $\tau \vDash x_i^u < x_j^l$, holds. □

The result achieved as part of Lemma 7.17 is extended in order to cover not just the neighbouring nodes in SG, but also transactions connected through a directed path of conflicts. In fact, the serialization graph itself does not give us a relationship between nodes in a path (longer than 1), but we can compute one knowing that there is a chain of pairs of transactions with conflicting operations. In fact, if node i and j are connected by a path, there must be at least one unlock operation done by i which happens before a lock one in j.

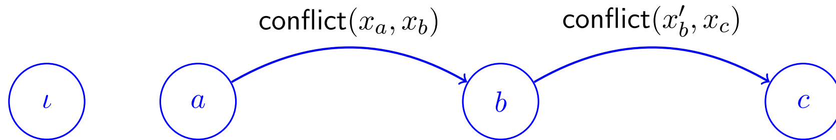


Figure 7.3: A serialization graph which shows conflicting operations between nodes.

In order to better understand this, let's focus on the example provided in Figure 7.3 where we labelled the edges of a serialization graph for clarity. We assume that the conflicting operations are instantiated as $x_a = op(a,k), x_b = op(b,k), x'_b = op(b,k')$ and $x_c = op(c,k')$. Now, since it could be the case that $k \neq k'$, there might not exists a conflict between an operation of transaction a and c. Still, from Lemma 7.17 we know that transaction a must release the lock on k before b acquires it, and b release the lock on k' before c acquires it. It follows that the unlock operation of a in k precedes the lock on k' done by c. This intuition is formally verified as part of the following lemma.

Lemma 7.18.

$$\forall \tau, i, j, n > 0.\, i \to^n j \in \mathsf{SG}(\tau) \implies$$
$$\exists k, k', \kappa.\, op(i,k) \in \tau \wedge op(j,k') \in \tau \wedge \tau \vDash \mathsf{unlock}(i,k) < \mathsf{lock}(j,k',\kappa)$$

Proof. Let's pick an arbitrary trace $\tau \in [\mathsf{Act} \times \mathbb{N}]$, such that $\tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P})$ for some $h \in \mathsf{Storage}$, $\mathbb{P} \in \mathsf{Prog}$, $S \in \mathsf{TState}$, $n \in \mathbb{Z}$ such that $n > 0$ and transaction identifiers $i, j \in \mathsf{Tid}$. We will prove the lemma by induction on n, i.e. the length of the path from node i to node j.

Base case: $n = 1$
We assume that $i \to^1 j \in \mathsf{SG}(\tau)$ which, by definition, is equivalent to $i \to j \in \mathsf{SG}(\tau)$. From Lemma 7.17 we get that $\exists k, \kappa.\, \alpha_i(k) \in \tau \wedge \alpha_j(k) \in \tau \wedge \tau \vDash \mathsf{unlock}(i,k) < \mathsf{lock}(j,k,\kappa)$ that concludes the proof for this case.

Inductive case: $n > 1$
Inductive hypothesis: Assume the property holds for n.
We now want to prove the same property for $n+1$ so we assume that:

$$i \to^{n+1} j \in \mathsf{SG}(\tau) \tag{14}$$

From (14) and the definition of $\to^{n+1}$, we know that for n and for some $t \in \mathsf{Tid}$, the following holds:

$$i \to^n t \in \mathsf{SG}(\tau) \tag{15}$$
$$\wedge\, t \to j \in \mathsf{SG}(\tau) \tag{16}$$

From (15) and the inductive hypothesis on n we obtain that, for keys $k_1, k_2 \in \mathsf{Key}$ and lock mode $\kappa_t \in \mathsf{Lock}$, there are two operations $op(i,k_1)$ and $op(t,k_2)$ which are part of τ and for which the following is true:

$$\tau \vDash \mathsf{unlock}(i,k_1) < \mathsf{lock}(t,k_2,\kappa_t) \tag{17}$$

From (16) we know that $t \to j \in \mathsf{SG}(\tau)$ holds and together with Lemma 7.17 we obtain that, for a storage key $k \in \mathsf{Key}$ and lock mode $\kappa \in \mathsf{Lock}$, there are two conflicting actions $op(t,k)$ and $op(j,k)$ inside τ such that $\tau \vDash \mathsf{unlock}(t,k) < \mathsf{lock}(j,k,\kappa)$. From Lemma 7.15 we obtain that $\tau \vDash \mathsf{lock}(t,k_2,\kappa_t) < \mathsf{unlock}(t,k)$ holds, as a consequence of the two-phase rule. Therefore, combined with (17), we then have:

$$\begin{pmatrix} \tau \vDash \mathsf{unlock}(i,k_1) < \mathsf{lock}(t,k_2,\kappa_t) \wedge \tau \vDash \mathsf{lock}(t,k_2,\kappa_t) < \mathsf{unlock}(t,k) \\ \wedge\, \tau \vDash \mathsf{unlock}(t,k) < \mathsf{lock}(j,k,\kappa) \end{pmatrix}$$
$$\implies \tau \vDash \mathsf{unlock}(i,k_1) < \mathsf{lock}(t,k_2,\kappa_t) < \mathsf{unlock}(t,k) < \mathsf{lock}(j,k,\kappa)$$
$$\implies \tau \vDash \mathsf{unlock}(i,k_1) < \mathsf{lock}(j,k,\kappa)$$

We can thus conclude with the required result. □

At this point we have formulated and shown all of the necessary ingredients to prove that the serialization graph of any 2PL trace does not contain any cycles. As part of the proof, we assume the presence of a generic cycle and using a combination of the previously defined lemmata, show that we arrive at a contradiction.

Theorem 7.19. (Acyclic serialization graph).

$$\forall \tau, i.\, i \in \mathsf{SG}(\tau)\downarrow_1 \implies \neg i \to^* i \in \mathsf{SG}(\tau)$$

Proof. Let's pick an arbitrary trace $\tau \in [\mathsf{Act} \times \mathbb{N}]$, such that $\tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P})$ for some $h \in \mathsf{Storage}$, $\mathbb{P} \in \mathsf{Prog}$, $S \in \mathsf{TState}$ and transaction identifier $i \in \mathsf{Tid}$. We assume that i is part of the transaction identifiers in the serialization graph, i.e. $i \in \mathsf{SG}(\tau) \downarrow_1$. Let's now also assume that the graph contains a cycle on i, which means that $\exists n.\, i \to^n i \in \mathsf{SG}(\tau)$. From Lemma 7.18 we obtain that for some keys $k, k' \in \mathsf{Key}$ and lock mode $\kappa \in \mathsf{Lock}$, $\tau \vDash \mathsf{unlock}(i, k) < \mathsf{lock}(i, k', \kappa)$ which contradicts the two-phase rule defined and proved in Lemma 7.15. Therefore, by contradiction we conclude that $\neg i \to^* i \in \mathsf{SG}(\tau)$. □

For completion, we combine the theorem which was just proven with our results from Section 7.1 in order to formally show that every 2PL trace produced by our operational semantics is serializable.

Theorem 7.20. (Serializability).

$$\forall \tau, h, S, \mathbb{P}.\, \tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P}) \implies \mathsf{serializable}(\tau)$$

Proof. Let's pick an arbitrary trace $\tau \in [\mathsf{Act} \times \mathbb{N}]$ and assume that $\tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P})$ for program $\mathbb{P}$, transactions' stack S and storage h. From Theorem 7.19 we obtain that $\mathsf{acyclic}(\tau)$ holds and from Theorem 7.10 we know that $\mathsf{serializable}(\tau)$ holds, which concludes our proof. □

7.2 Semantics Equivalence

Serializability gives us an important consistency property on the 2PL operational semantics that we defined. In abstract terms, it allows us to think about program executions in some sequential order, without having to consider all possible interleavings that may arise. At this point, we want to show an even stronger result which is the semantics equivalence between the semantics defined in Section 6.2 and some *atomic* operational semantics, defined in Section 7.2, that reduce transactions in one go, as if the system stops when they begin execution and starts again the moment they are done. In database terms, this is referred to as *strict serializability*, a property which enforces that, on top of some order between parallel transactions, the internal program order is also preserved. The equivalence not only allows us to forget about the peculiar locking details of 2PL and thus only to focus on *atomic* reductions, but also provides us with the key element in proving soundness with respect to the mCAP program logic defined in Section 4. This empowers us to build mCAP proofs of programs running under the 2PL semantics.

The final outcome of this section will be a proof of the following statement, which says that any program that terminates, i.e. reduces to `skip`, under the 2PL operational semantics will have the same overall effect on the global storage as the same terminating program running under the ATOM semantics and starting with the same state.

$$\forall h, h', S, S', \mathbb{P}.\, (h, \emptyset, S, \mathbb{P}) \to^* (h', \emptyset, S', \mathtt{skip}) \implies (h, \mathbb{P}) \to^*_{\text{ATOM}} (h', \mathtt{skip})$$

In order for the semantics equivalence claim to be complete, the inverse implication also needs to be established. This states that the final storage of any terminating program reduction in ATOM, will be reachable by reducing the same program starting from the same initial state in 2PL, where the transactions' state components are existentially quantified.

$$\forall h, h'.\, (h, \mathbb{P}) \to^*_{\text{ATOM}} (h', \mathtt{skip}) \implies \exists S, S'.\, (h, \emptyset, S, \mathbb{P}) \to^* (h', \emptyset, S', \mathtt{skip})$$

We do not formally prove this result given that it is trivial to understand that the ATOM semantics can be replicated by the 2PL ones, simply by always chosing to reduce the same transaction until it reaches `skip`, without allowing concurrent interleaving. No transaction will get *stuck* as part of the overall reduction under 2PL, given that we start with an empty lock manager component $\emptyset$, run transactions one after the other, and each of those needs to remove its footprint from locks before terminating.

7.2.1 Atomic semantics

The atomic operational semantics shown here are behaviourally equivalent to the ones presented in Section 4.7. They are converted to a transition relation for clarity and in order to ease the general proof, since the 2PL semantics are also expressed with a comparable structure.

The rules that follow determine a relation between storages under the effect of a program. At the level of programs, they are very similar to the 2PL ones, a part from the absence of a lock manager or a global transactions stack. These two structures are not needed here, since there is no interleaving which happens as a result of running transactions concurrently. For this reason, there is no need to globally track information about a transaction's internal execution. Note how such behaviour is obtained through the AtExec rule, which reduces a transaction's body at once, by running a multi-step reduction on the body command $\mathbb{C}$ until it hits `skip`. Parallelism is again obtained by nondeterministically reducing one of the two programs that are composed together.

$$(-,-) \rightarrow (-,-) : (\mathsf{Storage} \times \mathsf{Prog})^2$$

$$\frac{(h, \mathbb{T}) \rightarrow_{\textsc{Atom}} (h', \texttt{begin skip end})}{(h, \mathbb{T}) \rightarrow_{\textsc{Atom}} (h', \texttt{skip})} \textsc{AtTrans} \qquad \frac{}{(h, \texttt{skip}; \mathbb{P}) \rightarrow_{\textsc{Atom}} (h, \mathbb{P})} \textsc{AtPSkip}$$

$$\frac{(h, \mathbb{P}_1) \rightarrow_{\textsc{Atom}} (h', \mathbb{P}_1')}{(h, \mathbb{P}_1; \mathbb{P}_2) \rightarrow_{\textsc{Atom}} (h', \mathbb{P}_1'; \mathbb{P}_2)} \textsc{AtPSeq} \qquad \frac{}{(h, \texttt{skip}\|\texttt{skip}) \rightarrow_{\textsc{Atom}} (h, \texttt{skip})} \textsc{AtPar}$$

$$\frac{(h, \mathbb{P}_1) \rightarrow_{\textsc{Atom}} (h', \mathbb{P}_1')}{(h, \mathbb{P}_1 \| \mathbb{P}_2) \rightarrow_{\textsc{Atom}} (h', \mathbb{P}_1' \| \mathbb{P}_2)} \textsc{AtParL} \qquad \frac{(h, \mathbb{P}_2) \rightarrow_{\textsc{Atom}} (h', \mathbb{P}_2')}{(h, \mathbb{P}_1 \| \mathbb{P}_2) \rightarrow_{\textsc{Atom}} (h', \mathbb{P}_1 \| \mathbb{P}_2')} \textsc{AtParR}$$

$$\frac{}{(h, \mathbb{P}_1 + \mathbb{P}_2) \rightarrow_{\textsc{Atom}} (h, \mathbb{P}_1)} \textsc{AtChoiceL} \qquad \frac{}{(h, \mathbb{P}_1 + \mathbb{P}_2) \rightarrow_{\textsc{Atom}} (h, \mathbb{P}_2)} \textsc{AtChoiceR}$$

$$\frac{}{(h, \mathbb{P}^*) \rightarrow_{\textsc{Atom}} (h, \texttt{skip} + (\mathbb{P}; \mathbb{P}^*))} \textsc{AtLoop} \qquad \frac{(h, \emptyset, \mathbb{C}) \rightarrow^*_{\textsc{Atom}} (h', -, \texttt{skip})}{(h, \texttt{begin } \mathbb{C} \texttt{ end}) \rightarrow_{\textsc{Atom}} (h', \texttt{begin skip end})} \textsc{AtExec}$$

The atomic operational semantics of commands are defined through the rules that follow. They show the reduction of a command when executed on a storage h and variable stack s. The rules are equivalent to the 2PL ones but, given the atomic setting, there is no need for a state component that determines the phase of a transaction as the process of locking is entirely absent. From this, it follows that there are no rules concerned with locking and unlocking either, since under the atomic semantics, transactions run in concrete and real isolation without the need to be managed when accessing storage cells.

$$(-,-,-) \rightarrow_{\textsc{Atom}} (-,-,-) : (\mathsf{Storage} \times \mathsf{Stack} \times \mathsf{Cmd})^2$$

$$\frac{}{(h, s, \texttt{skip}; \mathbb{C}) \rightarrow_{\text{Atom}} (h, s, \mathbb{C})} \textsc{AtSkip} \qquad \frac{[\![\mathbb{B}]\!]^{\text{B}}_s = \top}{(h, s, \texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2) \rightarrow_{\text{Atom}} (h, s, \mathbb{C}_1)} \textsc{AtCondT}$$

$$\frac{(h, s, \mathbb{C}_1) \rightarrow_{\text{Atom}} (h', s', \mathbb{C}'_1)}{(h, s, \mathbb{C}_1; \mathbb{C}_2) \rightarrow_{\text{Atom}} (h', s', \mathbb{C}'_1; \mathbb{C}_2)} \textsc{AtSeq} \qquad \frac{k = [\![\mathbb{E}_1]\!]^{\text{E}}_s \quad k \in \mathsf{dom}(h) \quad v = [\![\mathbb{E}_2]\!]^{\text{E}}_s}{(h, s, [\mathbb{E}_1] := \mathbb{E}_2) \rightarrow_{\text{Atom}} (h[k \mapsto v], s, \texttt{skip})} \textsc{AtWrite}$$

$$\frac{v = [\![\mathbb{E}]\!]^{\text{E}}_s}{(h, s, \texttt{x} := \mathbb{E}) \rightarrow_{\text{Atom}} (h, s[\texttt{x} \mapsto v], \texttt{skip})} \textsc{AtAssign} \qquad \frac{[\![\mathbb{B}]\!]^{\text{B}}_s = \bot}{(h, s, \texttt{while}\,(\mathbb{B})\ \mathbb{C}) \rightarrow_{\text{Atom}} (h, s, \texttt{skip})} \textsc{AtLoopF}$$

$$\frac{[\![\mathbb{B}]\!]^{\text{B}}_s = \bot}{(h, s, \texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2) \rightarrow_{\text{Atom}} (h, s, \mathbb{C}_2)} \textsc{AtCondF} \qquad \frac{k = [\![\mathbb{E}]\!]^{\text{E}}_s \quad k \in \mathsf{dom}(h) \quad v = h(k)}{(h, s, \texttt{x} := [\mathbb{E}]) \rightarrow_{\text{Atom}} (h, s[\texttt{x} \mapsto v], \texttt{skip})} \textsc{AtRead}$$

$$\frac{[\![\mathbb{B}]\!]^{\text{B}}_s = \top}{(h, s, \texttt{while}\,(\mathbb{B})\ \mathbb{C}) \rightarrow_{\text{Atom}} (h, s, \mathbb{C}; \texttt{while}\,(\mathbb{B})\ \mathbb{C})} \textsc{AtLoopT}$$

$$\frac{n = [\![\mathbb{E}]\!]^{\text{E}}_s \quad n > 0 \quad \{l, \dots, l+n-1\} \cap \mathsf{dom}(h) \equiv \emptyset}{(h, s, \texttt{x} := \texttt{alloc}(\mathbb{E})) \rightarrow_{\text{Atom}} (h[l \mapsto 0] \dots [l+n-1 \mapsto 0], s[\texttt{x} \mapsto l], \texttt{skip})} \textsc{AtAlloc}$$

Rules for Atom commands differ from the semantics given in Section 5.3, since they model reaching a faulting state by getting *stuck* and not being able to reduce further, instead of setting the state component to $\lightning$. This is done in a similar way by the 2PL operational semantics as well, where action labels can transform the state into the faulting one ($\lightning$), but from there no further reduction is possible. The overall needed result of equivalence is that any terminating execution of running a program under the Atom semantics can be replicated by the Views' ones.

Theorem 7.21. (Atom to Views).

$$\forall h, h', \mathbb{P}.\, (h, \mathbb{P}) \rightarrow^*_{\text{ATOM}} (h', \texttt{skip}) \implies$$
$$\left(\exists \sigma, \sigma'.\, \mathbb{P}, \sigma \rightarrow^*_{\mathsf{Views}} \texttt{skip}, \sigma' \wedge h = \sigma \downarrow_1 \wedge h' = \sigma' \downarrow_1\right)$$

We only prove the above in the case of $\mathbb{T} \in \mathsf{Prog}$, for $\mathbb{C} \in \mathsf{Cmd}$ and $\hat{\mathbb{C}} \in \mathsf{ECmd}$, since the rules for programs $\mathbb{P} \neq \mathbb{T}$ have the same exact structure as the labelled transition system for the operational semantics presented in Views and therefore inherited by mCAP and our transactional instantiation. The formal proof appears in Lemma C.11.

7.2.2 Trace equivalence

The final proof of equivalence will need a number of preliminary lemmata in order to succeed. Most of them describe a series of concrete transformations applied to traces that preserve the semantic meaning of the trace in terms of the effects on the global storage. We start by giving some helpful definitions that will be later used as part of the lemmata.

The first one allows to express that a trace τ is able to generate a particular storage starting from a given initial state with a storage, lock manager, transactions' stack and program components. It serves the purpose of comparing different traces by establishing a type of trace equivalence. In fact, we obtain important information knowing that two distinct traces τ and τ' can generate the same heap under specific circumstances.

Definition 7.22. (Trace generated). A trace τ *generates* a storage $\underline{h}$ starting from the state $h, \Phi, S, \mathbb{P}$ if and only if every action in τ labels one of consecutive reductions that bring $\mathbb{P}$ to `skip` under the 2PL operational semantics.

$$\begin{aligned}
\mathsf{tgen}([], h, \underline{h}, \Phi, S, \mathbb{P}) &\iff h = \underline{h} \wedge \mathbb{P} = \texttt{skip} \wedge \Phi = \emptyset \\
\mathsf{tgen}((\alpha, n) : \tau, h, \underline{h}, \Phi, S, \mathbb{P}) &\iff \exists h', \Phi', S', \mathbb{P}'.\, (h, \Phi, S, \mathbb{P}) \xrightarrow{\alpha} (h', \Phi', S', \mathbb{P}') \\
&\qquad \wedge \mathsf{tgen}(\tau, h', \underline{h}, \Phi', S', \mathbb{P}')
\end{aligned}$$

In order for the 2PL operational semantics to model any kind of two phase locking pattern, we utilize nondeterministic locking of storage cells, as described in Section 6.2. This comes with the downside of allowing a transaction to lock (and later unlock) any cell it wants. Such phenomenon can happen even if later in the execution the same transaction does not use the locks to access the respective storage cells to read from or write to them. Lock operations performed by a transaction as part of a trace, which are never used to later access the corresponding cell are often and informally referred to as *spurious*. The next definition formally describes this behaviour.

Definition 7.23. (Absent access). A cell which is neither read, or written to, by a given transaction as part of a trace is *absent* and the corresponding predicate is defined as follows:

$$\mathsf{absent}(\iota, k, \tau) \iff \neg\exists v, n.\, (\mathsf{read}(\iota, k, v), n) \in \tau \vee (\mathsf{write}(\iota, k, v), n) \in \tau$$

Another situation that might arise as a consequence of nondeterministic locking, which is not covered by the absent predicate, is when a transaction locks a cell in exclusive mode only to read from it, when it could just use the same lock in shared mode.

Definition 7.24. (Redundant lock). A lock operation for mode κ on cell k done by transaction ι as part of trace τ is *redundant* if and only if κ is exclusive and every time ι accesses k, it does so only to read from it.

$$\mathsf{redundant}(\iota, k, \kappa, \tau) \iff$$
$$(\kappa = \mathrm{X} \wedge \forall v, n.\, \alpha = \alpha(\iota, k) \wedge (\alpha, n) \in \tau \implies \alpha = \mathsf{read}(\iota, k, v))$$

When none of the two described phenomena occur as part of a given trace, we say that the latter is *clean* and we are ready to start modifying it.

Definition 7.25. (Clean trace). A trace is *clean* if and only if its transactions do not lock or unlock any absent cell and they don't hold reduntant locks.

$$\mathsf{clean}(\tau) \iff$$
$$\forall \iota, k, \kappa, n.\, ((\mathsf{lock}(\iota, k, \kappa), n) \in \tau \vee (\mathsf{unlock}(\iota, k), n) \in \tau)$$
$$\implies \neg\mathsf{absent}(\iota, k, \tau) \wedge \neg\mathsf{redundant}(\iota, k, \kappa, \tau)$$

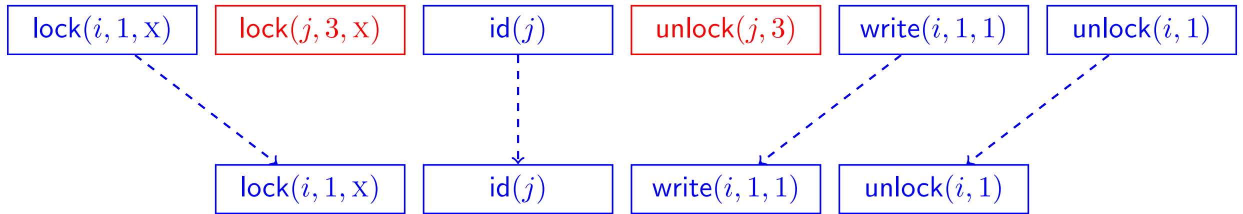


Figure 7.4: A trace containing spurious locks and its clean equivalent.

We are able to change the structure of a trace through the union or the difference with a set of operations, which we formally define as follows.

Definition 7.26. (Trace difference).

$$\tau' = \tau \setminus S \iff (\forall x.\, x \notin S \implies (x \in \tau \iff x \in \tau'))$$

Definition 7.27. (Trace union).

$$\tau' = \tau \cup S \iff$$
$$(\forall \alpha, n.\, (\alpha, n) \in S \implies (\neg\exists\alpha'.\, (\alpha', n) \in \tau)$$
$$\wedge\, \forall x.\, (x \in \tau \vee x \in S) \iff x \in \tau')$$

It is now possible to show that any trace τ that contains a spurious lock (followed by an unlock on the same item) and is able to generate a particular storage h', has a corresponding trace τ' which contains all of τ's operations apart from the spurious ones, and τ' can generate h' starting with the same initial state and program. This intuitively means that spurious locks and unlocks are unnecessary operations and as a consequence they do not affect the final state.

Lemma 7.28. (Proof in C.1). Lock and unlock operations done by a transaction on items which it does not read or write can be removed without affecting the program or the global state.

$$\begin{gathered}\forall \tau, \tau', h, h', \Phi, S, \mathbb{P}, n, n', \iota, k, \kappa, x, y. \\ \mathsf{tgen}(\tau, h, h', \Phi, S, \mathbb{P}) \wedge \mathsf{absent}(\iota, k, \tau) \wedge x = (\mathsf{lock}(\iota, k, \kappa), n) \wedge y = (\mathsf{unlock}(\iota, k), n') \\ \wedge x \in \tau \wedge y \in \tau \wedge \tau' = \tau \setminus \{x, y\} \implies \mathsf{tgen}(\tau', h, h', \Phi, S, \mathbb{P})\end{gathered}$$

In a similar way we also show that if redundant locks are replaced with their shared versions, then the original program reductions can still be performed and they terminate with the same resulting storage. The following statement uses the swap predicate which is presented in Definition 7.32.

Lemma 7.29. (Proof in C.2). Redundant exclusive lock operations present in a trace, can be converted to the corresponding shared equivalent without affecting the end storage.

$$\begin{gathered}\forall \tau, \tau', h, h', \Phi, S, \mathbb{P}, n, \iota, k, \kappa, x. \\ \mathsf{tgen}(\tau, h, h', \Phi, S, \mathbb{P}) \wedge \mathsf{redundant}(\iota, k, \kappa, \tau) \wedge x = (\mathsf{lock}(\iota, k, \kappa), n) \\ \wedge x \in \tau \wedge \tau' = \mathsf{swap}(\tau, x, (\mathsf{lock}(\iota, k, \mathrm{S}), n)) \implies \mathsf{tgen}(\tau', h, h', \Phi, S, \mathbb{P})\end{gathered}$$

Both proofs use another important property of operations inside a trace, which is the fact that a lock on an item is not needed for any reductions a part from a read, a write or an unlock action performed by the same transaction on the same item. As we will see later, we leverage this proof in order to *clean* traces from any spurious locks they contain.

Lemma 7.30. (Proof in C.3).

$$\begin{gathered}\forall \mathbb{P}, \mathbb{P}', h, h', \Phi, \Phi', S, S', \alpha, i, k, v, I, \kappa. \\ (h, \Phi, S, \mathbb{P}) \xrightarrow{\alpha} (h', \Phi', S', \mathbb{P}') \wedge (\{i\} \uplus I, \kappa) = \Phi(k) \wedge \\ \alpha \notin \{\mathsf{read}(i, k, v), \mathsf{write}(i, k, v), \mathsf{unlock}(i, k)\} \implies \exists \Phi_m, \Phi'_m, I', \kappa', \kappa''. \\ (h, \Phi_m, S, \mathbb{P}) \xrightarrow{\alpha} (h', \Phi'_m, S, \mathbb{P}') \wedge \Phi_m = \Phi[k \mapsto (I, \kappa')] \wedge \Phi'_m = \Phi'[k \mapsto (I', \kappa'')] \wedge \kappa' \leq \mathrm{S}\end{gathered}$$

Once a trace is clean, we can proceed to manipulate it and, under certain circumstances, swap the order of its operations. We only do so when the reordering does not change the final program state. Repeating this process will give us a new trace, syntactically different but semantically equivalent, as observed by its effects on the global heap. This is what we are interested in, given our requirement of simulating a serial trace. Whenever in the following definitions we encounter a term of the shape $\alpha(\iota)$ or $\alpha(\iota, k)$ we actually mean the first projection of their operation-level equivalent, i.e. $op(\iota) \downarrow_1$ and $op(\iota, k) \downarrow_1$ respectively.

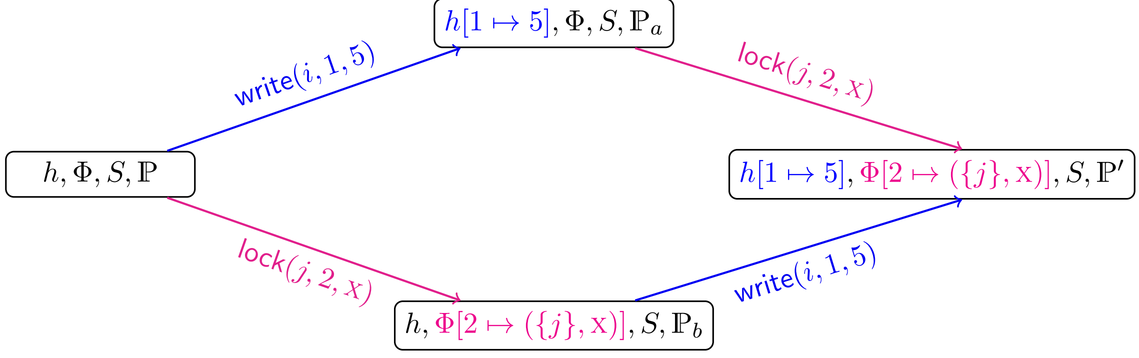


Figure 7.5: An example of swapping consecutive actions in a program reduction. Note how the end state does not change as a result of the reordering.

We only consider swaps of consecutive actions inside a trace and identify the pairs of operations α, α' that can be swapped by focusing on *feasible* sequences of two immediate program reductions as depicted in Figure 7.5. These reductions are the ones which are allowed by the 2PL operational semantics. For example, the action $\alpha = \mathsf{write}(1, k, v)$ immediately followed by $\alpha' = \mathsf{read}(2, k, v)$ is not a feasible pair, since under 2PL such a reduction could never happen due to the locking constraints. In all other feasible cases, we show that the order of two consecutive operations does not matter, by explicitly finding a new intermediate program state that allows α' to reduce first followed by α. This last reduction must bring the program to the same state as α' did in the original case, before the swap took place.

Lemma 7.31. (Feasible swap rules).

$$\forall \alpha, \alpha', i, j, k, \kappa, h, h', h_a, \Phi, \Phi', \Phi_a, S, S', S_a, \mathbb{P}, \mathbb{P}', \mathbb{P}_a.$$
$$(h, \Phi, S, \mathbb{P}) \xrightarrow{\alpha} (h_a, \Phi_a, S_a, \mathbb{P}_a) \xrightarrow{\alpha'} (h', \Phi', S', \mathbb{P}') \wedge i \neq j$$
$$\wedge \neg \left(\alpha = \mathsf{unlock}(i, k) \wedge \alpha' = \mathsf{lock}(j, k, \kappa) \wedge \Phi_a(k) \downarrow_2 = \mathrm{X}\right) \wedge \alpha = \alpha(i) \wedge \alpha' = \alpha(j)$$
$$\implies \exists h_b, \Phi_b, S_b, \mathbb{P}_b.\, (h, \Phi, S, \mathbb{P}) \xrightarrow{\alpha'} (h_b, \Phi_b, S_b, \mathbb{P}_b) \xrightarrow{\alpha} (h', \Phi', S', \mathbb{P}')$$

Proof. Let's pick arbitrary $\alpha, \alpha' \in \mathsf{Act}, i, j \in \mathsf{Tid}, k \in \mathsf{Key}, \kappa \in \mathsf{Lock}, h, h', h_a \in \mathsf{Storage}$, $\Phi, \Phi', \Phi_a \in \mathsf{LMan}, S, S', S_a \in \mathsf{TState}, \mathbb{P}, \mathbb{P}', \mathbb{P}_a \in \mathsf{Prog}$ and assume that the following holds:

$$(h, \Phi, S, \mathbb{P}) \xrightarrow{\alpha} (h_a, \Phi_a, S_a, \mathbb{P}_a) \xrightarrow{\alpha'} (h', \Phi', S', \mathbb{P}') \wedge i \neq j \tag{18}$$
$$\wedge \neg \left(\alpha = \mathsf{unlock}(i, k) \wedge \alpha' = \mathsf{lock}(j, k, \kappa)\right) \wedge \alpha = \alpha(i) \wedge \alpha' = \alpha(j) \tag{19}$$

We now proceed in finding $h_b, \Phi_b, S_b, \mathbb{P}_b$ by doing a case-by-case analysis on α and α' based on the feasible reductions they can be part of, according to (18). From (19) we know that neither α nor α' are system transitions sys because of the definition of $\alpha(i)$ and $\alpha(j)$. Similarly, we can't have a situation where α' is a lock on a key that has just been unlocked by α which released a lock in exclusive mode. Also, since from (18) we know that $i \neq j$, we obtain that the two actions come from parallel transactions. This implies that, without loss of generality, programs $\mathbb{P}, \mathbb{P}_a, \mathbb{P}'$ have the following shape:

$$\mathbb{P} = (\mathbb{T}_i; \mathbb{P}_1) \parallel (\mathbb{T}_j; \mathbb{P}_2) \parallel \mathbb{P}_3 \quad \mathbb{P}_a = (\mathbb{T}'_i; \mathbb{P}_1) \parallel (\mathbb{T}_j; \mathbb{P}_2) \parallel \mathbb{P}_3 \quad \mathbb{P}' = (\mathbb{T}'_i; \mathbb{P}_1) \parallel \left(\mathbb{T}'_j; \mathbb{P}_2\right) \parallel \mathbb{P}_3$$

Let us also define the new intermediate program, $\dot{\mathbb{P}}$, for all the cases we will consider:

$$\dot{\mathbb{P}} = (\mathbb{T}_i; \mathbb{P}_1) \parallel \left(\mathbb{T}'_j; \mathbb{P}_2\right) \parallel \mathbb{P}_3$$

In the following list we only consider one of the two combinations for each pair as the other one can be trivially found with the appropriate substitutions.

(i) If $\alpha = \mathsf{id}(i)$ then from the semantic interpretation of id, we obtain that $h_a = h, \Phi_a = \Phi, S_a = S$. It follows that for any action α' performed by j we have the following reduction, from (18):

$$(h, \Phi, S, \mathbb{P}) \xrightarrow{\alpha'} (h', \Phi', S', \dot{\mathbb{P}}) \xrightarrow{\alpha} (h', \Phi', S', \mathbb{P}')$$

We can therefore pick $h_b = h', \Phi_b = \Phi', S_b = S'$ and $\mathbb{P}_b = \dot{\mathbb{P}}$. The same occurs when picking $\alpha' = \mathsf{id}(j)$ and an arbitrary $\alpha = \alpha(i)$.

(ii) If $\alpha = \mathsf{read}(i, k, v)$ and $\alpha' = \mathsf{read}(j, k', v')$ then from the semantic interpretation of read we know that $h' = h_a = h, \Phi' = \Phi_a = \Phi$. Also, the two read actions will update transaction-local variables to the values read from the storage so we select S_b to be $S[j \mapsto (s[\mathtt{x} \mapsto v], p)]$ where $(s, p) = S(j)$. This implies that we can always reduce in the following way:

$$(h, \Phi, S, \mathbb{P}) \xrightarrow{\alpha'} (h, \Phi, S_b, \dot{\mathbb{P}}) \xrightarrow{\alpha} (h, \Phi, S', \mathbb{P}')$$

We can therefore pick $h_b = h', \Phi_b = \Phi'$ and $\mathbb{P}_b = \dot{\mathbb{P}}$.

(iii) If $\alpha = \mathsf{unlock}(i, k)$ and $\alpha' = \mathsf{lock}(j, k', \kappa)$ then from (19) it must either be the case that $k \neq k'$ or $k = k'$ and $\Phi(k) = (\{i\} \uplus I, \mathrm{S})$. From (18) we get that $\kappa = \mathrm{S}$ in the case of $k = k'$.

- If $k \neq k'$ then we can always find $\Phi_b = \Phi[k' \mapsto (\{j\} \uplus I', \kappa)]$ for $(I', \kappa') = \Phi(k'), j \notin I$ and $\kappa' \leq \mathrm{S}$
- If $k = k'$ and $\Phi(k) = (\{i\} \uplus I, \mathrm{S})$ then $\Phi_b = \Phi[k \mapsto (\{i, j\}, \mathrm{S})]$

In both cases the storage, stack and program components of the intermediate state will be $h_b = h, S_b = S, \mathbb{P}_b = \dot{\mathbb{P}}$.

(iv) If $\alpha = \mathsf{read}(i, k, v)$ and $\alpha' = \mathsf{write}(j, k', v')$ then from (18) it must be the case that $k \neq k'$. This means that given the disjointness of keys, we can always find $h_b = h', \Phi_b = \Phi, S_b = S, \mathbb{P}_b = \dot{\mathbb{P}}$.

(v) If $\alpha = \mathsf{write}(i, k, v)$ and $\alpha' = \mathsf{write}(j, k', v')$ then from (18) it must be the case that $k \neq k'$. Now we can always find $h_b = h[k' \mapsto v'], \Phi_b = \Phi, S_b = S, \mathbb{P}_b = \dot{\mathbb{P}}$.

(vi) If $\alpha = \mathsf{read}(i, k, v)$ and $\alpha' = \mathsf{lock}(j, k', \kappa)$ then from (18) it must be the case that either $k \neq k'$ or $k = k'$ and $\Phi(k) = (\{i\} \uplus I, \mathrm{S})$ and $\kappa = \mathrm{S}$. In both cases we can find $h_b = h, \Phi_b = \Phi', S_b = S, \mathbb{P}_b = \dot{\mathbb{P}}$.

(vii) If $\alpha = \mathsf{write}(i, k, v)$ and $\alpha' = \mathsf{lock}(j, k', \kappa)$ then from (18) it must be the case that $k \neq k'$. Now we can always find $h_b = h, \Phi_b = \Phi', S_b = S, \mathbb{P}_b = \dot{\mathbb{P}}$.

(viii) If $\alpha = \mathsf{read}(i, k, v)$ and $\alpha' = \mathsf{unlock}(j, k')$ then from (18) it must be the case that either $k \neq k'$ or $k = k'$ and $\Phi(k) = (\{i, j\} \uplus I, \mathrm{S})$. In both cases we can find $h_b = h, \Phi_b = \Phi', S_b = S[j \mapsto (s, \curlyvee)], \mathbb{P}_b = \dot{\mathbb{P}}$ for $s = S(j) \downarrow_1$.

(ix) If $\alpha = \mathsf{write}(i, k, v)$ and $\alpha' = \mathsf{unlock}(j, k')$ then from (18) it must be the case that $k \neq k'$. Now we can always find $h_b = h, \Phi_b = \Phi', S_b = S', \mathbb{P}_b = \dot{\mathbb{P}}$.

(x) If $\alpha = \mathsf{read}(i, k, v)$ and $\alpha' = \mathsf{alloc}(j, n, l)$ then from (18) it must be the case that $k < l \vee l \geq l + n$. Now we can always find $h_b = h', \Phi_b = \Phi', S_b = S[j \mapsto (s[\mathtt{x} \mapsto l], p)], \mathbb{P}_b = \dot{\mathbb{P}}$. Where $\mathtt{x}$ is the variable recording the newly allocated address l in transaction j and $(s, p) = S(j)$.

(xi) If $\alpha = \mathsf{write}(i, k, v)$ and $\alpha' = \mathsf{alloc}(j, n, l)$ then from (18) it must be the case that $k < l \vee l \geq l + n$. Now we can always find $h_b = h[l \mapsto 0] \ldots [l + n - 1 \mapsto 0], \Phi_b = \Phi', S_b = S[j \mapsto (s[\mathtt{x} \mapsto l], p)], \mathbb{P}_b = \dot{\mathbb{P}}$. Where $\mathtt{x}$ is the variable recording the newly allocated address l in transaction j and $(s, p) = S(j)$.

(xii) If $\alpha = \mathsf{lock}(i, k, \kappa)$ and $\alpha' = \mathsf{lock}(j, k', \kappa')$ then from (18) it must be the case that either $k \neq k'$ or $k = k'$ and $\kappa = \kappa' = \text{S}$. In the first scenario we can find $\Phi_b = \Phi[k' \mapsto (\{j\} \uplus I', \kappa')]$ for $I' = \Phi(k') \downarrow_1$ while in the second one $\Phi_b = \Phi[k \mapsto (\{j\} \uplus I, \text{S})]$ for $I = \Phi(k) \downarrow_1$ with $i \notin I$. In both cases we pick $h_b = h, S_b = S, \mathbb{P}_b = \dot{\mathbb{P}}$.

(xiii) If $\alpha = \mathsf{unlock}(i, k)$ and $\alpha' = \mathsf{unlock}(j, k')$ then from (18) it must be the case that either $k \neq k'$ or $k = k'$ and $\Phi(k) = (\{i, j\} \uplus I, \text{S})$. In the first scenario we can find $\Phi_b = \Phi[k' \mapsto (I', \kappa'')]$ for $(\{j\} \uplus I', \kappa') = \Phi(k')$, while in the second one $\Phi_b = \Phi[k \mapsto (\{i\} \uplus I, \text{S})]$ with $j \notin I$. In both cases we pick $h_b = h, S_b = S[j \mapsto (s, \curlyvee)], \mathbb{P}_b = \dot{\mathbb{P}}$. for $s = S(j) \downarrow_1$.

(xiv) If $\alpha = \mathsf{lock}(i, k, \kappa)$ and $\alpha' = \mathsf{unlock}(j, k')$ then from (18) it must be the case that either $k \neq k'$ and we find $\Phi_b = \Phi[k' \mapsto (I', \kappa'')]$ for $(\{j\} \uplus I', \kappa') = \Phi(k')$ or $k = k'$ and $\Phi(k) = (\{j\} \uplus I, \text{S})$ and $\kappa = \text{S}$. In the latter case we can find $\Phi_b = \Phi[k \mapsto (I, \text{S})]$. In both cases we pick $h_b = h, S_b = S', \mathbb{P}_b = \dot{\mathbb{P}}$.

(xv) If $\alpha = \mathsf{alloc}(i, n, l)$ and $\alpha' = \mathsf{lock}(j, k, \kappa)$ then from (18) it must be the case that $k < l \vee k \geq l + n$. Now we can always find $h_b = h, \Phi_b = \Phi[k \mapsto (\{j\} \uplus I, \kappa)], S_b = S, \mathbb{P}_b = \dot{\mathbb{P}}$ for $I = \Phi(k) \downarrow_1$.

(xvi) If $\alpha = \mathsf{alloc}(i, n, l)$ and $\alpha' = \mathsf{unlock}(j, k)$ then from (18) it must be the case that $k < l \vee k \geq l + n$. Now we can always find $h_b = h, \Phi_b = \Phi[k \mapsto (I, \kappa')], S_b = S[j \mapsto (s, \curlyvee)], \mathbb{P}_b = \dot{\mathbb{P}}$, for $(\{j\} \uplus I, \kappa) = \Phi(k), \kappa' \leq \text{S}$ and $s = S(j) \downarrow_1$.

(xvii) If $\alpha = \mathsf{alloc}(i, n, l)$ and $\alpha' = \mathsf{alloc}(j, n', l')$ then from (18) it must be the case that $\{l, \ldots, l + n - 1\} \cap \{l', \ldots, l' + n' - 1\} \equiv \emptyset$. Now we can always find $h_b = h[l' \mapsto 0] \ldots [l' + n' - 1 \mapsto 0], \Phi_b = \Phi[l' \mapsto (\{j\}, \text{X})] \ldots [l' + n' - 1 \mapsto (\{j\}, \text{X})], S_b = S[j \mapsto (s[\mathtt{x} \mapsto l'], p)], \mathbb{P}_b = \dot{\mathbb{P}}$ for $(s, p) = S(j)$.

All other cases are either ruled out by assumption (19) or by the fact that they are unfeasible under the 2PL semantics. We provide one of such unfeasible examples for clarity. If $\alpha = \mathsf{write}(i, k, v)$ and $\alpha' = \mathsf{lock}(j, k, \text{S})$ for any $v \in \mathsf{Val}$ then in no possible way the original consecutive reduction could have happened. This is because α requires $\Phi(k) = (\{i\}, \text{X})$ while α' needs $\Phi_a(k) = (I, \kappa)$ for $\kappa \leq \text{S}$. We know that it must be the case that $\Phi_a = \Phi$ since α does not modify the lock manager. It is now clear that such situation could have never occurred. □

We now establish two predicates that will aid us in formulating the final statement of expressing equivalence between traces, where one is the same as the other, with one or more operations swapped according to specific rules. The latter concept is in fact described by the $\tau' = \mathsf{swap}(\tau, x, y)$ predicate, which sets a relationship between two traces by asserting that τ' is equivalent to τ with operations x and y swapped. On the other hand, $\mathsf{swappable}(\tau, x, y)$ indicates that when x and y are two consecutive actions as part of trace τ, they can be swapped without changing the final state that can be reached by the program execution.

Definition 7.32. (Swapped trace). A trace τ' is the *swapped* version of τ for some operations x and y if and only if τ' contains all of τ's operations in the same exact order a part from the one of x and y, which is swapped.

$$\begin{gathered} \tau' = \mathsf{swap}(\tau, x, y) \iff \\ \exists \alpha_x, n_x, \alpha_y, n_y.\, x = (\alpha_x, n_x) \wedge y = (\alpha_y, n_y) \wedge \\ \tau' = \tau \setminus \{(\alpha_x, n_x), (\alpha_y, n_y)\} \cup \{(\alpha_x, n_y), (\alpha_y, n_x)\} \end{gathered}$$

Definition 7.33. (Swappable trace actions). Two operations as part of a trace τ are *swappable* if and only if their indeces are consecutive and for any τ-generated full reduction, there is a point

where two consecutive reductions are labelled by them and they are allowed to be swapped.

$$\begin{gathered}\mathsf{swappable}(\tau, (\alpha, n), (\alpha', n')) \\ \Longleftrightarrow \\ n' = n + 1 \land \forall h, \underline{h}, \Phi, S, \mathbb{P}.\, \mathsf{tgen}(\tau, h, \underline{h}, \Phi, S, \mathbb{P}) \implies \\ ((\alpha, n) \in \tau \land (\alpha', n') \in \tau \land \exists h_1, h_2, h_a, h_b, \Phi_1, \Phi_2, \Phi_a, \Phi_b, S_1, S_2, S_a, S_b, S', \mathbb{P}, \mathbb{P}_1, \mathbb{P}_2, \mathbb{P}_a, \mathbb{P}_b. \\ (h, \Phi, S, \mathbb{P}) \rightarrow^* (h_1, \Phi_1, S_1, \mathbb{P}_1) \xrightarrow{\alpha} (h_a, \Phi_a, S_a, \mathbb{P}_a) \xrightarrow{\alpha'} (h_2, \Phi_2, S_2, \mathbb{P}_2) \rightarrow^* (\underline{h}, \emptyset, S', \mathtt{skip}) \\ \land\ (h_1, \Phi_1, S_1, \mathbb{P}_1) \xrightarrow{\alpha'} (h_b, \Phi_b, S_b, \mathbb{P}_b) \xrightarrow{\alpha} (h_2, \Phi_2, S_2, \mathbb{P}_2))\end{gathered}$$

Next, we combine the previous definitions and results into a single expressive statement which establishes trace equivalence among swapped transactions, in terms of the terminating state, i.e. storage, lock manager and transactions's stack.

Lemma 7.34. (Trace swap equivalence).

$$\begin{gathered}\forall h, \underline{h}, \Phi, S, \mathbb{P}, \tau, \tau', x, y. \\ \mathsf{tgen}(\tau, h, \underline{h}, \Phi, S, \mathbb{P}) \land \mathsf{swappable}(\tau, x, y) \land \tau' = \mathsf{swap}(\tau, x, y) \implies \mathsf{tgen}(\tau, h, \underline{h}, \Phi, S, \mathbb{P})\end{gathered}$$

Proof. Let's pick arbitrary $h, \underline{h} \in \mathsf{Storage}, \Phi \in \mathsf{LMan}, S \in \mathsf{TState}, \mathbb{P} \in \mathsf{Prog}, \tau, \tau' \in [\mathsf{Act} \times \mathbb{N}], x, y \in \mathsf{Act} \times \mathbb{N}$. We now assume that the following holds:

$$\mathsf{tgen}(\tau, h, \underline{h}, \Phi, S, \mathbb{P}) \land \mathsf{swappable}(\tau, x, y) \land \tau' = \mathsf{swap}(\tau, x, y)$$

The above means that we can generate the $\underline{h}$ storage from τ starting with state $(h, \Phi, S, \mathbb{P})$ and there are two operations $x = (\alpha, n), y = (\alpha', n+1)$ which are swappable as specified in Definition 7.33. We also know that another trace, τ', is equivalent to τ with operations x and y swapped. It is required to show that τ' can also generate $\underline{h}$ starting from $(h, \Phi, S, \mathbb{P})$.

From the definition of swappable, we know that the following must hold, where the non-bound variables are assumed to be existentially quantified for conciseness:

$$(h, \Phi, S, \mathbb{P}) \rightarrow^* (h_1, \Phi_1, S_1, \mathbb{P}_1) \xrightarrow{\alpha} (h_a, \Phi_a, S_a, \mathbb{P}_a) \tag{20}$$

$$\xrightarrow{\alpha'} (h_2, \Phi_2, S_2, \mathbb{P}_2) \rightarrow^* (\underline{h}, \emptyset, S', \mathtt{skip}) \tag{21}$$

$$\land\ (h_1, \Phi_1, S_1, \mathbb{P}_1) \xrightarrow{\alpha'} (h_b, \Phi_b, S_b, \mathbb{P}_b) \xrightarrow{\alpha} (h_2, \Phi_2, S_2, \mathbb{P}_2) \tag{22}$$

From (20) and the definition of swap we obtain that, following τ' for $n-1$ steps we can reduce $(h, \Phi, S, \mathbb{P}) \rightarrow^* (h_1, \Phi_1, S_1, \mathbb{P}_1)$. Next, we know that α and α' are swapped in τ', meaning that from (22) we can reduce $(h_1, \Phi_1, S_1, \mathbb{P}_1) \xrightarrow{\alpha'} (h_b, \Phi_b, S_b, \mathbb{P}_b) \xrightarrow{\alpha} (h_2, \Phi_2, S_2, \mathbb{P}_2)$ and we reach step $n+2$. From then on τ and τ' are equivalent, therefore the from (21) we reduce until `skip` to a state whose storage component is $\underline{h}$. □

7.2.3 Strict total order

Given that our goal is to find a way to compare traces produced under the 2PL operational semantics to the ones retrieved from the ATOM one, we need to establish a strict total order on the transactions that appear in a trace. The properties of such an order relation, enable us to effectively simulate a serial reduction, as we know that, from an abstract point of view, we can look at two parallel transactions as if one *happens before* the other.

The serialization graph structure $G = (N, E) = \mathsf{SG}(\tau)$, which was formalised in Definition 7.7, implicitly gives us a relation on the transactions participating to trace τ through E, the set of edges. The latter is in fact a partial order relation on the transaction identifiers which represents the set of ordered conflicts inside of a trace. All of the non-conflicting transactions

are accounted for by disconnected nodes in G, meaning that they are not ordered with respect to any of the others. It follows that E must be extended to include all of the transactions in N.

On the other hand, the edges relation is acyclic, as shown in Theorem 7.19, and therefore a great starting point from which to build the total order we need. As part of the building process, it is crucial to preserve the program order in which the transactions originally executed. For example, if we were to build the serialization graph for a trace that was generated by program $(\mathbb{T}_1; \mathbb{T}_2) \| \mathbb{T}_3$ and discover that the only edge was $2 \to 3$ we would need to make sure that our final order is going to be of the shape $\{1 \to 2, 1 \to 3, 2 \to 3\}$ since if we were to add edges $3 \to 1$ or $2 \to 1$, we would clash with the original program structure, which imposed the sequential composition of $\mathbb{T}_1$ and $\mathbb{T}_2$ and expects that all of $\mathbb{T}_2$'s operations happen after $\mathbb{T}_1$'s ones.

Definition 7.35. (Reflexive relation). The *reflexive relation* of a set X, written $\mathsf{Id}(X)$, is defined as:

$$\mathsf{Id}(X) = \{(x, x) \mid x \in X\}$$

Definition 7.36. (Reflexive closure). The *reflexive closure* of a given relation R on a set X, written R^{id}, is defined as:

$$R^{\mathsf{id}} = R \cup \mathsf{Id}(X)$$

Definition 7.37. (Relation composition). The *composition* of two relations R and S, written $R; S$, is defined as:

$$R; S = \{(a, b) \mid \exists c. (a, c) \in R \wedge (c, b) \in S\}$$

Composition on relations is associative.

The total transactions relation is build iteratively, by adding a pair of transaction identifiers at each step in the process, as described in [8]. Given a trace τ, we start by building its serialization graph $(N, E) = \mathsf{SG}(\tau)$ and setting the initial relation (step 0) to be the transitive closure of E, formally E^*. This addition is not going to include any cycles or links between disconnected nodes, as it will simply set *shortcut* edges between transactions that were on the same directed path.

Next, at every successive iteration, we pick two transactions i and j, such that $i < j$ and there is no immediate edge between them (in either way, i.e. $i \to j$ or $j \to i$) yet. We then add the $i \to j$ edge to the relation, together with all of the possible transitive combinations that result from the addition of the new edge. For example, if our current relation includes pairs $(1, 2), (3, 4)$, then by picking $i = 2$ and $j = 3$, we would add $(2, 3), (2, 4), (1, 3), (1, 4)$ as new entries. In fact, at every step, we compute the transitive closure of the relation: this will always guarantee its totality and its transitivity.

The preservation of the program order is achieved through the $i < j$ constraint on the relation build steps, which has a precise link to the operational semantics used to generate the traces we consider. In fact, if we consider once again the 2PL rules for reduction and in particular the START rule, we have the following:

$$\frac{\iota \in \{i \mid \forall j \in \mathsf{dom}(S).\, j < i\}}{(h, \Phi, S, \texttt{begin } \mathbb{C} \texttt{ end}) \xrightarrow{\mathsf{id}(\iota)} (h, \Phi, S[\iota \mapsto (\emptyset, \curlywedge)], \texttt{begin } \mathbb{C} \texttt{ end}_\iota)} \text{START}$$

The rule's premiss requires the identifier for the new transaction, namely ι, to be greater than the one of any existing transaction that has already started executing. This implies that any time we encounter a program of the shape $\mathbb{P} = \mathbb{T}; \mathbb{T}'$, we are sure that $\mathbb{T}$ will be assigned an identifier, a, which will always be less than the one assigned to $\mathbb{T}'$, i.e. b. We also know that, since $\mathbb{T}$ and $\mathbb{T}'$ are sequentially composed, there can never be an edge $b \to a$ in a serialization graph of a trace generated by $\mathbb{P}$, since all of $\mathbb{T}$'s operations will appear in the trace after the ones of $\mathbb{T}'$.

Figure 7.6 graphically shows all of the steps involved in the creation of the order we have described. The first step shows the serialization graph for the particular trace followed by the addition of the edges generated by the transitive closure of the starting relation. The next two steps illustrate the selection and addition of two nodes (shaded in blue) which are the i and j transactions we referred to earlier.

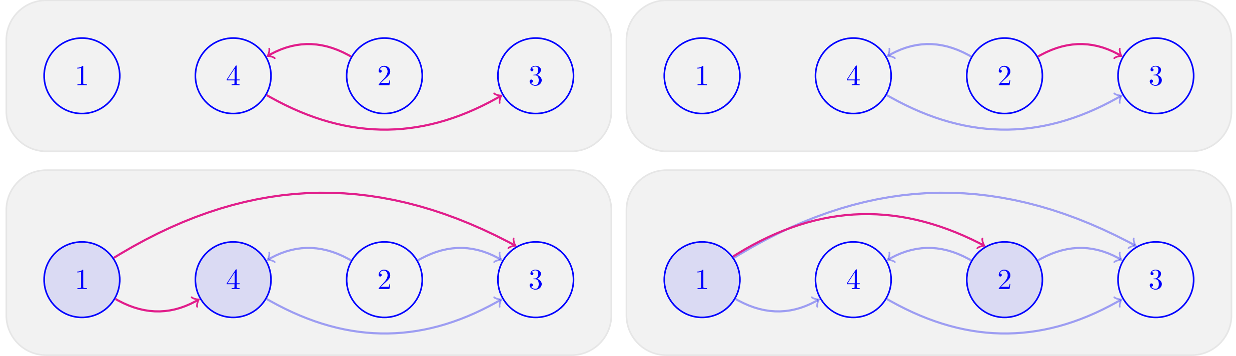


Figure 7.6: Build steps for the $\sqsubset$ order on a trace $\tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P})$ for some $h \in \mathsf{Storage}$, $S \in \mathsf{TState}$ and program $\mathbb{P} = (\mathbb{T}_1; \mathbb{T}_4) \parallel (\mathbb{T}_2; \mathbb{T}_3)$. The purple edges are the ones created as part of the current step.

Definition 7.38. (2PL Transactions order). Let $(N, E) = \mathsf{SG}(\tau)$ be the serialization graph in the definition of the *transactions order* $\sqsubset$ for trace τ:

$$
\begin{aligned}
\sqsubset_0 &= E^* \\
\sqsubset_{n+1} &= \sqsubset_n \cup \left(\sqsubset_n^{\mathsf{id}}; \{(i,j)\}; \sqsubset_n^{\mathsf{id}}\right) \\
&\quad \text{where } i, j \in N \text{ and } i < j \\
&\quad \text{and } i \not\sqsubset_n j \text{ and } j \not\sqsubset_n i
\end{aligned}
$$

Now that we have a formal definition of the transactions order, we are required to show that $\sqsubset$ is a strict total order on the transactions appearing as part of a trace. We need it to be both strict and total, since we are interested in finding the effective serial execution order within a program run, therefore given two transactions, we must be able to find which one happened first. The strictness is required, since reflexivity of the order of transactions' execution does not make sense in this context. It follows that we can always find the minimal element of the relation, and once removed, a new one will be present. This particular property will prove very useful when proving the semantics equivalence.

Theorem 7.39. (Order of transactions). The $\sqsubset$ relation is a strict total order on the set of transactions N in $(N, E) = \mathsf{SG}(\tau), \tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P}), \mathbb{P} \in \mathsf{Prog}, h \in \mathsf{Storage}, S \in \mathsf{TState}$.

Proof. In order to show the theorem, we are required to prove that, for all $a, b, c \in N$:

- (Irreflexivity). $a \not\sqsubset a$
- (Asymmetry). If $a \sqsubset b$ then $b \not\sqsubset a$
- (Transitivity). If $a \sqsubset b$ and $b \sqsubset c$ then $a \sqsubset c$
- (Totality). $a \sqsubset b$ or $b \sqsubset a$ or $a = b$

Let's pick an arbitrary program $\mathbb{P} \in \mathsf{Prog}$, initial storage $h \in \mathsf{Storage}$, transactions' state $S \in \mathsf{TState}$ and build one of its corresponding traces $\tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P})$. We now consider the incrementally built $\sqsubset$ relation on N, where $(N, E) = \mathsf{SG}(\tau)$.

(Irreflexivity). The proof follows by induction on the number of $\sqsubset$ relation construction steps, n. Let's pick an arbitrary transaction identifier $a \in N$.
Base case: $n = 0$
To show: $a \not\sqsubset_0 a$
By definition, we know that $\sqsubset_0 = E^*$, i.e. the transitive closure on the edges of the serialization graph $\mathsf{SG}(\tau)$. We directly obtain that $a \not\sqsubset_0 a$ from Theorem 7.19, since $\mathsf{SG}(\tau)$ contains no cycles.

Inductive case: $n > 0$
Inductive hypothesis: $a \not\sqsubset_n a$
To show: $a \not\sqsubset_{n+1} a$
Let's assume that $a \sqsubset_{n+1} a$ and, by definition, we know that this means that, for some $i, j \in N$ such that $i < j \wedge i \not\sqsubset_n j \wedge j \not\sqsubset_n i$ we have:

$$(a, a) \in \sqsubset_n \cup \left(\sqsubset_n^{\mathsf{id}}; \{(i, j)\}; \sqsubset_n^{\mathsf{id}}\right)$$

and by I.H. we can rewrite it as $(a, a) \in \left(\sqsubset_n^{\mathsf{id}}; \{(i, j)\}; \sqsubset_n^{\mathsf{id}}\right)$ given we assumed that $\sqsubset_n$ is irreflexive and therefore (a, a) cannot belong to it. It follows that it must be the case that (a, i) and (j, a) are in $\sqsubset_n^{\mathsf{id}}$ and moreover they must be in $\sqsubset_n$ given that $i < j$ meaning that $i \neq j$. By transitivity of $\sqsubset_n$, there must be a $(j, i) \in \sqsubset_n$. By contradiction we state that $a \not\sqsubset_{n+1} a$.

(Asymmetry). The proof follows by induction on the number of $\sqsubset$ relation construction steps, n. Let's pick arbitrary transaction identifiers $a, b \in N$.
Base case: $n = 0$
To show: $a \sqsubset_0 b \implies b \not\sqsubset_0 a$
By definition we know that $\sqsubset_0 = E^*$, i.e. the transitive closure on the edges of the serialization graph $\mathsf{SG}(\tau)$. Let's assume that $a \sqsubset_0 b$ meaning that $a \rightarrow^* b \in E$. We directly obtain that $b \not\sqsubset_0 a$ from Theorem 7.19, as $\mathsf{SG}(\tau)$ contains no cycles.

Inductive case: $n > 0$
Inductive hypothesis: $a \sqsubset_n b \implies b \not\sqsubset_n a$
To show: $a \sqsubset_{n+1} b \implies b \not\sqsubset_{n+1} a$
Let's assume that $a \sqsubset_{n+1} b$ and by definition we know this means that, for some $i, j \in N$ such that $i < j \wedge i \not\sqsubset_n j \wedge j \not\sqsubset_n i$ we have:

$$(a, b) \in \sqsubset_n \cup \left(\sqsubset_n^{\mathsf{id}}; \{(i, j)\}; \sqsubset_n^{\mathsf{id}}\right)$$

- If $a \sqsubset_n b$ we know by I.H. that $b \not\sqsubset_n a$. Let's instead assume that $(b, a) \in \left(\sqsubset_n^{\mathsf{id}}; \{(i, j)\}; \sqsubset_n^{\mathsf{id}}\right)$ from which it follows that there is a $(b, i) \in \sqsubset_n^{\mathsf{id}}$ and $(j, a) \in \sqsubset_n^{\mathsf{id}}$. By transitivity of $\sqsubset_n$ we obtain that $(j, i) \in \sqsubset_n^{\mathsf{id}}$ and moreover that $j \sqsubset_n i$ since $i \neq j$ as $i < j$ from our assumption. By contradiction we obtain that $(b, a) \notin \left(\sqsubset_n^{\mathsf{id}}; \{(i, j)\}; \sqsubset_n^{\mathsf{id}}\right)$. We conclude that $b \not\sqsubset_{n+1} a$.
- If $(a, b) \in \left(\sqsubset_n^{\mathsf{id}}; \{(i, j)\}; \sqsubset_n^{\mathsf{id}}\right)$ then this implies that there is a $(a, i) \in \sqsubset_n^{\mathsf{id}}$ and $(j, b) \in \sqsubset_n^{\mathsf{id}}$. Let's now assume that $(b, a) \in \left(\sqsubset_n^{\mathsf{id}}; \{(i, j)\}; \sqsubset_n^{\mathsf{id}}\right)$ meaning that there is a $(b, i) \in \sqsubset_n^{\mathsf{id}}$ and $(j, a) \in \sqsubset_n^{\mathsf{id}}$. By transitivity of $\sqsubset_n$ we obtain that $(j, i) \in \sqsubset_n^{\mathsf{id}}$ and moreover that $j \sqsubset_n i$ since $i \neq j$ as $i < j$. By contradiction we obtain that $(b, a) \notin \left(\sqsubset_n^{\mathsf{id}}; \{(i, j)\}; \sqsubset_n^{\mathsf{id}}\right)$. We now assume that $b \sqsubset_n a$ which implies that $(b, a) \in \sqsubset_n^{\mathsf{id}}$. By transitivity of $\sqsubset_n$ we obtain that $(j, i) \in \sqsubset_n^{\mathsf{id}}$ and moreover that $j \sqsubset_n i$ since $i \neq j$ as $i < j$. By contradiction we obtain that $(b, a) \notin \sqsubset_n$. We conclude that $b \not\sqsubset_{n+1} a$.

(Transitivity). We are required to show that $\forall m \geq 1. \sqsubset^m \subseteq \sqsubset$ The proof follows by induction on the number of self-composition steps, m.
Base case: $m = 1$
To show: $\sqsubset^1 \subseteq \sqsubset$
The result follows directly by definition $\sqsubset^1 = \sqsubset \subseteq \sqsubset$.

Inductive case: $m > 1$
Inductive hypothesis: $\sqsubset^m \subseteq \sqsubset$
To show: $\sqsubset^{m+1} \subseteq \sqsubset$

$$\begin{aligned} \sqsubset^{m+1} &= \sqsubset^m ; \sqsubset \text{ by associativity} \\ &\subseteq \sqsubset ; \sqsubset \text{ by I.H.} \\ &\subseteq \sqsubset^2 \text{ by definition} \\ &\subseteq \sqsubset \text{ by Lemma 7.40} \end{aligned}$$

(Totality). Let's pick arbitrary transaction identifiers $a, b \in N$ (I) for a finite N and build the $\sqsubset$ relation on it until convergence, i.e. in a finite number of steps. If $(a, b) \in E^*$ or $(b, a) \in E^*$ then we know that either $a \sqsubset b$ or $b \sqsubset a$ holds. On the other hand if there is no edge connecting a to b or b to a in E^* (II) then:

- If $a = b$ then by irreflexivity of $\sqsubset$ we are done, as totality is met.
- Without loss of generality, we say that $a < b$ (III). Given that the construction of $\sqsubset$ terminated in some $m > 0$ steps (being N a finite set), by (I), (II) and (III) we know that there must exist a construction step n such that $0 < n < m$ where the tuple (a, b) was inserted in the relation given that $\sqsubset_{n-1} \cup \left(\sqsubset_{n-1}^{\mathsf{id}}; \{(a, b)\}; \sqsubset_{n-1}^{\mathsf{id}}\right) \implies a \sqsubset_n b \implies a \sqsubset b$.

□

Theorem 7.39 used the following lemma as the key component to prove the transitivity of the transactions order. Lemma 7.40 shows that the relation composition of $\sqsubset$ with itself is included in $\sqsubset$.

Lemma 7.40. Given a serialization graph $(N, E) = \mathsf{SG}(\tau)$ for $\tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P}), \mathbb{P} \in \mathsf{Prog}, h \in \mathsf{Storage}, S \in \mathsf{TState}$, and the $\sqsubset$ relation on the set N we have that $\sqsubset^2 \subseteq \sqsubset$.

Proof. We proceed by induction on the number of $\sqsubset$ construction steps, n.

Base case: $n = 0$
To show: $\sqsubset_0^2 \subseteq \sqsubset_0$
By definition we know that $\sqsubset_0 = E^*$, i.e. the transitive closure on the edges of the serialization graph $\mathsf{SG}(\tau)$. It follows that by definition of transitive closure, $\sqsubset_0^2 = E^*; E^* = E^*$ meaning that $\sqsubset_0^2 \subseteq \sqsubset_0$.

Inductive case: $n > 0$
Inductive hypothesis: $\sqsubset_n^2 \subseteq \sqsubset_n$
To show: $\sqsubset_{n+1}^2 \subseteq \sqsubset_{n+1}$

We can rewrite the formula to be proven as the following, for some $i, j \in N$ such that $i < j \wedge i \not\sqsubset_n j \wedge j \not\sqsubset_n i$:

$$\left(\sqsubset_n \cup \underbrace{\left(\sqsubset_n^{\mathsf{id}}; \{(i,j)\}; \sqsubset_n^{\mathsf{id}}\right)}_{R}\right); \left(\sqsubset_n \cup \left(\sqsubset_n^{\mathsf{id}}; \{(i,j)\}; \sqsubset_n^{\mathsf{id}}\right)\right) \subseteq \sqsubset_{n+1} \tag{23}$$

$$\underbrace{\sqsubset_n; \sqsubset_n}_{\mathrm{a}} \cup \underbrace{\sqsubset_n; R}_{\mathrm{b}} \cup \underbrace{R; \sqsubset_n}_{\mathrm{c}} \cup \underbrace{R; R}_{\mathrm{d}} \subseteq \sqsubset_{n+1} \text{ by distributivity} \tag{24}$$

It now suffices to show that each of the unioned sets in the l.h.s. of (24) is a subset of $\sqsubset_{n+1}$ itself. We start by proving that both sets $S = \sqsubset_n; \sqsubset_n^{\mathsf{id}}$ and $S' = \sqsubset_n^{\mathsf{id}}; \sqsubset_n$ are subsets of $\sqsubset_n^{\mathsf{id}}$ which will become useful in the following cases.

$$\begin{aligned} S &= \sqsubset_n; \sqsubset_n^{\mathsf{id}} \\ &= \sqsubset_n; (\sqsubset_n \cup \mathsf{Id}(N)) \\ &= \sqsubset_n; \sqsubset_n \cup \sqsubset_n; \mathsf{Id}(N) \\ &= \sqsubset_n^2 \cup \sqsubset_n \\ &\subseteq \sqsubset_n \cup \sqsubset_n \text{ by I.H.} \\ &\subseteq \sqsubset_n^{\mathsf{id}} \text{ by definition} \end{aligned} \qquad \begin{aligned} S' &= \sqsubset_n^{\mathsf{id}}; \sqsubset_n \\ &= (\sqsubset_n \cup \mathsf{Id}(N)); \sqsubset_n \\ &= \sqsubset_n; \sqsubset_n \cup \mathsf{Id}(N); \sqsubset_n \\ &= \sqsubset_n^2 \cup \sqsubset_n \\ &\subseteq \sqsubset_n \cup \sqsubset_n \text{ by I.H.} \\ &\subseteq \sqsubset_n^{\mathsf{id}} \text{ by definition} \end{aligned}$$

Next, we show that the set arising from the pre- and post-composition of the singleton relation $\{(i,j)\}$ with $\sqsubset_n^{\mathsf{id}}$, formally $S'' = \{(i,j)\}; \sqsubset_n^{\mathsf{id}}; \{(i,j)\}$ is always empty.

$$\begin{aligned} S'' = \{(i,j)\}; \sqsubset_n^{\mathsf{id}}; \{(i,j)\} &= \{(i,j)\}; (\sqsubset_n \cup \mathsf{Id}(N)); \{(i,j)\} \\ &= \{(i,j)\}; (\sqsubset_n; \{(i,j)\} \cup \mathsf{Id}(N); \{(i,j)\}) \\ &= \{(i,j)\}; (\sqsubset_n; \{(i,j)\} \cup \{(i,j)\}) \\ &= (\{(i,j)\}; \sqsubset_n; \{(i,j)\}) \cup (\{(i,j)\}; \{(i,j)\}) \\ &= (\{(i,j)\}; \sqsubset_n; \{(i,j)\}) = \emptyset \end{aligned}$$

The final step is justified by the fact that we assumed that neither (i,j) nor (j,i) are part of the $\sqsubset_n$ relation, and the only way not to have a resulting empty set in this case, would be to have $j \sqsubset_n i$, which is clearly impossible.

a) *To show*: $\sqsubset_n; \sqsubset_n \subseteq \sqsubset_{n+1}$

$$\begin{aligned} \sqsubset_n; \sqsubset_n &= \sqsubset_n^2 \\ \text{by I.H.} &\subseteq \sqsubset_n \subseteq \sqsubset_{n+1} \end{aligned}$$

b) *To show*: $\sqsubset_n; \left(\sqsubset_n^{\mathsf{id}}; \{(i,j)\}; \sqsubset_n^{\mathsf{id}}\right) \subseteq \sqsubset_{n+1}$

$$\begin{aligned} \sqsubset_n; \left(\sqsubset_n^{\mathsf{id}}; \{(i,j)\}; \sqsubset_n^{\mathsf{id}}\right) &= \left(\sqsubset_n; \sqsubset_n^{\mathsf{id}}; \{(i,j)\}\right); \sqsubset_n^{\mathsf{id}} \text{ by associativity} \\ &= (S; \{(i,j)\}); \sqsubset_n^{\mathsf{id}} \text{ by associativity} \\ &\subseteq \sqsubset_n^{\mathsf{id}}; \{(i,j)\}; \sqsubset_n^{\mathsf{id}} \\ &\subseteq \sqsubset_{n+1} \end{aligned}$$

c) *To show*: $\left(\sqsubset_n^{\mathsf{id}}; \{(i,j)\}; \sqsubset_n^{\mathsf{id}}\right); \sqsubset_n \subseteq \sqsubset_{n+1}$

$$\begin{aligned} \left(\sqsubset_n^{\mathsf{id}}; \{(i,j)\}; \sqsubset_n^{\mathsf{id}}\right); \sqsubset_n &= \sqsubset_n^{\mathsf{id}}; \left(\{(i,j)\}; \sqsubset_n^{\mathsf{id}}; \sqsubset_n\right) \text{ by associativity} \\ &= \sqsubset_n^{\mathsf{id}}; (\{(i,j)\}; S') \text{ by associativity} \\ &\subseteq \sqsubset_n^{\mathsf{id}}; \{(i,j)\}; \sqsubset_n^{\mathsf{id}} \\ &\subseteq \sqsubset_{n+1} \end{aligned}$$

d) *To show*: $\left(\sqsubset_n^{\mathsf{id}}; \{(i,j)\}; \sqsubset_n^{\mathsf{id}}\right) ; \left(\sqsubset_n^{\mathsf{id}}; \{(i,j)\}; \sqsubset_n^{\mathsf{id}}\right) \subseteq \sqsubset_{n+1}$

$$\begin{aligned}
\left(\sqsubset_n^{\mathsf{id}}; \{(i,j)\}; \sqsubset_n^{\mathsf{id}}\right) ; \left(\sqsubset_n^{\mathsf{id}}; \{(i,j)\}; \sqsubset_n^{\mathsf{id}}\right) &= \left(\sqsubset_n^{\mathsf{id}}; \{(i,j)\}\right) ; \left(\sqsubset_n^{\mathsf{id}}; \left(\sqsubset_n^{\mathsf{id}}; \{(i,j)\}; \sqsubset_n^{\mathsf{id}}\right)\right) \\
&= \left(\sqsubset_n^{\mathsf{id}}; \{(i,j)\}\right) ; \left(\left(\sqsubset_n^{\mathsf{id}}; \sqsubset_n^{\mathsf{id}}\right) ; \left(\{(i,j)\}; \sqsubset_n^{\mathsf{id}}\right)\right) \\
&= \left(\sqsubset_n^{\mathsf{id}}; \{(i,j)\}\right) ; \left(\sqsubset_n^{\mathsf{id}}; \left(\{(i,j)\}; \sqsubset_n^{\mathsf{id}}\right)\right) \\
&= \sqsubset_n^{\mathsf{id}}; \left(\{(i,j)\}; \sqsubset_n^{\mathsf{id}}; \{(i,j)\}\right) ; \sqsubset_n^{\mathsf{id}} \\
&= \sqsubset_n^{\mathsf{id}}; \emptyset; \sqsubset_n^{\mathsf{id}} \\
&= \emptyset \subseteq \sqsubset_{n+1}
\end{aligned}$$

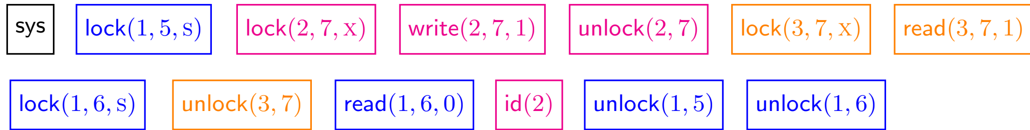
□

7.2.4 Proof

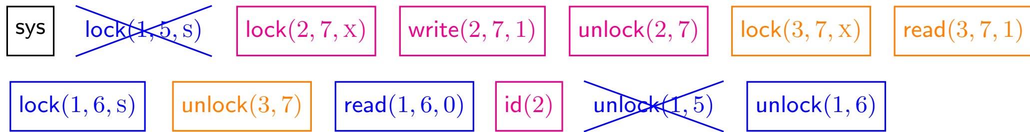
We are ready to tackle the general proof which allows us to show that any full program reduction that happens under the 2PL semantics can be replicated by the ATOM ones. This relation will be verified with regards to the final storage, achieved by the reductions under the two different semantics. The first level of proof is an induction on the syntactic structure of programs.

The single transaction case `begin` $\mathbb{C}$ `end` follows from the single step equivalence of the two operational semantics when reducing sequential commands (i.e. the transaction's body $\mathbb{C}$). In the case of a loop $\mathbb{P}^*$ and of nondeterministic choice $\mathbb{P}_1 + \mathbb{P}_2$ we get the needed result from the inductive hypothesys on $\mathbb{P}$ and $\mathbb{P}_1, \mathbb{P}_2$ respectively. The sequential composition of programs is dealt with using a series of auxiliary lemmata together with the inductive hypothesis on $\mathbb{P}_1, \mathbb{P}_2$. Parallel composition is, as expected, the most challenging case, and its proof requires a particular multi-step strategy summarized at an intuitive level below.

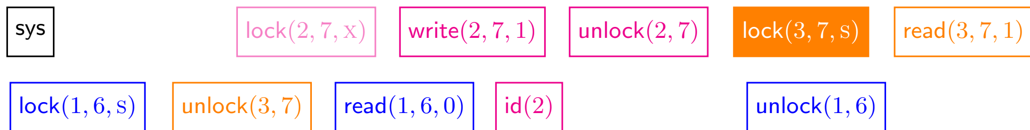
1. Retrieve a full trace τ from the terminating reduction of $\mathbb{P}_1 \| \mathbb{P}_2$.



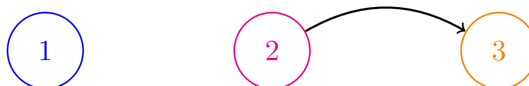
2. Clean τ from any spurious locks that appear inside of it in order to obtain τ'.



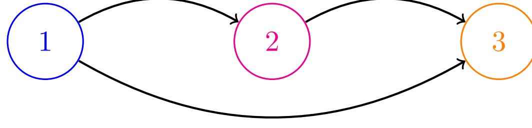
3. Convert any redundant exclusive lock in τ' into a shared one and compute τ_c.



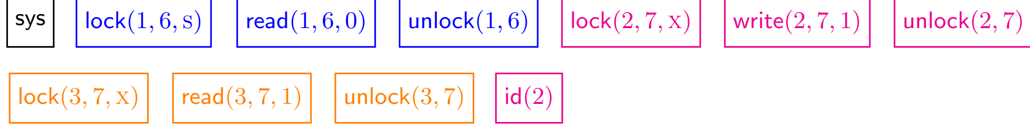
4. Build a serialization graph out of τ_c.



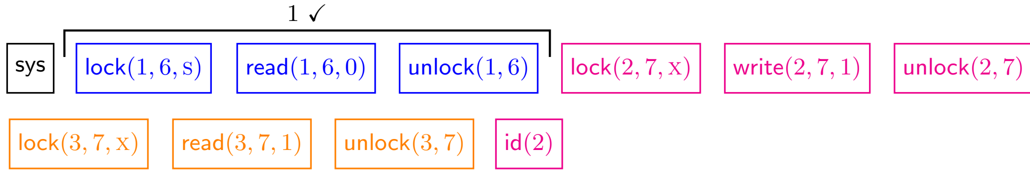
5. Extend the serialization graph to the strict total order $\sqsubset$ and find its minimal element ι.



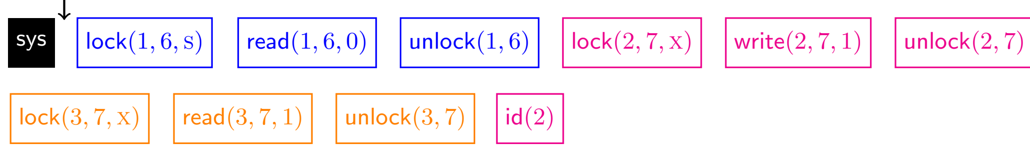
6. Swap all of ι's operations to the left of the trace until no swap is possible anymore. The final trace will be τ_{seq}.



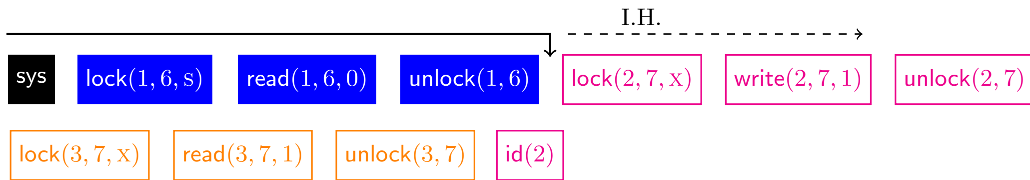
7. Show that no operation done by another transaction can appear in τ_{seq} before one done by ι.



8. Replicate any system transition labelled with sys in the Atom semantics, knowing that the state does not change.



9. Use the semantics equivalence for a single transaction together with the inductive hypothesis to conclude the proof.



Theorem 7.41.

$$\forall h, h', S, S'. (h, \emptyset, S, \mathbb{P}) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}) \rightarrow^*_{\text{ATOM}} (h', \texttt{skip})$$

Proof. The proof is done by induction on the structure of programs Prog.

Base case 1: $\texttt{skip} \in \mathsf{Prog}$
To show:

$$\forall h, h', S, S'. (h, \emptyset, S, \texttt{skip}) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \texttt{skip}) \rightarrow^*_{\text{ATOM}} (h', \texttt{skip})$$

For arbitrary h, h', S, S' we assume that $(h, \emptyset, S, \texttt{skip}) \rightarrow^* (h', \emptyset, S', \texttt{skip})$ holds, and given that skip has no possible one-step reductions, it must be the case that it is a zero-step reduction.

Therefore we have $h = h'$ and $S = S'$. Starting from $(h, \texttt{skip})$ through the $\rightarrow_{\text{Atom}}$ relation, we can always reach $(h, \texttt{skip})$ via a zero-step reduction $(h, \texttt{skip}) \rightarrow^0_{\text{Atom}} (h, \texttt{skip})$. We can conclude that $(h, \emptyset, S, \texttt{skip}) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \texttt{skip}) \rightarrow^*_{\text{Atom}} (h', \texttt{skip})$ where $h = h'$.

Base case 2: $\mathbb{T} \in \mathsf{Prog}$
To show:

$$\forall h, h', S, S'. (h, \emptyset, S, \mathbb{T}) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{T}) \rightarrow^*_{\text{Atom}} (h', \texttt{skip})$$

We will proceed with the proof by induction on the structure of transactions Trans. Given that the Atom semantics only support user transactions, all that is required to show is:

$$\forall h, h', S, S'.$$
$$(h, \emptyset, S, \texttt{begin } \mathbb{C} \texttt{ end}) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \texttt{begin } \mathbb{C} \texttt{ end}) \rightarrow^*_{\text{Atom}} (h', \texttt{skip})$$

For arbitrary h, h', S, S' we assume that $(h, \emptyset, S, \texttt{begin } \mathbb{C} \texttt{ end}) \rightarrow^* (h', \emptyset, S', \texttt{skip})$ holds. Given the overall reduction from $\texttt{begin } \mathbb{C} \texttt{ end}$ to $\texttt{skip}$ it must be the case that the following holds.

$$(h, \emptyset, S, \texttt{begin } \mathbb{C} \texttt{ end}) \xrightarrow{\mathsf{id}(\iota)} (h, \emptyset, S[\iota \mapsto (\emptyset, \curlywedge)], \texttt{begin } \mathbb{C} \texttt{ end}_\iota)$$
$$\rightarrow^* (h', \emptyset, S', \texttt{begin skip end}_\iota) \xrightarrow{\mathsf{sys}} (h', \emptyset, S', \texttt{skip})$$

Which implies that $\mathbb{C}$ reduces to $\texttt{skip}$ through the repeated use of the Exec rule. From the transitive closure of the $\rightarrow$ relation and Lemma C.5 we obtain the result that $(h, \texttt{begin } \mathbb{C} \texttt{ end}) \rightarrow^*_{\text{Atom}} (h', \texttt{skip})$.

Inductive case 1: $\mathbb{P}_1 + \mathbb{P}_2 \in \mathsf{Prog}$
To show:

$$\forall h, h', S, S'. (h, \emptyset, S, \mathbb{P}_1 + \mathbb{P}_2) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}_1 + \mathbb{P}_2) \rightarrow^*_{\text{Atom}} (h', \texttt{skip})$$

Inductive hypothesis:

$$\forall h, h', S, S'. (h, \emptyset, S, \mathbb{P}_1) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}_1) \rightarrow^*_{\text{Atom}} (h', \texttt{skip})$$
$$\wedge$$
$$\forall h, h', S, S'. (h, \emptyset, S, \mathbb{P}_2) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}_2) \rightarrow^*_{\text{Atom}} (h', \texttt{skip})$$

For arbitrary h, h', S, S' we assume that $(h, \emptyset, S, \mathbb{P}_1 + \mathbb{P}_2) \rightarrow^* (h', \emptyset, S', \texttt{skip})$ holds. Now we are presented with two cases:

1. We can reduce $(h, \emptyset, S, \mathbb{P}_1 + \mathbb{P}_2) \xrightarrow{\mathsf{sys}} (h, \emptyset, S, \mathbb{P}_1)$ with one step through the ChoiceL rule, which we can always apply since it has an empty premiss. We can also always reduce $(h, \mathbb{P}_1 + \mathbb{P}_2) \rightarrow_{\text{Atom}} (h, \mathbb{P}_1)$ through the rule AtChoiceL given it has an empty premiss. By inductive hypothesis on $\mathbb{P}_1$ we obtain that $(h, \emptyset, S, \mathbb{P}_1) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}_1) \rightarrow^*_{\text{Atom}} (h', \texttt{skip})$. Therefore we can conclude that:

$$(h, \emptyset, S, \mathbb{P}_1 + \mathbb{P}_2) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}_1 + \mathbb{P}_2) \rightarrow^*_{\text{Atom}} (h', \texttt{skip})$$

2. We can reduce $(h, \emptyset, S, \mathbb{P}_1 + \mathbb{P}_2) \xrightarrow{\mathsf{sys}} (h, \emptyset, S, \mathbb{P}_2)$ with one step through the ChoiceR rule, which we can always apply since it has an empty premiss. We can also always reduce $(h, \mathbb{P}_1 + \mathbb{P}_2) \rightarrow_{\text{Atom}} (h, \mathbb{P}_2)$ through the rule AtChoiceR given it has an empty premiss. By inductive hypothesis on $\mathbb{P}_2$ we obtain that $(h, \emptyset, S, \mathbb{P}_2) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}_2) \rightarrow^*_{\text{Atom}} (h', \texttt{skip})$. Therefore we can conclude that:

$$(h, \emptyset, S, \mathbb{P}_1 + \mathbb{P}_2) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}_1 + \mathbb{P}_2) \rightarrow^*_{\text{Atom}} (h', \texttt{skip})$$

Inductive case 2: $\mathbb{P}_1; \mathbb{P}_2 \in \mathsf{Prog}$
To show:

$$\forall h, h', S, S'. (h, \emptyset, S, \mathbb{P}_1; \mathbb{P}_2) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}_1; \mathbb{P}_2) \rightarrow^*_{\text{ATOM}} (h', \texttt{skip})$$

Inductive hypothesis:

$$\forall h, h', S, S'. (h, \emptyset, S, \mathbb{P}_1) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}_1) \rightarrow^*_{\text{ATOM}} (h', \texttt{skip})$$
$$\wedge$$
$$\forall h, h', S, S'. (h, \emptyset, S, \mathbb{P}_2) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}_2) \rightarrow^*_{\text{ATOM}} (h', \texttt{skip})$$

For arbitrary h, h', S, S' we assume that $(h, \emptyset, S, \mathbb{P}_1; \mathbb{P}_2) \rightarrow^* (h', \emptyset, S', \texttt{skip})$ holds. Given the overall reduction from $\mathbb{P}_1; \mathbb{P}_2$ to $\texttt{skip}$ we must have a chain of reductions of the following shape, for some h'', S'', Φ'' and where $\Phi'' = \emptyset$ by Lemma C.8.

$$\underbrace{(h, \emptyset, S, \mathbb{P}_1; \mathbb{P}_2) \rightarrow^* (h'', \Phi'', S'', \texttt{skip}; \mathbb{P}_2)}_{\text{(I)}} \xrightarrow{\mathsf{sys}} (h'', \emptyset, S'', \mathbb{P}_2) \rightarrow^* (h', \emptyset, S', \texttt{skip})$$

1. By (I) and Lemma C.7 we get that $(h, \emptyset, S, \mathbb{P}_1) \rightarrow^* (h'', \emptyset, S'', \texttt{skip})$ holds.
2. By 1. and the inductive hypothesis on $\mathbb{P}_1$ we obtain that $(h, \mathbb{P}_1) \rightarrow^*_{\text{ATOM}} (h'', \texttt{skip})$.
3. By 2. and Lemma C.6 we get that $(h, \mathbb{P}_1; \mathbb{P}_2) \rightarrow^*_{\text{ATOM}} (h'', \texttt{skip}; \mathbb{P}_2)$.

At this point we can apply the following rules in the two semantics:

$$(h'', \emptyset, S'', \texttt{skip}; \mathbb{P}_2) \xrightarrow{\mathsf{sys}} (h'', \emptyset, S'', \mathbb{P}_2) \quad \text{via PSeqSkip}$$
$$(h'', \texttt{skip}; \mathbb{P}_2) \rightarrow_{\text{ATOM}} (h'', \mathbb{P}_2) \quad \text{via AtPSeqSkip}$$

By inductive hypothesis on $\mathbb{P}_2$ we can conclude that $(h, \emptyset, S, \mathbb{P}_1; \mathbb{P}_2) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}_1; \mathbb{P}_2) \rightarrow^*_{\text{ATOM}} (h', \texttt{skip})$.

Inductive case 3: $\mathbb{P}^* \in \mathsf{Prog}$
To show:

$$\forall h, h', S, S'. (h, \emptyset, S, \mathbb{P}^*) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}^*) \rightarrow^*_{\text{ATOM}} (h', \texttt{skip})$$

Inductive hypothesis:

$$\forall h, h', S, S'. (h, \emptyset, S, \mathbb{P}) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}) \rightarrow^*_{\text{ATOM}} (h', \texttt{skip})$$

We prove this case by mathematical induction on n, the number of $\rightarrow^*$ reduction steps.

Base case 3.1: $n = 2$
To show:

$$\forall h, h', S, S', \mathbb{P}. (h, \emptyset, S, \mathbb{P}^*) \rightarrow^2 (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}^*) \rightarrow^*_{\text{ATOM}} (h', \texttt{skip})$$

For arbitrary $h, h', S, S', \mathbb{P}$ we assume that $(h, \emptyset, S, \mathbb{P}^*) \rightarrow^2 (h', \emptyset, S', \texttt{skip})$ holds. The only possible reduction that is able to bring $\mathbb{P}^*$ to $\texttt{skip}$ in exactly 2 steps is the following:

$$(h, \emptyset, S, \mathbb{P}^*) \xrightarrow{\mathsf{sys}} (h, \emptyset, S, \texttt{skip} + (\mathbb{P}; \mathbb{P}^*)) \xrightarrow{\mathsf{sys}} (h, \emptyset, S, \texttt{skip})$$

where the storage h is left unchanged. This result follows from the application of semantic rules Loop and ChoiceL. This implies that we can replicate the same reduction in the Atom

semantics using rules AtLoop and AtChoiceL that will bring us to a final state where the storage component is not changed.

$$(h, \mathbb{P}^*) \to_{\text{Atom}} (h, \texttt{skip} + (\mathbb{P}; \mathbb{P}^*)) \to_{\text{Atom}} (h, \texttt{skip})$$

Inductive case 3.2: $n > 2$
Inductive hypothesis:

$$\forall\, 2 \leq m < n, h, h', S, S', \mathbb{P}.$$
$$(h, \emptyset, S, \mathbb{P}^*) \to^m (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}^*) \to^*_{\text{Atom}} (h', \texttt{skip})$$

To show:

$$\forall h, h', S, S', \mathbb{P}.\, (h, \emptyset, S, \mathbb{P}^*) \to^{n+1} (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}^*) \to^*_{\text{Atom}} (h', \texttt{skip})$$

For arbitrary $h, h', S, S', \mathbb{P}$ we assume that $(h, \emptyset, S, \mathbb{P}^*) \to^{n+1} (h', \emptyset, S', \texttt{skip})$. From here we can always apply the Loop rule , as it does not have any requirements on the state, in order to get:

$$(h, \emptyset, S, \mathbb{P}^*) \xrightarrow{\mathsf{sys}} (h, \emptyset, S, \texttt{skip} + (\mathbb{P}; \mathbb{P}^*)) \to^* (h', \emptyset, S', \texttt{skip}) \tag{25}$$

From (25) there are two possible cases to consider:

1. We utilize rule ChoiceL in order to reduce $(h, \emptyset, S, \texttt{skip} + (\mathbb{P}; \mathbb{P}^*)) \xrightarrow{\mathsf{sys}} (h, \emptyset, S, \texttt{skip})$, which we can always do. It is now possible to replicate the same reduction in the Atom semantics by applying rule AtChoiceL, which reduces:

   $$(h, \texttt{skip} + (\mathbb{P}; \mathbb{P}^*)) \to_{\text{Atom}} (h, \texttt{skip})$$

   In this scenario we directly obtain the final result.

2. We reduce $(h, \emptyset, S, \texttt{skip} + (\mathbb{P}; \mathbb{P}^*)) \xrightarrow{\mathsf{sys}} (h, \emptyset, S, \mathbb{P}; \mathbb{P}^*)$ through the ChoiceR rule which we can always do, together with AtChoiceR that reduces in the following way:

   $$(h, \texttt{skip} + (\mathbb{P}; \mathbb{P}^*)) \to_{\text{Atom}} (h, \mathbb{P}; \mathbb{P}^*)$$

   From Lemma C.7, Lemma C.6 and the top-level inductive hypothesis on $\mathbb{P}$ we get that:

   $$(h, \emptyset, S, \mathbb{P}; \mathbb{P}^*) \to^* (h'', \emptyset, S'', \texttt{skip}; \mathbb{P}^*) \implies (h, \mathbb{P}; \mathbb{P}^*) \to^*_{\text{Atom}} (h'', \texttt{skip}; \mathbb{P}^*) \tag{26}$$

   It is now possible to further reduce the states in (26):

   $$(h'', \emptyset, S'', \texttt{skip}; \mathbb{P}^*) \xrightarrow{\mathsf{sys}} (h'', \emptyset, S'', \mathbb{P}^*) \quad \text{via PSeqSkip}$$
   $$(h'', \texttt{skip}; \mathbb{P}^*) \to_{\text{Atom}} (h'', \mathbb{P}^*) \quad \text{via AtPSeqSkip}$$

   We have now completed a number of $\to$ reduction steps greater than 1 meaning that we can use the *inductive hypothesis 3.2* to obtain the required result.

Inductive case 4: $\mathbb{P}_1 \| \mathbb{P}_2 \in \mathsf{Prog}$
To show:

$$\forall h, h', S, S'.\, (h, \emptyset, S, \mathbb{P}_1 \| \mathbb{P}_2) \to^* (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}_1 \| \mathbb{P}_2) \to^*_{\text{Atom}} (h', \texttt{skip})$$

We will prove the parallel composition case by mathematical induction on the number of reduction steps n.

Base case: $n = 1$
To show:

$$\forall h, h', S, S', \mathbb{P}_1, \mathbb{P}_2. \\ (h, \emptyset, S, \mathbb{P}_1 \| \mathbb{P}_2) \to (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}_1 \| \mathbb{P}_2) \to^*_{\text{ATOM}} (h', \texttt{skip})$$

For arbitrary $h, h', S, S', \mathbb{P}_1, \mathbb{P}_2$ we assume that $(h, \emptyset, S, \mathbb{P}_1 \| \mathbb{P}_2) \to (h', \emptyset, S', \texttt{skip})$ holds. Since a one-step reduction happened, it must be the case that $\mathbb{P}_1 = \mathbb{P}_2 = \texttt{skip}$, $h' = h$ and $\texttt{skip} \| \texttt{skip}$ reduced to $\texttt{skip}$ through the ParEnd rule. Now, we immediately obtain that $(h, \texttt{skip} \| \texttt{skip}) \to_{\text{ATOM}} (h, \texttt{skip})$ from rule AtParEnd.

Inductive case 4.1: $n > 1$
Inductive hypothesis:

$$\forall\ 1 \leq m \leq n, h, h', S, S', \mathbb{P}_1, \mathbb{P}_2. \\ (h, \emptyset, S, \mathbb{P}_1 \| \mathbb{P}_2) \to^m (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}_1 \| \mathbb{P}_2) \to^*_{\text{ATOM}} (h', \texttt{skip})$$

To show:

$$\forall h, h', \Phi, S, S', \mathbb{P}_1, \mathbb{P}_2. \\ (h, \emptyset, S, \mathbb{P}_1 \| \mathbb{P}_2) \to^{n+1} (h', \emptyset, S', \texttt{skip}) \implies (h, \mathbb{P}_1 \| \mathbb{P}_2) \to^*_{\text{ATOM}} (h', \texttt{skip})$$

For arbitrary $h, h', S, S', \mathbb{P}_1, \mathbb{P}_2$ we assume that $(h, \emptyset, S, \mathbb{P}_1 \| \mathbb{P}_2) \to^{n+1} (h', \emptyset, S', \texttt{skip})$ holds. As a consequence, from the definition of trace and tgen we can state there is a trace $\tau \in [\mathsf{Act} \times \mathbb{N}]$ of length $n + 1$ such that:

$$\tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P}_1 \| \mathbb{P}_2) \wedge \mathsf{tgen}(\tau, h, h', \emptyset, S, \mathbb{P}_1 \| \mathbb{P}_2) \tag{27}$$

By repeatedly applying Lemma C.1 until convergence, we obtain a trace $\tau_c \in [\mathsf{Act} \times \mathbb{N}]$ such that τ_c does not contain spurious lock and unlock operations, i.e. $\mathsf{clean}(\tau_c)$ holds, and for which the following is true, from (27).

$$\mathsf{tgen}(\tau_c, h, h', \emptyset, S, \mathbb{P}_1 \| \mathbb{P}_2) \tag{28}$$

From Theorem 7.39 we are always able to build the strict total order $\sqsubset$ on the set of transactions N that appear in τ_c, for which $(N, E) = \mathsf{SG}(\tau)$.
From the definition of strict total order, we know we can find the minimal (or first) element ι of $\sqsubset$ such that:

$$\forall j \in N. \iota \neq j \implies \iota \sqsubset j \tag{29}$$

From (28) we know that there must be an overall reduction of the following shape, as imposed by trace τ_c:

$$(h, \emptyset, S, \mathbb{P}_1 \| \mathbb{P}_2) \to^* (h', \emptyset, S', \texttt{skip}) \tag{30}$$

Given that ι's actions must appear inside trace τ_c, let us consider the following consecutive reductions as a consequence of (30):

$$\begin{gathered} (h, \emptyset, S, \mathbb{P}_1 \| \mathbb{P}_2) \to^* \\ (h_a, \Phi_a, S_a, \mathbb{P}_a) \xrightarrow{\alpha} (h_b, \Phi_b, S_b, \mathbb{P}_b) \xrightarrow{\alpha'} (h_c, \Phi_c, S_c, \mathbb{P}_c) \\ \to^* (h', \emptyset, S', \texttt{skip}) \end{gathered} \tag{31}$$

Whenever we encounter a situation like the one in (31), and α' is an action performed by transaction ι, i.e. $\alpha' = \alpha(\iota)$, we apply Lemma 7.31 in order to find the new intermediate state

$(h'_b, \Phi'_b, S'_b, \mathbb{P}'_b)$ that allows to swap the operations. With this information, we use Lemma 7.34 to find a resulting trace τ'_c which contains the swap and for which $\mathsf{tgen}(\tau'_c, h, h', \emptyset, S, \mathbb{P}_1 \| \mathbb{P}_2)$ holds. We repeat the stated process until no more swap is possible. We obtain a final trace τ_{seq}, for which, from (28) and the fact that the original trace τ_c is clean, we can state the following:

$$\mathsf{tgen}(\tau_{seq}, h, h', \emptyset, S, \mathbb{P}_1 \| \mathbb{P}_2) \tag{32}$$

$$\wedge\ \mathsf{clean}(\tau_{seq}) \tag{33}$$

We now claim that all of ι's operations appear in τ_{seq} before the actions performed by any other transaction j in N. Let's instead assume that there is a transaction j which has an action in τ_{seq} happening before one done by ι, formally:

$$\exists x, y, n, n', j.\, \iota \neq j \wedge x = (\alpha(j), n) \wedge y = (\alpha(\iota), n') \wedge \tau_{seq} \vDash x < y \tag{34}$$

From (34) we know there must exist actions α and α' which label a sequence of consecutive transitions of the shape described in (31) and are performed by transactions j and ι respectively, for $j \neq \iota$. We proceed by analyzing the only feasible case of assignment to α and α' that was not considered as part of Lemma 7.31 and show that we end up in a contradiction.
If $\alpha = \mathsf{unlock}(j, k)$ and $\alpha' = \mathsf{lock}(\iota, k, \kappa)$ and $\Phi_a(k) = (\{j\}, \mathrm{X})$, then from (33) we know that τ_{seq} does not contain any spurious locks which implies that ι is obtaining a lock on k in order to later read or write to it. Given that j is releasing a lock on k which was held in exclusive mode, it means that it wrote to it beforehand through some action $\alpha_c = \mathsf{write}(j, k, v)$ (since again $\mathsf{clean}(\tau_{seq})$ holds from (33) and therefore no $\mathsf{redundant}$ lock is held). We proceed with two cases:

1. If $\kappa = \mathrm{X}$ then from (33) we know that $\mathsf{clean}(\tau_{seq})$ holds and as a consequence no $\mathsf{redundant}$ locks are in τ_{seq} meaning that ι later writes to k through an action $\alpha_w = \mathsf{write}(\iota, k, v')$. From the definition of $\mathsf{conflict}$ it follows that the actions α_c and α_w must be conflicting. From the definition of $\mathsf{SG}(\tau_{\mathsf{seq}})$ and the fact that the strict total order $\sqsubset$ keeps the serialization graph's edges, it must be the case that $j \sqsubset \iota$ which is not possible due to the fact that ι is the minimal element of the $\sqsubset$ relation from (29) and we get a contradiction.

2. If $\kappa = \mathrm{S}$ then ι later only reads storage cell k through an action $\alpha_r = \mathsf{read}(\iota, k, v')$. In this situation we are back to the previous case (a) as we would have a conflict between α_c and α_r. This case also ends with a contradiction.

By contradiction we can state that the negation of (34) must hold and therefore that the minimal transaction ι's operations appear in τ_{seq} before the ones of any other transaction. Let's now analyse the structure of the reduction described by τ_{seq}, for some fresh $\alpha \in \mathsf{Act}$.

$$(h, \emptyset, S, \mathbb{P}_1 \| \mathbb{P}_2) \xrightarrow{\alpha} \underbrace{(h'', \Phi'', S'', \mathbb{P}'') \rightarrow^* (h', \emptyset, S', \mathtt{skip})}_{n \text{ steps}}$$

- If $\alpha = \mathsf{sys}$ then by Lemma C.9 we obtain that $(h, \mathbb{P}_1 \| \mathbb{P}_2) \rightarrow_{\text{ATOM}} (h'', \mathbb{P}'')$ and the final result follows by I.H.

- If $\alpha \neq \mathsf{sys}$ then from (34) we know that the action was performed by transaction ι, the minimal one according to $\sqsubset$. Without loss of generality, we can assume that the program $\mathbb{P}_1 \| \mathbb{P}_2$ is of the following shape:

$$\left(\mathbb{T}_\iota; \mathbb{P}'_1\right) \| \mathbb{P}_2 \tag{35}$$

  From the assumption that $\alpha \neq \mathsf{sys}$ and Lemma C.10 we know that all of the labels generated from the reduction ι, will appear before any system transition. This means

that under τ_{seq} we are able to reduce the initial state and program as follows, for some $m < n+1$:

$$(h, \emptyset, S, (\mathbb{T}_\iota; \mathbb{P}'_1) \| \mathbb{P}_2) \rightarrow^m (h_{fin}, \Phi_{fin}, S_{fin}, (\texttt{skip}; \mathbb{P}'_1) \| \mathbb{P}_2) \tag{36}$$

From (36), *Base case 2* of this proof and the fact that by rule AtTrans a transaction can always run without conditions on the global storage (i.e. empty premiss), we obtain that:

$$(h, (\mathbb{T}_\iota; \mathbb{P}'_1) \| \mathbb{P}_2) \rightarrow_{\text{Atom}} (h_{fin}, (\texttt{skip}; \mathbb{P}'_1) \| \mathbb{P}_2) \tag{37}$$

From (36), (37) and I.H given that we have reduced the starting program for m steps, we know that:

$$(h_{fin}, \Phi_{fin}, S_{fin}, (\texttt{skip}; \mathbb{P}'_1) \| \mathbb{P}_2) \rightarrow^* (h', \emptyset, S', \texttt{skip}) \tag{38}$$

$$\implies (h_{fin}, (\texttt{skip}; \mathbb{P}'_1) \| \mathbb{P}_2) \rightarrow^*_{\text{Atom}} (h', \texttt{skip}) \tag{39}$$

which concludes our proof.

□

7.3 Soundness

All the ingredients necessary to show the soundness of the mCAP logic with respect to the 2PL semantics have been illustrated and proven. We are now left to combine all of the results into a proof of soundness, which follows from the fact that mCAP is sound with respect to the Atom semantics, as determined in Theorem 4.52, and every terminating reduction in 2PL can be replicated in Atom, by Theorem 7.41.

We first define the meaning of a 2PL semantic judgement, and later prove that any syntactic program judgement in mCAP is sound with respect to the 2PL semantics.

Definition 7.42. (2PL Semantic Judgement).

$$\vDash_{\text{2PL}} \{P\}\ \mathbb{P}\ \{Q\}$$
$$\iff$$
$$\left(\begin{array}{c} \forall e, \delta, w, \sigma, h.\, w \in [\![P]\!]_{e,\delta} \wedge \sigma \in \lfloor w \rfloor_W \wedge h = \sigma \downarrow_1 \wedge (h, \emptyset, \emptyset, \mathbb{P}) \rightarrow^* (h', \emptyset, -, \texttt{skip}) \\ \implies \exists w', \sigma'.\, w' \in [\![Q]\!]_{e,\delta} \wedge \sigma' \in \lfloor w' \rfloor_W \wedge \sigma' \downarrow_1 = h' \end{array} \right)$$

The meaning of such judgement is that for any world that satisfies P, we take the heap component from its reification, and run it through the 2PL semantics until termination. The final storage, h', will then need to be one of the possible reifications of a terminating world w' which satisfies assertion Q.

Theorem 7.43. (2PL Soundness). For all $\mathbb{P}, P, Q$ if $\vdash \{P\}\ \mathbb{P}\ \{Q\}$ then $\vDash_{\text{2PL}} \{P\}\ \mathbb{P}\ \{Q\}$.

Proof. Let's pick arbitrary $\mathbb{P} \in \mathsf{Prog}$ and $P, Q \in \mathsf{Assn}$. We now assume that $\vdash \{P\}\ \mathbb{P}\ \{Q\}$ holds. It is required to show that:

$$\forall e, \delta, w, \sigma, h. \tag{40}$$

$$w \in [\![P]\!]_{e,\delta} \wedge \sigma \in \lfloor w \rfloor_W \wedge h = \sigma \downarrow_1 \wedge (h, \emptyset, \emptyset, \mathbb{P}) \rightarrow^* (h', \emptyset, -, \texttt{skip}) \tag{41}$$

$$\implies \exists w', \sigma'.\, w' \in [\![Q]\!]_{e,\delta} \wedge \sigma' \in \lfloor w' \rfloor_W \wedge \sigma' \downarrow_1 = h' \tag{42}$$

We pick an arbitrary $w \in \mathsf{World}, e \in \mathsf{LEnv}, \delta \in \mathsf{PEnv}, \sigma \in (\mathsf{Storage} \times \mathsf{Stack}), h \in \mathsf{Storage}$ and assume that:

$$w \in [\![P]\!]_{e,\delta} \wedge \sigma \in \lfloor w \rfloor_W \tag{43}$$

$$\wedge\ h = \sigma \downarrow_1 \wedge (h, \emptyset, \emptyset, \mathbb{P}) \rightarrow^* (h', \emptyset, -, \texttt{skip}) \tag{44}$$

From (44) and Theorem 7.41 we know that the following holds:

$$(h, \mathbb{P}) \rightarrow^*_{\text{ATOM}} (h', \texttt{skip}) \tag{45}$$

From (45) and Theorem 7.21, i.e. the one-way equivalence between ATOM and the Views operational semantics, we obtain:

$$\exists \sigma'.\, h' = \sigma' \downarrow_1 \wedge\ \mathbb{P}, \sigma \rightarrow^*_{\mathsf{Views}} \texttt{skip}, \sigma' \tag{46}$$

From (43), Theorem 4.52, i.e. soundness of mCAP's transactions instantiation, and (46) we can conclude that, for some $w' \in \mathsf{World}$:

$$w' \in [\![Q]\!]_{e,\delta} \wedge \sigma' \in \lfloor w' \rfloor_W \wedge \sigma' \downarrow_1 = h'$$

$\square$

8. Evaluation & Comparisons

The project was laid out as an exploration of transactional reasoning, with the objective of focusing on locking protocols, in particular the two-phase locking one. The main goal was to understand the protocol's behaviour from an abstract point of view, formally define it and later find a way to allow client reasoning around systems adopting it. These requirements were met and extended, with some limitations.

Overall, the program logic for serializable transactions we defined is the first kind of effort in the area and a more general result than was initially required. In terms of its applications, a core strength of the framework that was built, is the flexibility of its 2PL model which is able to describe a large set of two-phase locking instantiations. Together with the operational semantics, it is also generic enough to allow its replicated usage in multiple settings, from databases to transactional applications that manage concurrency through two-phase locking. The fact that we prove serialisability on all traces produced by the semantics, and equivalence to the ATOMIC semantics, provides a great starting point for extensions or customisations to fit a particular context or need.

The primary limitation of the work is the lock manager abstraction, which appears as part of the 2PL model and semantics. First of all, it is used as a primitive structure and, together with the storage and the transactions' stack, it is embedded in the model. This might sound counterintuitive, since these kind of components are not usually built-in, but are rather part of user programs. We shape and use the lock manager as an *oracle*, by hiding away all of the effective complexities involved in the construction and management of this structure, together with its interface to client transactions. This choice constraints situations where lock managing for cells cannot be directly abstracted by the paradigm we chose. Still, it is designed in a reasonable way by taking inspiration from what literature suggests [3].

A substantial contribution of the project is that it served as a starting effort to build a bridge between separation logic style program reasoning, done in a concurrent setting, and database theory. Results in the latter group, are mainly conceived through the study of traces of executions and the analysis of graphs that arise from their processing. We combine both approaches and use database style reasoning, in order to build a proof strategy that allows us to show the soundness of a program logic. The latter enables to build proofs of concurrent programs adopting 2PL for concurrency control.

The logic is a particular instantiation of mCAP, whose model is itself formalised as part of this thesis. It inherits the characteristics of standard CAP [12], while amending its main shortcomings and constraints. The novelties introduced, as previously mentioned, are concerned with the parameterization of both the machine states and the capabilities model. These are in fact provided through any partial commutative monoid with multiple units, as defined in Section 4.1. It follows that such abstraction frees the framework user from being contstrained to the standard heap model or the fractional-permission capabilities [4] one, which cannot be modified in the original CAP work. As a consequence, the choice of the two parameters can then be taylored to the particular problem, algorithm or application that needs to be verified. This clearly eases the proof process and empowers users with more flexibility. We also vastly modified the original definitions of the *rely* and *guarantee* relations between worlds. They were

in fact changed in order to support not just one, but multiple shared region updates done by both the environment and the running thread in one step. This results in a more realistic model that is able to express behaviours which are common in real-world scenarios and fundamental for transactional reasoning, where we must treat a whole block of code as atomic, even with potentially many updates. Finally, we believe that the structure and definition of the model is very accessible to people relatively new to the field, and, given that the program logic is an instantiation of the Views framework [11], its proof of soundness is clearer and shorter.

During the early stages of the project, a considerable amount of time was used to investigate implementations of 2PL [17], together with the parallel effort of getting up to speed with modern ways to reason about concurrency inside heap-manipulating programs. The latter took longer than expected and, given the timescale of the project, could have been optimized to only focus on very relevant material. The initial thought was to start by proving a concrete implementation of the protocol through the TaDA logic [9] and potentially extend it to provide better support for transactions. A full C implementation was written and gradually proven through a WHILE language equivalent version. Nevertheless, after some discoveries, the theoretical results got very interesting, which implied that it was decided to take a step back from the implementation, in order to shift the focus to more abstract ideas.

The effort was then moved to the construction and improvement of the model and operational semantics we introduced as part of this work. The latter can be considered as one of the first attempts at formalising 2PL from a program logic point of view. Being able to construct concrete proofs, in the style of separation logic of programs, running under a 2PL concurrency control is also a novelty in many ways. Still, there has been interesting research work in the same area which is comparable to what was achieved here.

The model developed as part of the *Push/Pull* framework [16], and presented in Section 2.3, is more abstract than the one we describe in this thesis, as the main goal there was to cover a range of serializability protocols, not only 2PL, and find common ground among them. As a consequence, and given our focus on two-phase locking, we are able to describe much more precisely the behaviour and properties of 2PL in particular, while lacking the ability, for now, to reason about other serializability-preserving mechanisms. This is instead done by the compared model, even if mapping a given program or algorithm to the set of rules in the semantics might not be straightforward.

When building a proof of serialisability for the target semantics, a similar strategy is adopted between *Push/Pull* and the work presented here. This is because both approaches first define an atomic operational semantics, followed by a proof of one-way equivalence to determine a simulation. This step additionally enables us to link the 2PL semantics to the mCAP logic judgements and use our semantics within the context of a full-fledged concurrent program logic. On the other hand, building a program logic on top of the *Push/Pull* model would require a considerable effort.

On the contrary, the *Push/Pull*'s framework models a variety of input languages, as it assumes a generic set of method call names, M, together with features also included in our model, such as the ability to spawn a new thread, make local stack updates, start a transaction and `skip`. M's elements can be anything, as they represent the operations performed by transactions that are used to populate the local and global logs. As part of our work, the operations that can happen within a transaction are instead bound to the programming language we defined in Section 4.5, which nevertheless contains C-style elementary commands that can be used to formulate much more complicated programs.

In terms of the operational semantics for transactions we defined in this report, we find the comparison to the work in [18] to be very relevant. Both approaches consider the adoption of small-step semantics in order to model the interleaving between concurrent programs, and use a similar atomic baseline to show properties about the more complicated semantics by means of equivalence proofs. If on one hand, under the 2PL operational semantics, we allow

multiple transactions to execute concurrently with effective interleaving between them, on the other, the formal languages presented in [18], only permit one thread at a time to execute a transaction. Still, in the weaker isolation languages of the AtomsFamily, nontransactional code commands can run in parallel and therefore be interleaved with the running transaction. Each of the languages introduced, comes with a type system which enforces specific behaviours and conservatively prohibits reaching error states from the a syntax misuse. As part of our work, we do not need such a construct, since we reason about terminating programs which are, by definition, syntactically and semantically correct. This is because our semantics would get stuck, i.e. not be able to reduce further, if for example a transaction tries to read an item whose key is not in the domain of the storage. It follows that we can reason directly on the structure of programs. On the other hand, in the compared work, properties are proven under the constraints of a particular type system: for example, the Weak and StrongBasic languages, introduced in Section 2.3, are shown to be indistinguishable under a type system which imposes that the same heap location cannot be accessed inside and outside a transaction.

9. Conclusions

In this project we succesfully introduced a program logic for serializable transactions and its concrete application to the two-phase locking protocol. As part of the process, we started by taking CAP, a program logic for concurrency, and upgrading it to mCAP, by freeing it from specific constraints. In fact, the mCAP model was reformulated from scratch, together with parts of its assertion language and programming language, by adding the syntactic constructs necessary to support transactions. The program logic introduced was then further instantiated for our transactional model for serializability, and proven to be sound with respect to truly atomic semantics, ones where transactions are really treated as atomic blocks of code, that run in full isolation with no interleaving.

When applying the framework to the specific case of two-phase locking, we provided a formalisation of software systems adopting the protocol for concurrency control of transactions. This is given in terms of a model and corresponding operational semantics, that enable us to construct a framework where interesting properties can be proven: first of all the *serializability* of traces, followed by the equivalence between the operational semantics and the atomic ones we already mentioned. This last result was key in linking the effort to the part of the work related to the program logic, as it allowed to prove mCAP's soundness in terms of the 2PL small-step operational semantics.

In conclusion, we empower client reasoning around serializability in general and in the specific case of two-phase locking. In this case, a sound abstraction is established, one that frees users from the burden of considering the intricate details of the locking protocol, while shifting the focus to the more natural approach of treating transactions' executions as serial. Within the vast world of transactions, seen as programming blocks in databases, we took a first step in the approach of reasoning about serializable ones through a program logic, and intend to take this effort ahead in the future.

9.1 Future Work

The results obtained as part of this project in the context of serializable transactions, together with the approach taken in building an overall framework to reason about concurrency in a 2PL setting, open up the possibility of both tayloring its use to a particular application or extending the work to wider scenarios. We explore the latter by looking at how this work can be expanded or improved.

- Extend the focus of the framework to non-serializable or weak models of transactional concurrency. Nowadays, specially in the context of distributed systems, full serializability appears to be too expensive in terms of performance as the explicit synchronization of single items is a bottleneck. This is the reason why weaker consistency and isolation properties, usually enforced by optimistic concurrency control protocols, are preferred. It would therefore be extremely interesting to investigate how the work done here can be modified and ported to these different levels of consistency. One part of the research effort in the Program Specification and Verification Group is to look at this scenario. The focus

would be on the analysis of the snapshot isolation guarantee, as opposed to serializability, and the goal is to take a similar path as compared to the one in this thesis, and define a formal model, operational semantics and a program logic on top of it.

- Model other applications in addition to 2PL. These will be concurrency control protocols which are used in real-world database systems to guarantee the serializability of transactions' executions. Moreover, during the process of expanding the applications to other lock-based approaches, we could generalize the effort and keep the parts of our model that do not directly refer to 2PL, while parameterizing all the rest. Instantiations could then be built by providing the required well formed constructs that are able to express the behaviour of the particular protocol. In terms of operational semantics, we would need to find a set of rules that, while abstracting large parts of the particular scenario, still allow to prove serializability and atomic equivalence of instantiations.

- As mentioned in Section 8, the initial goal was to prove correctness of a fine-grained implementation through TaDA [9]. Theoretical results got more interesting than the implementation itself, and the focus was accordingly shifted. At this point, if the timescale of the project was larger, the next step would have been to expand on the original idea and to now prove a simulation between a concrete implementation of a system and our operational semantics. The mentioned system would be required to adopt a particular flavour of two-phase locking to manage concurrent access to data items. An implementation has already been written in the C language and, given the premature effort to prove its equivalence, is not included in this report.

- Take inspiration from the work done in TaDA [9] in terms of abstract atomicity, in order to investigate the formal meaning of combining multiple atomic transactions in a larger and *fictionally-atomic* one. In relation to this, it would also be interesting to further explore the inverse situation, where we divide an existing transaction into multiple ones, without affecting the overall result of executions. The latter technique is referred to in research as *transaction chopping* and understanding how it applies to our model and semantics would give us the possibility of establishing more powerful properties.

Bibliography

[1] American National Standards Institute. *American national standard for information systems: database language — SQL: ANSI X3.135-1992*. November 1992.

[2] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[4] J. Boyland. Checking interference with fractional permissions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[5] S. Brookes. A semantics for concurrent separation logic. *Theor. Comput. Sci.*, 375(1-3):227–270, Apr. 2007.

[6] C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proceedings of the 22Nd Annual IEEE Symposium on Logic in Computer Science*, LICS '07, pages 366–378, Washington, DC, USA, 2007. IEEE Computer Society.

[7] A. Cerone, G. Bernardi, and A. Gotsman. A Framework for Transactional Consistency Models with Atomic Visibility. In L. Aceto and D. de Frutos Escrig, editors, *26th International Conference on Concurrency Theory (CONCUR 2015)*, volume 42 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 58–71, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[8] A. Cerone, A. Gotsman, and H. Yang. Algebraic laws for weak consistency. *CoRR*, abs/1702.06028, 2017.

[9] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. TaDA: A logic for time and data abstraction. In *ECOOP*, pages 207–231, 2014.

[10] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. Steps in Modular Specifications for Concurrent Modules (Invited Tutorial Paper). In *Proceedings of the 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS)*, pages 3–18, June 2015.

[11] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional reasoning for concurrent programs. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 287–300, New York, NY, USA, 2013. ACM.

[12] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 504–528, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[14] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[15] C. B. Jones. Specification and design of (parallel) programs. In *IFIP congress*, volume 83, pages 321–332, 1983.

[16] E. Koskinen and M. Parkinson. The push/pull model of transactions. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 186–195, New York, NY, USA, 2015. ACM.

[17] P. Lakhina and N. Shay. An implementation of two phase locking (2pl) and optimistic concurrency control (occ) on google app engine. Technical report, University of California, Santa Barbara, 2015.

[18] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 51–62, New York, NY, USA, 2008. ACM.

[19] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, Apr. 2007.

[20] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6(4):319–340, Dec. 1976.

[21] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, Oct. 1979.

[22] A. Raad. *Abstraction, Refinement and Concurrent Reasoning.* Phd thesis, Imperial College London, 2017.

[23] A. Raad, J. Villard, and P. Gardner. CoLoSL: Concurrent Local Subjective Logic. In *Proceedings of the 24th European Symposium on Programming (ESOP)*, pages 710–735, 2015.

[24] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[25] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts.* McGraw-Hill, Inc., New York, NY, USA, 5 edition, 2006.

[26] V. Vafeiadis. Modular fine-grained concurrency verification. Technical Report UCAM-CL-TR-726, University of Cambridge, Computer Laboratory, July 2008.

Appendices

A. Auxiliary Lemmata for mCAP

Theorem A.1. (Command soundness). For all $\mathbb{C} \in \mathsf{Cmd}$, their corresponding axiom $(M_1, \mathbb{C}, M_2) \in \mathsf{Ax_C}$ and any given machine state $m \in \mathbb{M}$ the following must hold.

$$[\![\mathbb{C}]\!]_{\mathsf{C}} \left(\lfloor M_1 \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}\right) \subseteq \lfloor M_2 \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}$$

Proof. By coinduction on the structure of $\mathbb{C}$.

Base case 1: $\hat{\mathbb{C}}$
To show: $[\![\hat{\mathbb{C}}]\!]_{\mathsf{C}} \left(\lfloor M_1 \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}\right) \subseteq \lfloor M_2 \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}$
The result follows directly from Parameter 4.48.

Base case 2: `skip`
To show: $[\![\texttt{skip}]\!]_{\mathsf{C}} \left(\lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}\right) \subseteq \lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}$
By definition of $[\![\texttt{skip}]\!]_{\mathsf{C}}$ we have the following.

$$\begin{aligned} [\![\texttt{skip}]\!]_{\mathsf{C}} \left(\lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}\right) &= \lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}} \\ &\subseteq \lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}} \end{aligned}$$

Coinductive case 1: $\mathbb{C}_1; \mathbb{C}_2$
To show:

$$\begin{gathered} [\![\mathbb{C}_1; \mathbb{C}_2]\!]_{\mathsf{C}} \left(\lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}\right) \subseteq \lfloor M' \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}} \\ \text{where } (M, \mathbb{C}_1, M''), (M'', \mathbb{C}_2, M') \in \mathrm{Ax}_{\mathsf{C}} \end{gathered}$$

Coinductive hypothesis: Assume the property holds for $\mathbb{C}_1$ and for $\mathbb{C}_2$.
By definition of $[\![\mathbb{C}_1; \mathbb{C}_2]\!]_{\mathsf{C}}$ we have the following.

$$\begin{aligned} [\![\mathbb{C}_1; \mathbb{C}_2]\!]_{\mathsf{C}} \left(\lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}\right) &= [\![\mathbb{C}_2]\!] \left([\![\mathbb{C}_1]\!]_{\mathsf{C}} \left(\lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}\right)\right) \\ \text{by C.H. on } \mathbb{C}_1 &\subseteq [\![\mathbb{C}_1]\!]_{\mathsf{C}} \left(\lfloor M'' \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}\right) \\ \text{by C.H. on } \mathbb{C}_2 &\subseteq \lfloor M' \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}} \end{aligned}$$

Coinductive case 2: $\texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2$
To show:

$$\begin{gathered} [\![\texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2]\!]_{\mathsf{C}} \left(\lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}\right) \subseteq \lfloor M' \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}} \\ \text{where } (M, \mathbb{C}_1, M_1), (M, \mathbb{C}_2, M_2) \in \mathrm{Ax}_{\mathsf{C}} \text{ and } M' \equiv M_1 \cup M_2 \end{gathered}$$

Coinductive hypothesis: Assume the property holds for $\mathbb{C}_1$ and for $\mathbb{C}_2$.
Let $S_{in} = \lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}$ and $S_{out} = \lfloor M' \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}$. Let's pick an arbitrary $\sigma \in S_{in}$. It is now sufficient to show that the following holds:

$$[\![\texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2]\!]_{\mathsf{C}} (\sigma) \subseteq S_{out}$$

From the definition of $[\![\texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2]\!]_{\mathsf{C}}$ we have the following.

$$[\![\texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2]\!]_{\mathsf{C}}\,(\sigma) = \textbf{if}\ [\![\mathbb{B}]\!]^{\mathrm{B}}_{\sigma}\ \textbf{then}\ [\![\mathbb{C}_1]\!]_{\mathsf{C}}\,(\sigma)\,\textbf{else}\ [\![\mathbb{C}_2]\!]_{\mathsf{C}}\,(\sigma)$$

We now have two scenarios to consider, based on how $[\![\mathbb{B}]\!]^{\mathrm{B}}_{\sigma}$ evaluates.

- If $[\![\mathbb{B}]\!]^{\mathrm{B}}_{\sigma} = \top$, then we can proceed as follows.

$$\begin{aligned}&\textbf{if}\ \top\ \textbf{then}\ [\![\mathbb{C}_1]\!]_{\mathsf{C}}\,(\sigma)\,\textbf{else}\ [\![\mathbb{C}_2]\!]_{\mathsf{C}}\,(\sigma)\\ &= [\![\mathbb{C}_1]\!]_{\mathsf{C}}\,(\sigma)\\ &\subseteq \lfloor M_1 \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}} \text{ by C.H. on } \mathbb{C}_1\\ &\subseteq \lfloor M' \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}} \text{ as } M_1 \subseteq M'\end{aligned}$$

- If $[\![\mathbb{B}]\!]^{\mathrm{B}}_{\sigma} = \bot$, then we can proceed as follows.

$$\begin{aligned}&\textbf{if}\ \bot\ \textbf{then}\ [\![\mathbb{C}_1]\!]_{\mathsf{C}}\,(\sigma)\,\textbf{else}\ [\![\mathbb{C}_2]\!]_{\mathsf{C}}\,(\sigma)\\ &= [\![\mathbb{C}_2]\!]_{\mathsf{C}}\,(\sigma)\\ &\subseteq \lfloor M_2 \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}} \text{ by C.H. on } \mathbb{C}_2\\ &\subseteq \lfloor M' \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}} \text{ as } M_2 \subseteq M'\end{aligned}$$

Coinductive case 3: $\texttt{while}\,(\mathbb{B})\ \mathbb{C}$
To show:

$$\begin{gathered}[\![\texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}\,(\lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}) \subseteq \lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}\\ \text{where } \forall m \in M.\, [\![B]\!]^{\mathrm{B}}_{m} = \bot \text{ or}\\ \exists M'.\, M' \subseteq M \land \forall m \in M'.\, [\![B]\!]^{\mathrm{B}}_{m} = \top \land (M', \mathbb{C}, M) \in \mathrm{Ax}_{\mathsf{C}}\end{gathered}$$

Coinductive hypothesis: Assume that $(M', \mathbb{C}, M)$.
Let's pick an arbitrary $\sigma \in \lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}}$. It is now sufficient to show that the following holds:

$$[\![\texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}\,(\sigma) \subseteq \lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}} \tag{47}$$

By definition of $[\![\texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}$ we have the following.

$$\begin{aligned}[\![\texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}\,(\sigma) &= [\![\texttt{if}\,(\mathbb{B})\ \ (\mathbb{C};\texttt{while}\,(\mathbb{B})\ \mathbb{C})\ \ \texttt{else skip}]\!]_{\mathsf{C}}\,(\sigma)\\ &= \textbf{if}\ [\![\mathbb{B}]\!]^{\mathrm{B}}_{\sigma}\ \textbf{then}\ [\![\mathbb{C};\texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}\,(\sigma)\ \ \textbf{else}\ [\![\texttt{skip}]\!]_{\mathsf{C}}\,(\sigma)\end{aligned}$$

As before, we have two cases to consider, based on how $[\![\mathbb{B}]\!]^{\mathrm{B}}_{\sigma}$ evaluates.

- If $[\![\mathbb{B}]\!]^{\mathrm{B}}_{\sigma} = \bot$, then we can proceed as follows.

$$\begin{aligned}&\textbf{if}\ \bot\ \textbf{then}\ [\![\mathbb{C};\texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}\,(\sigma)\ \ \textbf{else}\ [\![\texttt{skip}]\!]_{\mathsf{C}}\,(\sigma)\\ &= [\![\texttt{skip}]\!]_{\mathsf{C}}\,(\sigma)\\ &= \{\sigma\} \text{ by definition}\\ &\subseteq \lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}} \text{ by original assumption}\end{aligned}$$

- If $[\![\mathbb{B}]\!]^{\mathrm{B}}_{\sigma} = \top$, then $\sigma \in M'$ and we can proceed as follows.

$$\begin{aligned}&\textbf{if}\ \top\ \textbf{then}\ [\![\mathbb{C};\texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}\,(\sigma)\ \ \textbf{else}\ [\![\texttt{skip}]\!]_{\mathsf{C}}\,(\sigma)\\ &= [\![\mathbb{C};\texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}\,(\sigma)\\ &= [\![\texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}\,([\![\mathbb{C}]\!]_{\mathsf{C}}\,(\sigma))\\ &\subseteq [\![\texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}\,(\lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}})\ \text{ by C.H.}\\ &\subseteq \lfloor M \bullet_{\mathbb{M}} \{m\} \rfloor_{\mathbb{M}} \text{ by coinduction}\end{aligned}$$

□

Lemma A.2. For all $w_1, w_2, w, w' = (l', g', \mathcal{J}') \in \mathsf{World}$,

$$w_1 \bullet w_2 = w \wedge (l', g', \mathcal{J}') \in G(w_1) \implies ((w_2)_\mathsf{L}, g', \mathcal{J}') \in R(w_2)$$

Proof. Let's pick arbitrary $w_1, w_2, w, w' = (l', g', \mathcal{J}') \in \mathsf{World}$ such that:

$$w_1 \bullet w_2 = w \wedge \tag{48}$$
$$(l', g', \mathcal{J}') \in G(w_1) \tag{49}$$

To show: $((w_2)_\mathsf{L}, g', \mathcal{J}') \in R(w_2)$
From (49) and the definition of G we know that:

$$(l', g', \mathcal{J}') \in (G^c \cup G^u)^*(w_1) \tag{50}$$

From (48), (50) and by Lemma A.3, Lemma A.4 we get:

$$((w_2)_\mathsf{L}, g', \mathcal{J}') \in (R^c \cup R^u)^*(w_2)$$

Therefore we can conclude the following.

$$((w_2)_\mathsf{L}, g', \mathcal{J}') \in R(w_2)$$

□

Lemma A.3. For all $w_1, w_2, w, w' = (l', g', \mathcal{J}') \in \mathsf{World}$,

$$w_1 \bullet w_2 = w \wedge (l', g', \mathcal{J}') \in G^u(w_1) \implies ((w_2)_\mathsf{L}, g', \mathcal{J}') \in R^u(w_2)$$

Proof. Let's pick arbitrary $w_1 = (l_1, g_1, \mathcal{J}_1), w_2 = (l_2, g_2, \mathcal{J}_2), w$ and $(l', g', \mathcal{J}') \in \mathsf{World}$, such that:

$$w_1 \bullet w_2 = w \wedge \tag{51}$$
$$(l', g', \mathcal{J}') \in G^u(w_1) \tag{52}$$

To show: $((w_2)_\mathsf{L}, g', \mathcal{J}') \in R^u(w_2)$
From (51) we know that

$$g_1 = g_2 \tag{53}$$
$$\mathcal{J}_1 = \mathcal{J}_2 \tag{54}$$

From the definition of G^u and from (52) and (54) we know that:

$$\mathcal{J}' = \mathcal{J}_1 = \mathcal{J}_2 \wedge \tag{55}$$
$$((l_1 \oplus g_1)_\mathsf{K})^\perp_\mathbb{K} = ((l' \oplus g')_\mathsf{K})^\perp_\mathbb{K} \wedge \tag{56}$$
$$(g_1 = g' \vee (\exists r, \kappa \leq (l_1)_\mathsf{K}(r).\, (g_1, g') \in \lceil \mathcal{J}_1(r) \rceil(\kappa) \tag{57}$$
$$\wedge ((l_1 \oplus g_1)_\mathsf{M})^\perp_\mathbb{M} = ((l' \oplus g')_\mathsf{M})^\perp_\mathbb{M}) \tag{58}$$

Given the disjunction in (57), we need to consider two cases.

Case 1: $g_1 = g'$
From (53) and our assumption, we get that $g_2 = g'$. Now, from (55), it follows that:

$$((w_2)_\mathsf{L}, g', \mathcal{J}') = ((l_2, g_2, \mathcal{J}_2)) \tag{59}$$

From (59) and the definition of R^u we can conclude that:

$$((w_2)_{\mathsf{L}}, g', \mathcal{J}') \in R^u((l_2, g_2, \mathcal{J}_2))$$

Case 2:

$$\exists r, \kappa \leq (l_1)_{\mathsf{K}}(r).\, (g_1, g') \in \lceil \mathcal{J}_1(r) \rceil(\kappa) \tag{60}$$
$$\wedge((l_1 \oplus g_1)_{\mathsf{M}})^{\perp}_{\mathbb{M}} = ((l' \oplus g')_{\mathsf{M}})^{\perp}_{\mathbb{M}} \tag{61}$$

From (51), (53) and (54) we know that:

$$w = (l_1 \circ l_2, g_2, \mathcal{J}_2) \tag{62}$$

From the definition of World we know that $\mathsf{wf}(w)$ holds, and together with (53) we have:

$$((l_1 \circ l_2) \oplus g_2)_{\mathsf{K}} = (l_1)_{\mathsf{K}} \bullet_{\mathbb{K}} (l_2)_{\mathsf{K}} \bullet_{\mathbb{K}} \llbracket g_2 \rrbracket_{\mathsf{K}} = (l_1 \oplus g_1)_{\mathsf{K}} \bullet_{\mathbb{K}} (l_2)_{\mathsf{K}} \text{ defined} \tag{63}$$
$$\text{and } ((l_1 \circ l_2) \oplus g_1)_{\mathsf{M}} = (l_1 \oplus g_1)_{\mathsf{M}} \bullet_{\mathbb{M}} (l_2)_{\mathsf{M}} \text{ defined} \tag{64}$$

From (60) we obtain $\kappa \leq (l_1)_{\mathsf{K}}(r)$, from (63) and Lemma A.5, we know:

$$\kappa \,\sharp\, ((l_2)_{\mathsf{K}} \bullet_{\mathbb{K}} \llbracket g_2 \rrbracket_{\mathsf{K}}) \tag{65}$$

From (53), (61) and (64) we know:

$$(l' \oplus g')_{\mathsf{M}} \bullet_{\mathbb{M}} (l_2)_{\mathsf{M}} = ((l' \circ l_2) \oplus g')_{\mathsf{M}} \text{ defined} \tag{66}$$

From (56) and (63) we know:

$$(l' \oplus g')_{\mathsf{K}} \bullet_{\mathbb{K}} (l_2)_{\mathsf{K}} = ((l' \circ l_2) \oplus g')_{\mathsf{K}} \text{ defined} \tag{67}$$

From (66), (67) we get that $(l'_1 \circ l_2) \oplus g'$ is defined and as a consequence we obtain:

$$l_2 \oplus g' \text{ defined} \tag{68}$$

From (55), (60), (65), (68) and the definition of R^u we can conclude that:

$$((w_2)_{\mathsf{L}}, g', \mathcal{J}') \in R^u((l_2, g_2, \mathcal{J}_2))$$

□

Lemma A.4. For all $w_1, w_2, w, w' = (l', g', \mathcal{J}') \in \mathsf{World}$,

$$w_1 \bullet w_2 = w \wedge (l', g', \mathcal{J}') \in G^c(w_1) \implies ((w_2)_{\mathsf{L}}, g', \mathcal{J}') \in R^c(w_2)$$

Proof. Let's pick arbitrary $w_1 = (l_1, g_1, \mathcal{J}_1), w_2 = (l_2, g_2, \mathcal{J}_2), w$ and $w' = (l', g', \mathcal{J}') \in \mathsf{World}$, such that:

$$w_1 \bullet w_2 = w \wedge \tag{69}$$
$$(l', g', \mathcal{J}') \in G^c(w_1) \tag{70}$$

To show: $((w_2)_{\mathsf{L}}, g', \mathcal{J}') \in R^c(w_2)$
From (51) we know that

$$g_1 = g_2 \tag{71}$$
$$\mathcal{J}_1 = \mathcal{J}_2 \tag{72}$$

From the definition of G^c and from (70), (71) and (72) we know that:

$$\exists r, m, l, l_r, a, \rho.\, \mathsf{fresh}(r, w) \wedge \mathsf{dom}(\rho) = \{r\} \wedge \tag{73}$$
$$l_1 = l \circ l_r \wedge l' = l \circ (m, \rho) \wedge m \in \mathbf{0}_{\mathbb{M}} \wedge \tag{74}$$
$$g' = g_2[r \mapsto l_r] \wedge \mathcal{J}' = \mathcal{J}_2[r \mapsto a] \tag{75}$$

From (69) and (74) we know that $(l_1 \circ l_2) \oplus g$ and that $l_1 = l \circ l_r$ are defined which implies that:

$$l_r \circ l_2 \text{ defined} \tag{76}$$
$$l_2 \oplus g_2 \text{ defined} \tag{77}$$

Now, from (75) we know that $g' = g_2[r \mapsto l_r]$ and together with (76) and (77) we obtain that:

$$l_2 \oplus g' \text{ defined} \tag{78}$$

From (69) and (73) we obtain that:

$$\mathsf{fresh}(r, w_2) \tag{79}$$

From (75), (78) and (79) we can conclude that:

$$((w_2)_{\mathsf{L}}, g', \mathcal{J}') \in R^c(w_2)$$

□

Lemma A.5. Given any separation algebra $(\mathcal{M}, \bullet_{\mathcal{M}}, \mathbf{0}_{\mathcal{M}})$

$$\forall a, b, c, d \in \mathcal{M}.\, a \bullet_{\mathcal{M}} b = d \wedge c \leq b \implies \exists f.\, a \bullet_{\mathcal{M}} c = f$$

Proof. Let's pick arbitrary $a, b, c, d \in \mathcal{M}$ such that:

$$a \bullet_{\mathcal{M}} b = d \tag{80}$$
$$c \leq b \tag{81}$$

From (81) we obtain:

$$\exists e \in \mathcal{M}.\, c \bullet_{\mathcal{M}} e = b \tag{82}$$

Now, as a consequence, from (80) and (82) we have:

$$a \bullet_{\mathcal{M}} (c \bullet_{\mathcal{M}} e) = d \tag{83}$$

From (83) and the associativity of a pcm we obtain:

$$(a \bullet_{\mathcal{M}} c) \bullet_{\mathcal{M}} e = d \tag{84}$$

From (84) we know that $a \bullet_{\mathcal{M}} c$ is defined which implies that $\exists f \in \mathcal{M}.\, a \bullet_{\mathcal{M}} c = f$ holds. □

Lemma A.6. (Predicate introduction). If $[\![\Delta]\!]^{\mathrm{P}} \subseteq [\![\Delta']\!]^{\mathrm{P}}$ and $\Delta' \vdash \{P\}\ \mathbb{P}\ \{Q\}$ then $\Delta \vdash \{P\}\ \mathbb{P}\ \{Q\}$.

Proof. Let's pick arbitrary $\Delta, \Delta' \in \mathsf{Ax}, \mathbb{P} \in \mathsf{Prog}, P, Q \in \mathsf{Assn}$ and assume that the following holds:

$$[\![\Delta]\!]^{\mathrm{P}} \subseteq [\![\Delta']\!]^{\mathrm{P}} \tag{85}$$
$$\wedge\ \Delta' \vDash \{P\}\ \mathbb{P}\ \{Q\} \tag{86}$$

From (85) we obtain that:

$$\forall \delta.\, \delta \in [\![\Delta]\!]^{\mathrm{P}} \implies \delta \in [\![\Delta']\!]^{\mathrm{P}} \tag{87}$$

While from (86) and from the definition of $\vdash$ we know that:

$$\forall e, \delta \in [\![\Delta']\!]^{\mathrm{P}}. \vDash \left\{[\![P]\!]_{e,\delta}\right\} \;\mathbb{P}\; \left\{[\![Q]\!]_{e,\delta}\right\} \tag{88}$$

From (87) and (88) we conclude that:

$$\forall e, \delta \in [\![\Delta]\!]^{\mathrm{P}}. \vDash \left\{[\![P]\!]_{e,\delta}\right\} \;\mathbb{P}\; \left\{[\![Q]\!]_{e,\delta}\right\}$$

which from the definition of $\vdash$ means that $\Delta \vdash \left\{P\right\} \;\mathbb{P}\; \left\{Q\right\}$ as needed. □

Definition A.7. (Predicate environment similarity). We say that two predicate environments δ and δ' are *similar up to predicate* α, written $\delta \approx^{\alpha} \delta'$, if and only if:

$$\forall \beta.\, \beta \neq \alpha \implies \delta(\beta) = \delta'(\beta)$$

Lemma A.8. (Similarity satisfaction).

$$\forall e, \delta, \delta', \alpha, P.\, \delta \approx^{\alpha} \delta' \wedge \alpha \notin P \implies [\![P]\!]_{e,\delta} = [\![P]\!]_{e,\delta'}$$

Proof. Let's pick arbitrary $e \in \mathsf{LEnv}, \delta, \delta' \in \mathsf{PEnv}, \alpha \in \mathsf{PName}, P \in \mathsf{Assn}$ and assume that $\delta \approx^{\alpha} \delta' \wedge \alpha \notin P$ holds. We will now proceed with the proof using induction on the structure of assertion P.

Base case 1: $p \in \mathsf{Assn}$

It trivially follows that $[\![p]\!]_{e,\delta} = [\![p]\!]_{e,\delta'}$ since the satisfaction of logic assertion p does not depend on predicate environments δ and δ'. In fact, we check for satisfaction of p through $l, e \vDash_{\mathsf{SL}} p$.

Base case 2: $\beta(\mathbb{E}_1, \ldots, \mathbb{E}_n) \in \mathsf{Assn}$

There are two cases to consider based on the predicate name β:

- If $\beta \neq \alpha$ then from our original assumption and the definition of $\approx^{\alpha}$ we obtain that $[\![\beta(\mathbb{E}_1, \ldots, \mathbb{E}_n)]\!]_{e,\delta} = [\![\beta(\mathbb{E}_1, \ldots, \mathbb{E}_n)]\!]_{e,\delta'}$ holds.
- It can never be the case that $\beta = \alpha$ since we initially assumed that $\alpha \notin P$.

Inductive case: $P = Q \vee R \in \mathsf{Assn}$

Inductive hypothesis: Assume that the property holds for assertions Q and R meaning that for the selected α, δ, δ' we know that $\forall e.\, [\![Q]\!]_{e,\delta} = [\![Q]\!]_{e,\delta'}$ and $\forall e.\, [\![R]\!]_{e,\delta} = [\![R]\!]_{e,\delta'}$ holds.

By the assertions satisfaction definition on the $\vee$ case, we are required to show that, for any world $w \in \mathsf{World}$:

$$w, e, \delta \vDash Q \vee w, e, \delta \vDash R \tag{89}$$

$$\iff \tag{90}$$

$$w, e, \delta' \vDash Q \vee w, e, \delta' \vDash R \tag{91}$$

Both the *if* and the *only if* parts of the proof are directly satisfied by the I.H. on assertions Q and R. All other inductive cases are similar to the presented one and follow straight from their I.H. □

Lemma A.9. (Predicate elimination). If $\forall \delta \in [\![\Delta]\!]^{\mathrm{P}}. \mathsf{stable}_{\delta}(R)$ and $\alpha \notin \Delta, P, Q$ and $\Delta, (\forall \vec{x}.\, \alpha(\vec{x}) \equiv R) \vdash \left\{P\right\} \;\mathbb{P}\; \left\{Q\right\}$ then $\Delta \vdash \left\{P\right\} \;\mathbb{P}\; \left\{Q\right\}$.

Proof. Let's pick arbitrary $\Delta \in \mathsf{Ax}, \mathbb{P} \in \mathsf{Prog}, \alpha \in \mathsf{PName}, P, Q, R \in \mathsf{Assn}$ and assume that the following holds:

$$\forall \delta \in [\![\Delta]\!]^{\mathrm{P}}.\, \mathsf{stable}_{\delta}(R) \tag{92}$$

$$\wedge\ \alpha \notin \Delta, P, Q \tag{93}$$

$$\wedge\ \Delta, (\forall \vec{x}.\, \alpha(\vec{x}) \equiv R) \vDash \{P\}\ \mathbb{P}\ \{Q\} \tag{94}$$

We now pick arbitrary $\delta \in [\![\Delta]\!]^{\mathrm{P}}$ and $\delta' \in [\![\Delta, (\forall \vec{x}.\, \alpha(\vec{x}) \equiv R)]\!]^{\mathrm{P}}$. From (92), (93) and the definition of $[\![-]\!]^{\mathrm{P}}$ we obtain that:

$$\delta \approx^{\alpha} \delta' \tag{95}$$

From (93), (94), (95) and Lemma A.8 we conclude that $\Delta \vdash \{P\}\ \mathbb{P}\ \{Q\}$. $\square$

B. Auxiliary Lemmata for Serializability

Lemma B.1. All read, write or alloc operations are followed by an unlock action on the same key done by the same transaction.

$$\forall \tau, \iota, k, \kappa, x.\, x = op(\iota, k) \land x \in \tau \implies (\tau \vDash x < \mathsf{unlock}(\iota, k))$$

Proof. Let's pick an arbitrary trace $\tau \in [(\mathsf{Act}, \mathbb{N})]$, such that $\tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P})$ for some $h \in \mathsf{Storage}, S \in \mathsf{TState}$ and $\mathbb{P} \in \mathsf{Prog}$, transaction identifier $\iota \in \mathsf{Tid}$, storage key $k \in \mathsf{Key}$ and lock mode $\kappa \in \mathsf{Lock}$. Now we assume that there exists an operation $x = op(\iota, k) = (\alpha, n)$ such that $x \in \tau$ (I). So α labels one of the transitions which is part of the sequence that reduces $\mathbb{P}$ to `skip` starting with storage h (reduced to h_{end}), empty lock manager $\Phi_0 = \emptyset$ (reduced to another empty one $\Phi_{end} = \emptyset$) and empty transactions' state (reduced to S_{end}) from the definition of trace. There are now three cases to consider, for some storage value $v \in \mathsf{Val}, m \in \mathbb{N}$ and $l \in \mathsf{Key}$ such that $l \leq k < l + m$.

1. If $\alpha = \mathsf{read}(\iota, k, v)$, then it must be the case that the α action label was produced by the following Read reduction rule $(h, \emptyset, S, \mathbb{P}) \to^* (h', \Phi', S', \mathbb{P}') \xrightarrow{\alpha} (h', \Phi'', S'', \mathbb{P}'') \to^* (h_{end}, \emptyset, S_{end}, \texttt{skip})$, which, in order to succeed, requires that $\hat{\Phi}'(k) = \hat{\Phi}''(k) = (\{\iota\} \uplus I, \kappa)$ for some set of transaction identifiers I and lock mode $\kappa \geq \mathrm{S}$, meaning that $\Phi'' \neq \emptyset$.

2. If $\alpha = \mathsf{write}(\iota, k, v)$, then it must be the case that the α action label was produced by the following Write reduction rule $(h, \emptyset, S, \mathbb{P}) \to^* (h', \Phi', S', \mathbb{P}') \xrightarrow{\alpha} (h', \Phi'', S'', \mathbb{P}'') \to^* (h_{end}, \emptyset, S_{end}, \texttt{skip})$, which, in order to succeed, requires that $\hat{\Phi}'(k) = \hat{\Phi}''(k) = (\{\iota\}, \mathrm{X})$, meaning that $\Phi'' \neq \emptyset$.

3. If $\alpha = \mathsf{alloc}(\iota, m, l)$, then it must be the case that the α action label was produced by the following Alloc reduction rule $(h, \emptyset, S, \mathbb{P}) \to^* (h', \Phi', S', \mathbb{P}') \xrightarrow{\alpha} (h', \Phi'', S'', \mathbb{P}'') \to^* (h_{end}, \emptyset, S_{end}, \texttt{skip})$, which makes $\Phi'' = \Phi'[\vec{l} \mapsto (\{\iota\}, \mathrm{X})]$. Given that $l \leq k < l + m$, then $\Phi''(k) = (\{\iota\}, \mathrm{X})$ which implies that $\Phi'' \neq \emptyset$.

Now we assume that there is no action $\alpha'' = \mathsf{unlock}(\iota, k)$ such that $\tau \vDash x < \alpha''$. We know that for all cases α makes $\Phi'' \neq \emptyset$ but then by assumption (I) we know that $\mathbb{P}''$ succesfully reduces to `skip` with $\Phi_{end} = \emptyset$. This means that along the chain of reductions that brought Φ'' to Φ_{end} there has been an update to the lock manager which removed the entry associated with k. This can only happen explicitly through actions labelled with unlock that are exclusively produced by the Unlock rule. It follows that there is no possible way that $\mathbb{P}''$ reduces to `skip` with $\Phi_{end} = \emptyset$, therefore by contradiction we must have an $\alpha'' = \mathsf{unlock}(\iota, k)$ such that $\tau \vDash x < \alpha''$. □

Lemma B.2. All reads are preceded by the appropriate shared lock acquisition.

$$\forall \tau, \iota, k, v, \kappa, x, n.$$
$$x = (\mathsf{read}(\iota, k, v), n) \land x \in \tau \implies (\tau \vDash \mathsf{lock}(\iota, k, \kappa) < x \land \kappa \geq \mathrm{S})$$

Proof. Let's pick an arbitrary trace $\tau \in [\mathsf{Act} \times \mathbb{N}]$, such that $\tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P})$ for some $h \in \mathsf{Storage}, S \in \mathsf{TState}$ and $\mathbb{P} \in \mathsf{Prog}$, transaction identifier $\iota \in \mathsf{Tid}$, storage key $k \in \mathsf{Key}$, storage value $v \in \mathsf{Val}$ and lock mode $\kappa \in \mathsf{Lock}$. Now we assume that there exists an operation $x = (\alpha, n) = (\mathsf{read}(\iota, k, v), n)$ such that $x \in \tau$ (I). So α labels one of the transitions which is part of the sequence that reduces $\mathbb{P}$ to `skip` starting with storage h (reduced to h_{end}), empty lock manager $\Phi_0 = \emptyset$ (reduced to another empty one $\Phi_{end} = \emptyset$) and empty transactions' state (reduced to S_{end}) from the definition of trace. It must be the case that the α action label was produced by the following READ reduction rule $(h, \emptyset, S, \mathbb{P}) \rightarrow^* (h', \Phi', S', \mathbb{P}') \xrightarrow{\alpha} (h', \Phi', S'', \mathbb{P}'') \rightarrow^* (h_{end}, \emptyset, S_{end}, \mathtt{skip})$, which, in order to succeed, requires that $\hat{\Phi}'(k) = (\{\iota\} \uplus I, \kappa)$ for some set of transaction identifiers I and lock mode $\kappa \geq \textsc{s}$.

Then we assume that there is no action $\alpha' = \mathsf{lock}(\iota, k, \kappa)$ such that $\tau \vDash \alpha' < x$, and given that the lock manager is only updated for acquisition through reductions labelled with lock as part of the explicit LOCK rule, then there is no possible way that the state $(h', \Phi', S', \mathbb{P}')$ succesfully reduced through α since $\hat{\Phi}'(k) = (I', \kappa')$ would be such that $\iota \notin I'$. By contradiction, this means that we must have an $\alpha' = \mathsf{lock}(\iota, k, \kappa)$ such that $\tau \vDash \alpha' < x$. □

Lemma B.3. All writes to a cell are preceded by the appropriate exclusive lock acquisition.

$$\forall \tau, x, i, k, v, n.\, x = (\mathsf{write}(i, k, v), n) \wedge x \in \tau \implies \tau \vDash \mathsf{lock}(i, k, \textsc{x}) < x$$

Proof. Let's pick an arbitrary trace $\tau \in [\mathsf{Act} \times \mathbb{N}]$, such that $\tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P})$ for some $h \in \mathsf{Storage}, S \in \mathsf{TState}$ and $\mathbb{P} \in \mathsf{Prog}$, transaction identifier $i \in \mathsf{Tid}$, storage key $k \in \mathsf{Key}$. Now we assume that $x \in \tau$ where $x = (\mathsf{write}(i, k, v), n) = (\alpha, n)$ for some $n \in \mathbb{N}$. Then, α must label a reduction that brings $\mathbb{P}$ to `skip` in τ, by definition of trace. Therefore we must have the following reduction for some $h', h'', \Phi', \Phi'', S', \mathbb{P}', \mathbb{P}''$:

$$(h, \emptyset, S, \mathbb{P}) \rightarrow^* (h', \Phi', S', \mathbb{P}') \xrightarrow{\alpha} (h'', \Phi'', S', \mathbb{P}'')$$

Then we assume that there is no action $\alpha' = \mathsf{lock}(\iota, k, \textsc{x})$ such that $\tau \vDash \alpha' < x$. Given that the α label can only be generated by the WRITE rule, it is required for Φ' to be such that (I) $(\{i\}, \textsc{x}) = \hat{\Phi}'(k)$. Since the lock manager starts as an empty one ($\Phi_0 = \emptyset$), condition (I) is only satisfied when a lock label is generated by the LOCK rule before α and the lock mode obtained is exclusive (X). By contradiction we therefore obtain that $\tau \vDash \alpha' < x$ must hold. □

Lemma B.4. A read or write operation accessing a cell allocated as part of the trace, must appear after the corresponding alloc action.

$$\begin{gathered}\forall \tau, i, j, x, x', n, n', l, m, k, v, \kappa. \\ x = (\mathsf{alloc}(i, m, l), n) \wedge x' \in \{(\mathsf{read}(j, k, v), n'), (\mathsf{write}(j, k, v), n')\} \wedge l \leq k < l + m \\ \wedge x \in \tau \wedge x' \in \tau \implies \big(\tau \vDash x < x' \wedge \tau \vDash \mathsf{unlock}(i, k) < \mathsf{lock}(j, k, \kappa)\big)\end{gathered}$$

Proof. Let's pick an arbitrary trace $\tau \in [\mathsf{Act} \times \mathbb{N}]$, such that $\tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P})$ for some $h \in \mathsf{Storage}, S \in \mathsf{TState}$ and $\mathbb{P} \in \mathsf{Prog}$, transaction identifiers $i, j \in \mathsf{Tid}$, storage keys $k, l \in \mathsf{Key}, m, n, n' \in \mathbb{N}$, value $v \in \mathsf{Val}$, lock mode $\kappa \in \mathsf{Lock}$ and operations $x, x' \in \mathsf{Act} \times \mathbb{N}$. We now assume that the following holds:

$$\begin{gathered}x = (\mathsf{alloc}(i, m, l), n) \wedge x' \in \{(\mathsf{read}(j, k, v), n'), (\mathsf{write}(j, k, v), n')\} \\ \wedge\, l \leq k < l + m \wedge x \in \tau \wedge x' \in \tau\end{gathered} \tag{96}$$

From the definition of trace and (96) we obtain that the following reduction occurred as part of τ, for $\alpha' = x' \downarrow_1$:

$$(h, \emptyset, S, \mathbb{P}) \rightarrow^* (h', \Phi', S', \mathbb{P}') \xrightarrow{\alpha'} (h'', \Phi'', S'', \mathbb{P}'') \tag{97}$$

In both cases where α' is a read or a write on k, it is required for $k \in \mathsf{dom}(h')$ to hold, from their label semantic interpretation. From (96) we know that k is part of the keys allocated by x, therefore by (97), $\alpha = x \downarrow_1$ must appear in one of the reductions from $(h, \Phi, S, \mathbb{P})$ to $(h', \Phi', S', \mathbb{P}')$ which by definition means that $\tau \vDash x < x'$.

Now, from Lemma B.1 we know that there must be an action $u_i = (\mathsf{unlock}(i, k), n_u)$ for $n_u \in \mathbb{N}$, such that $\tau \vDash x < u_i$. We also know from Lemma B.3 that there exists an action $l_j = (\alpha_l, n_l) = (\mathsf{lock}(j, k, \kappa_j), n_l)$ for $\kappa_j \geq \mathrm{S}$ and $n_l \in \mathbb{N}$, such that $\tau \vDash l_j < x'$. As α sets an exclusive lock on k for transaction i, it follows that, in order for α_l to succesfully reduce, there must be an unlock operation that happens before α_l. This implies that $\tau \vDash u_i < l_j$ as needed. □

Lemma B.5. No lock is acquired by a transaction after one gets released by the same transaction.

$$\begin{gathered}\forall \tau, \iota, k, k', n, n', x, x', \kappa.\\ \big(x = (\mathsf{lock}(\iota, k, \kappa), n) \wedge x' = (\mathsf{unlock}(\iota, k'), n') \wedge x \in \tau \wedge x' \in \tau\big)\\ \implies \big(\tau \vDash x < x'\big)\end{gathered}$$

Proof. Let's pick an arbitrary trace $\tau \in [\mathsf{Act} \times \mathbb{N}]$, such that $\tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P})$ for some $h \in \mathsf{Storage}, S \in \mathsf{TState}$ and $\mathbb{P} \in \mathsf{Prog}$, transaction identifier $\iota \in \mathsf{Tid}$, storage keys $k, k' \in \mathsf{Key}$ and lock mode $\kappa \in \mathsf{Lock}$, $n, n' \in \mathbb{N}$. Now we assume that $x \in \tau$ and $x' \in \tau$, where $x = (\alpha_l, n) = (\mathsf{lock}(\iota, k, \kappa), n)$ and $x' = (\alpha_u, n') = (\mathsf{unlock}(\iota, k), n)$, therefore α_l and α_u are two lock and unlock actions, respectively, performed by the same transaction identified with ι as part of trace τ.

Let's start by assuming that the unlock operation happens before the lock one, in τ:

$$\tau \vDash x' < x \tag{98}$$

The action label unlock is only generated by the reduction rule Unlock. Therefore it must be the case that, for some $h', \Phi', \Phi'', S', S'', \mathbb{P}'$:

$$(h, \emptyset, S, \mathbb{P}) \to^* (h', \Phi', S', \mathbb{P}') \xrightarrow{\alpha_u} (h', \Phi'', S'', \mathbb{P}') \tag{99}$$

We know that the reduction in (99) can only happen when $(s', p', \mathbb{C}) \xrightarrow{\alpha_u} (s', \Upsilon, \mathbb{C})$, for some command $\mathbb{C}$ in $\mathbb{P}'$, where $(s', p') = S'(\iota)$ and $S'' = S'[\iota \mapsto (s', \Upsilon)]$.

The action label lock is only generated by the reduction rule Lock. From (98) and (99), it must be the case that, for some $h_l, S_l, S_l', \Phi_l, \Phi_l', \mathbb{P}_l$:

$$(h', \Phi'', S'', \mathbb{P}') \to^* (h_l, \Phi_l, S_l, \mathbb{P}_l) \xrightarrow{\alpha_l} (h_l, \Phi_l', S_l', \mathbb{P}_l)$$

that only reduces when $(s_l, \curlywedge, \mathbb{C}') \xrightarrow{\alpha_l} (s_l, \curlywedge, \mathbb{C}')$, for some $\mathbb{C}'$ in $\mathbb{P}_l$, where $(s_l, \curlywedge) = S_l(\iota)$. The latter condition is impossible since there is no semantic rule that allows a transaction phase to go from shrinking to growing, and $(s', \Upsilon) = S''(\iota)$ which is set before S_l. By contradiction, we obtain that $\tau \vDash x < x'$ must hold. □

Lemma B.6. If two transactions run conflicting operations on the same item, either one releases its lock before the other acquires it or vice versa.

$$\begin{gathered}\forall \tau, i, j, k, \kappa, \kappa', x, x'.\\ x = op(i, k) \in \tau \wedge x' = op(j, k) \in \tau \wedge \mathsf{conflict}(x, x')\\ \implies (\tau \vDash \mathsf{unlock}(i, k) < \mathsf{lock}(j, k, \kappa)) \vee \big(\tau \vDash \mathsf{unlock}(j, k) < \mathsf{lock}(i, k, \kappa')\big)\end{gathered}$$

Proof. Let's pick an arbitrary trace $\tau \in [\mathsf{Act} \times \mathbb{N}]$, such that $\tau = \mathsf{trace}(h, \emptyset, S, \mathbb{P})$ for some $h \in \mathsf{Storage}, S \in \mathsf{TState}$ and $\mathbb{P} \in \mathsf{Prog}$, transaction identifiers $i, j \in \mathsf{Tid}$, storage key $k \in \mathsf{Key}$,

operations $x, x' \in (\mathsf{Act}, \mathbb{N})$ and lock modes $\kappa, \kappa' \in \mathsf{Lock}$. We assume that the statements $x = op(i,k) \in \tau, x' = op(j,k) \in \tau$ and $\mathsf{conflict}(x, x')$ hold.

Let's consider the case where one of x or x' is an alloc operation. Without loss of generality we say that $x = (\mathsf{alloc}(i,m,l), n)$ for $n, m \in \mathbb{N}, l \in \mathsf{Key}$. Given that x and x' are in conflict, it follows that $x' = (\alpha', n')$ is either a read or a write on k done by a transaction j such that $i \neq j$ and $l \leq k < l + m$. We can now directly apply Lemma B.4 and obtain the required result.

The focus now shifts to the scenario where both x and x' are read or write operations on k. We reasonably assume that κ and κ' are the needed lock modes for transactions i and j to perform actions α and α' respectively. Let $\alpha_i^l = \mathsf{lock}(i,k,\kappa), \alpha_i^u = \mathsf{unlock}(i,k), \alpha_j^l = \mathsf{lock}(j,k,\kappa'), \alpha_j^u = \mathsf{unlock}(j,k)$. From Lemma B.1, Lemma B.2 and Lemma B.3 it follows that:

$$\alpha_i^l, \alpha_i^u, \alpha_j^l, \alpha_j^u \in \tau \qquad\qquad \tau \vDash \alpha_i^l < \alpha_i^u \wedge \tau \vDash \alpha_j^l < \alpha_j^u \tag{100}$$

In the case that $i = j$, meaning that the two operations are being done by the same transaction, the result follows directly from Lemma B.5 by the two-phase constraint. This leaves us to proceed with the proof considering $i \neq j$.

We now assume that $\neg\left(\tau \vDash \alpha_i^u < \alpha_j^l \vee \tau \vDash \alpha_j^u < \alpha_i^l\right)$ holds, which is equivalent to assuming that $\tau \vDash \alpha_j^l < \alpha_i^u$ and $\tau \vDash \alpha_i^l < \alpha_j^u$ hold, given that we know that all operations belong to τ and there must be an order among them. The statement, together with (100), implies that the one of the following statements must hold:

$$\tau \vDash \alpha_j^l < \alpha_i^l < \alpha_j^u < \alpha_i^u \qquad \tau \vDash \alpha_j^l < \alpha_i^l < \alpha_i^u < \alpha_j^u$$

$$\tau \vDash \alpha_i^l < \alpha_j^l < \alpha_j^u < \alpha_i^u \qquad \tau \vDash \alpha_i^l < \alpha_j^l < \alpha_i^u < \alpha_j^u$$

Without loss of generality, for the rest of the proof we will use the first scenario, since an equivalent argument with the appropriate substitutions can be made for the other three.

Therefore, the reduction chain which generated τ and that brings $\mathbb{P}$ to `skip` (by definition of trace) must have the following shape, for some $h_1, \Phi_1, S_1, \mathbb{P}_1, \Phi_1'$, $h_2, \Phi_2, S_2, \mathbb{P}_2, \Phi_2'$, $h_3, \Phi_3, S_3, \mathbb{P}_3, \Phi_3', S_3'$, $h_4, \Phi_4, S_4, \mathbb{P}_4, \Phi_4', S_4'$.

$$\begin{aligned}(h, \emptyset, S, \mathbb{P}) \to^* (h_1, \Phi_1, S_1, \mathbb{P}_1) \xrightarrow{\alpha_j^l} (h_1, \Phi_1', S_1, \mathbb{P}_1) \to^* (h_2, \Phi_2, S_2, \mathbb{P}_2) \xrightarrow{\alpha_i^l} (h_2, \Phi_2', S_2, \mathbb{P}_2)\\ \to^* (h_3, \Phi_3, S_3, \mathbb{P}_3) \xrightarrow{\alpha_j^u} (h_3, \Phi_3', S_3', \mathbb{P}_3) \to^* (h_4, \Phi_4, S_4, \mathbb{P}_4) \xrightarrow{\alpha_i^u} (h_4, \Phi_4', S_4', \mathbb{P}_4)\end{aligned}$$

By our initial assumption, we know that the α and α' actions are conflicting, meaning that one of them (or both) is a write operation on k. Without loss of generality, we will consider the situation where α is a write action, noting that an equivalent proof can be obtained by having α' a write and making the appropriate substitutions. The transition labelled as α_j^l updates the lock manager Φ_1 to be:

$$\Phi_1' = \Phi_1[k \mapsto (\{j\} \uplus I, \kappa)] \tag{101}$$

for some (potentially empty) set of transaction identifiers I.

Now we can establish that there cannot be any $\alpha_j^{u'} = \mathsf{unlock}(j,k)$ such that:

$$\tau \vDash \alpha_j^l < \alpha_j^{u'} < \alpha_i^l \tag{102}$$

Let's on the contrary assume that there exists such $\alpha_j^{u'}$. Then it must be the case that $\tau \vDash \alpha_j^{u'} < \alpha_j^u$ since $\tau \vDash \alpha_i^l < \alpha_j^u$ and $\tau \vDash \alpha_j^{u'} < \alpha_i^l$. This cannot be possible, given that from Lemma C.4 we know that a single transaction cannot unlock a particular item twice. By contradiction we obtain that no such $\alpha_j^{u'}$ exists.

From (101) and (102) we obtain that $(\{j\} \uplus I, \kappa) = \hat{\Phi}_2(k)$ meaning that the set of owners of k is non-empty and moreover it definitely contains j. It follows that there is no possible way of the α_i^l transition happening since it must acquire an exclusive lock on k and and the transition can only be produced by the LOCK rule which, in this situation, requires $(\emptyset, \text{U}) = \hat{\Phi}_2(k)$ or $(\{i\}, \text{S}) = \hat{\Phi}_2(k)$. By contradiction we get that $\tau \vDash \alpha_i^u < \alpha_j^l \vee \tau \vDash \alpha_j^u < \alpha_i^l$ must hold. □

C. Auxiliary Lemmata for Equivalence

Lemma C.1. Lock and unlock operations done by a transaction on items which it does not read or write can be removed without affecting the program or the global state.

$$\begin{gathered}\forall \tau, \tau', h, h', \Phi, S, \mathbb{P}, n, n', \iota, k, \kappa, x, y. \\ \mathsf{tgen}(\tau, h, h', \Phi, S, \mathbb{P}) \wedge \mathsf{absent}(\iota, k, \tau) \wedge x = (\mathsf{lock}(\iota, k, \kappa), n) \wedge y = (\mathsf{unlock}(\iota, k), n') \\ \wedge\, x \in \tau \wedge y \in \tau \wedge \tau' = \tau \setminus \{x, y\} \implies \mathsf{tgen}(\tau', h, h', \Phi, S, \mathbb{P})\end{gathered}$$

Proof. Let's pick arbitrary $\tau, \tau' \in [\mathsf{Act} \times \mathbb{N}], h, h' \in \mathsf{Storage}, \Phi \in \mathsf{LMan}, S \in \mathsf{TState}, \mathbb{P} \in \mathbb{P}, n, n' \in \mathbb{N}, \iota \in \mathsf{Tid}, k \in \mathsf{Key}, \kappa \in \mathsf{Lock}, x, y \in \mathsf{Act} \times \mathbb{N}$. We now assume that the following holds:

$$\begin{gathered}\mathsf{tgen}(\tau, h, h', \Phi, S, \mathbb{P}) \wedge \mathsf{absent}(\iota, k, \tau) \wedge x = (\mathsf{lock}(\iota, k, \kappa), n) \wedge y = (\mathsf{unlock}(\iota, k), n') \\ \wedge\, x \in \tau \wedge y \in \tau \wedge \tau' = \tau \setminus \{x, y\}\end{gathered}$$

From Lemma B.5 we obtain that $\tau \vDash x < y$. From the definition of tgen and the fact that both x and y are in τ it follows that $\kappa \geq \mathrm{S}$ and:

$$(h, \Phi, S, \mathbb{P}) \to^* (h_1, \Phi_1, S_1, \mathbb{P}_1) \xrightarrow{\mathsf{lock}(\iota,k,\kappa)} (h_1', \Phi_1', S_1', \mathbb{P}_1') \tag{103}$$

$$\to^* (h_2, \Phi_2, S_2, \mathbb{P}_2) \xrightarrow{\mathsf{unlock}(\iota,k)} (h_2', \Phi_2', S_2', \mathbb{P}_2') \to^* (h', \emptyset, S', \mathtt{skip}) \tag{104}$$

From the semantic interpretation of lock and unlock, we know that it is the case that $h_1' = h_1, S_1' = S_1, \mathbb{P}_1' = \mathbb{P}_1, h_2' = h_2, S_2' = S_2, \mathbb{P}_2' = \mathbb{P}_2$ and $\Phi_1' = \Phi_1[k \mapsto (\{\iota\} \uplus I, \kappa)], \Phi_2' = \Phi_2[k \mapsto (I', \kappa')]$ for $I, I' \in \mathcal{P}(\mathsf{Tid})$ such that $\iota \notin I'$ and $\kappa' \leq \mathrm{S}$. From the assumption that $\mathsf{absent}(\iota, k, \tau)$ holds, we know there is no read or write action on k done by ι happening in $(h_1', \Phi_1', S_1', \mathbb{P}_1') \to^* (h_2, \Phi_2, S_2, \mathbb{P}_2)$ meaning that actions which need a presence of ι's lock acquisition on k to succeed (i.e. read and write) are not there. From Lemma C.3 we obtain that all actions that are part of the sequence of reductions $(h_1', \Phi_1', S_1', \mathbb{P}_1') \to^* (h_2, \Phi_2, S_2, \mathbb{P}_2)$ will succesfully reduce with the Φ_1 lock manager not containing ι as an owner for k. It follows that, for $(\alpha, n+1) \in \tau$ and $(\alpha', n'+1) \in \tau$:

$$(h, \Phi, S, \mathbb{P}) \to^* (h_1, \Phi_1, S_1, \mathbb{P}_1) \xrightarrow{\alpha} (h_1'', \Phi_1'', S_1'', \mathbb{P}_1'') \tag{105}$$

$$\to^* (h_2'', \Phi_2'', S_2'', \mathbb{P}_2'') \xrightarrow{\alpha'} (h_2, \Phi_2, S_2, \mathbb{P}_2) \to^* (h', \emptyset, S', \mathtt{skip}) \tag{106}$$

From the initial assumption we also know that $\tau' = \tau \setminus \{x, y\}$ meaning that τ' has all of τ's actions a part from the ones at position n and n'. As a consequence we have that by following τ' up to (and not including) position n we have $(h, \Phi, S, \mathbb{P}) \to^* (h_1, \Phi_1, S_1, \mathbb{P}_1)$. Skipping the operation at position n which is not present in τ', we proceed with the one in position $n+1$ all the way to (and not including) the one in position n' to get by (105) and (106) that $(h_1, \Phi_1, S_1, \mathbb{P}_1) \xrightarrow{\alpha} (h_1'', \Phi_1'', S_1'', \mathbb{P}_1'') \to^* (h_2'', \Phi_2'', S_2'', \mathbb{P}_2'')$ holds. Now we apply the action from position $n'+1$ to the end in τ' to obtain $(h_2'', \Phi_2'', S_2'', \mathbb{P}_2'') \xrightarrow{\alpha'} (h_2, \Phi_2, S_2, \mathbb{P}_2) \to^* (h', \emptyset, S', \mathtt{skip})$. From the definition of tgen we can state that $\mathsf{tgen}(\tau', h, h', \Phi, S, \mathbb{P})$ holds as needed. □

Lemma C.2. Redundant exclusive lock operations present in a trace, can be converted to the corresponding shared equivalent without affecting the end storage.

$$\forall \tau, \tau', h, h', \Phi, S, \mathbb{P}, n, \iota, k, x.$$
$$\mathsf{tgen}(\tau, h, h', \Phi, S, \mathbb{P}) \wedge \mathsf{redundant}(\iota, k, \mathrm{X}, \tau) \wedge x = (\mathsf{lock}(\iota, k, \mathrm{X}), n)$$
$$\wedge\, x \in \tau \wedge \tau' = \mathsf{swap}(\tau, x, (\mathsf{lock}(\iota, k, \mathrm{S}), n)) \implies \mathsf{tgen}(\tau', h, h', \Phi, S, \mathbb{P})$$

Proof. Let's pick arbitrary $\tau, \tau' \in [\mathsf{Act} \times \mathbb{N}], h, h' \in \mathsf{Storage}, \Phi \in \mathsf{LMan}, S \in \mathsf{TState}, \mathbb{P} \in \mathbb{P}, n \in \mathbb{N}, \iota \in \mathsf{Tid}, k \in \mathsf{Key}, x \in \mathsf{Act} \times \mathbb{N}$. We now assume that the following holds:

$$\mathsf{tgen}(\tau, h, h', \Phi, S, \mathbb{P}) \wedge x = (\mathsf{lock}(\iota, k, \mathrm{X}), n) \wedge x \in \tau \tag{107}$$
$$\wedge\, \mathsf{redundant}(\iota, k, \mathrm{X}, \tau) \wedge \tau' = \mathsf{swap}(\tau, x, (\mathsf{lock}(\iota, k, \mathrm{S}), n)) \tag{108}$$

The above means that by following the actions in τ we can generate the storage h' starting from the $h, \Phi, S, \mathbb{P}$ state and that the same τ includes an exclusive lock operation $x = \mathsf{lock}(\iota, k, \mathrm{X})$ done by transaction ι on storage key k which is redundant. This means that ι never writes to k as part of τ. The statement also expresses that there is a trace τ' which is equivalent to τ with operation x converted into $\mathsf{lock}(\iota, k, \mathrm{S})$.

From (107) we know that there must be a reduction of the following shape, for $S' \in \mathsf{TState}, \alpha = x \downarrow_1$ and $\alpha' = \mathsf{unlock}(\iota, k)$:

$$\begin{gathered}(h, \Phi, S, \mathbb{P}) \to^* (h_l, \Phi_l, S_l, \mathbb{P}_l) \xrightarrow{\alpha} (h'_l, \Phi'_l, S'_l, \mathbb{P}'_l) \to^* \\ (h_a, \Phi_a, S_a, \mathbb{P}_a) \xrightarrow{\dot\alpha} (h'_a, \Phi'_a, S'_a, \mathbb{P}'_a) \to^* \\ (h_u, \Phi_u, S_u, \mathbb{P}_u) \xrightarrow{\alpha'} (h'_u, \Phi'_u, S'_u, \mathbb{P}'_u) \to^* (h', \emptyset, S', \mathtt{skip})\end{gathered} \tag{109}$$

Given the fact that x acquires an exclusive lock on k, it must be the case that $\Phi'_l(k) = (\{\iota\}, \mathrm{X})$. From (109) we obtain that all actions that accesses k done by transaction ι, i.e. $\dot\alpha$, must occur in the reduction sequence:

$$(h'_l, \Phi'_l, S'_l, \mathbb{P}'_l) \to^* (h_u, \Phi_u, S_u, \mathbb{P}_u) \tag{110}$$

Let's now convert action α into $\mathsf{lock}(\iota, k, \mathrm{S})$, making $\Phi'_l(k) = (\{\iota\}, \mathrm{S})$. From Lemma C.3 we know that the only actions in the reductions (110) that need the presence of a lock introduced on k are $\dot\alpha \in \{\mathsf{read}(\iota, k, v), \mathsf{write}(\iota, k, v), \mathsf{unlock}(\iota, k)\}$. Given that $\alpha' = \mathsf{unlock}(\iota, k)$ and form Lemma C.4 a single transaction can only unlock a particular cell once, there cannot be an unlock operation on k which happens as part of the sequence of reductions in (110). It follows that $\dot\alpha \neq \mathsf{unlock}(\iota, k)$ and $\Phi_a(k) = \Phi'_l = (\{\iota\}, \mathrm{S})$.

1. From (108) and the definition of $\mathsf{redundant}$ we know that $\dot\alpha$ cannot be $\mathsf{write}(\iota, k, v)$.

2. If $\dot\alpha = \mathsf{read}(\iota, k, v)$ then all it is required for this transition to succeed is for $k \in \mathsf{dom}(h)$ which must hold from the fact that $\dot\alpha$ suceeded in τ and $\iota \in I$ for $(I, \kappa) = \Phi_a(k)$ which we know holds since $\Phi_a(k) = (\{\iota\}, \mathrm{S})$.

We have proven that we can reach program state $(h_u, \Phi_u, S_u, \mathbb{P}_u)$ even by replacing the original α with $\mathsf{lock}(\iota, k, \mathrm{S})$, i.e. by following the actions in τ'. Now α' can always reduce since it needs $\iota \in I$ for $\Phi_u(k) = (I, \mathrm{S})$ which will be the case since the new α added a lock on k and no unlock operation removed it since. From program state $(h'_u, \Phi'_u, S'_u, \mathbb{P}'_u)$ we can follow the rest of the actions in τ' which in this part will be equivalent to the ones in τ to reach the final state $(h', \emptyset, S', \mathtt{skip})$. From the definition of tgen the following must hold:

$$\mathsf{tgen}(\tau', h, h', \Phi, S, \mathbb{P})$$

□

Lemma C.3. A lock on an item is not needed for any reductions a part from a read, a write or an unlock action performed by the same transaction on the same item.

$$\forall \mathbb{P}, \mathbb{P}', h, h', \Phi, \Phi', S, S', \alpha, i, k, v, I, \kappa.$$
$$(h, \Phi, S, \mathbb{P}) \xrightarrow{\alpha} (h', \Phi', S', \mathbb{P}') \wedge (\{i\} \uplus I, \kappa) = \Phi(k) \wedge$$
$$\alpha \notin \{\mathsf{read}(i, k, v), \mathsf{write}(i, k, v), \mathsf{unlock}(i, k)\} \implies \exists \Phi_m, \Phi'_m, I', \kappa', \kappa''.$$
$$(h, \Phi_m, S, \mathbb{P}) \xrightarrow{\alpha} (h', \Phi'_m, S, \mathbb{P}') \wedge \Phi_m = \Phi[k \mapsto (I, \kappa')] \wedge \Phi'_m = \Phi'[k \mapsto (I', \kappa'')] \wedge \kappa' \leq \mathrm{S}$$

Proof. Let's pick arbitrary $\mathbb{P}, \mathbb{P}' \in \mathsf{Prog}, h, h' \in \mathsf{Storage}, \Phi, \Phi' \in \mathsf{LMan}, S, S' \in \mathsf{TState}, \alpha \in \mathsf{Act}, i \in \mathsf{Tid}, k \in \mathsf{Key}, v \in \mathsf{Val}, I \in \mathcal{P}(\mathsf{Tid}), \kappa \in \mathsf{Lock}$. We now assume that the following holds:

$$(h, \Phi, S, \mathbb{P}) \xrightarrow{\alpha} (h', \Phi', S', \mathbb{P}') \wedge (\{i\} \uplus I, \kappa) = \Phi(k) \wedge \alpha \notin \{\mathsf{read}(i, k, v), \mathsf{write}(i, k, v)\} \quad (111)$$

From (111) we directly obtain that $\kappa \geq \mathrm{S}$ given that i is in the owners' set for item k. The proof proceeds with a case-by-case analysis on α.

- If $\alpha = \mathsf{sys}$, $\alpha = \mathsf{id}(\iota)$ or $\alpha = \mathsf{alloc}(\iota, n, l)$ for some $\iota \in \mathsf{Tid}, n \in \mathbb{N}, l \in \mathsf{Key}$, then the result trivially follows given that in these cases α has no requirementes on Φ to succesfully reduce.
- If $\alpha = \mathsf{read}(j, k', v')$ for $j \in \mathsf{Tid}, k' \in \mathsf{Key}, v' \in \mathsf{Val}$ then from (111) we know that $i \neq j$. Next we consider the following two cases:
  - If $k = k'$ then from (111) we obtain that, given the α action has succesfully reduced, $\kappa = \mathrm{S}$ and $j \in I$. Therefore we can find $\kappa' = \mathrm{S}, \kappa'' = \mathrm{S}$ and $I' = I$.
  - If $k \neq k'$ then α has no requirement on $\Phi(k)$ to succesfully reduce and the result follows.
- If $\alpha = \mathsf{write}(j, k', v')$ for $j \in \mathsf{Tid}, k' \in \mathsf{Key}, v' \in \mathsf{Val}$ then from (111) we know that $i \neq j$. Next we consider the following two cases:
  - If $k = k'$ then it is not possible that α succesfully reduced since from (111) we know that i was in the owners set for key k, then it must be the case that $k \neq k'$.
  - If $k \neq k'$ then α has no requirement on $\Phi(k)$ to succesfully reduce and the result follows.
- If $\alpha = \mathsf{lock}(j, k', \kappa_j)$ for some $j \in \mathsf{Tid}, k' \in \mathsf{Key}, \kappa_j \in \mathsf{Lock}$.
  - If $k \neq k'$ then α has no requirement on $\Phi(k)$ to succesfully reduce and the result follows.
  - If $k = k'$ and $i \neq j$ then from (111) we know that α succesfully reduced, and therefore $\kappa = \mathrm{S}$ and $\kappa_j = \mathrm{S}$. This also implies that we can find $\kappa' = \mathrm{S}, I' = I \uplus \{j\}$ and $\kappa'' = \mathrm{S}$.
  - If $k = k'$ and $i = j$ then given that $\Phi(k)$ already had i as part of the owners, it must be the case that $\kappa = \mathrm{S}, I = \emptyset$ and $\kappa_j = \mathrm{X}$ in order for α to reduce as imposed by (111). It follows that we can find $\kappa' = \mathrm{U}, I' = \{i\}$ and $\kappa'' = \mathrm{X}$.
- If $\alpha = \mathsf{unlock}(j, k')$ for $j \in \mathsf{Tid}, k' \in \mathsf{Key}$ then from (111) we know that $i \neq j$. Next we consider the following two cases:
  - If $k \neq k'$ then α has no requirement on $\Phi(k)$ to succesfully reduce and the result follows.
  - If $k = k'$ then from (111) we know that α succesfully reduced meaning that i and j were holding the lock at the same time, making $\kappa = \mathrm{S}, \kappa' = \mathrm{S}, \kappa'' = \mathrm{S}$ and $I' = I \backslash \{i, j\}$.

$\square$

Lemma C.4. A transaction cannot unlock a particular item twice.

$$\begin{gathered}\forall h, h', h'', \Phi, \Phi'', S, S', S'', \mathbb{P}, \mathbb{P}''. \\ (h, \Phi, S, \mathbb{P}) \xrightarrow{\mathsf{unlock}(\iota,k)} (h'', \Phi'', S'', \mathbb{P}'') \rightarrow^* (h', \emptyset, S', \mathtt{skip}) \\ \implies \neg\exists h_u, h'_u, \Phi_u, \Phi'_u, S_u, S'_u, \mathbb{P}_u, \mathbb{P}'_u. \\ (h'', \Phi'', S'', \mathbb{P}'') \rightarrow^* (h_u, \Phi_u, S_u, \mathbb{P}_u) \xrightarrow{\mathsf{unlock}(\iota,k)} (h'_u, \Phi'_u, S'_u, \mathbb{P}'_u)\end{gathered}$$

Let's pick arbitrary $h, h', h'' \in \mathsf{Storage}, \Phi, \Phi'' \in \mathsf{LMan}, S, S', S'' \in \mathsf{TState}, \mathbb{P}, \mathbb{P}'' \in \mathsf{Prog}$ and assume that the following holds:

$$(h, \Phi, S, \mathbb{P}) \xrightarrow{\mathsf{unlock}(\iota,k)} (h'', \Phi'', S'', \mathbb{P}'') \rightarrow^* (h', \emptyset, S', \mathtt{skip}) \tag{112}$$

From (112) we know that for $\iota \notin I, \kappa \leq \mathrm{S}$ and $(s, p) = S(\iota)$ we have:

$$\Phi''(k) = (I, \kappa) \tag{113}$$

$$S''(\iota) = (s, \curlyvee) \tag{114}$$

Now let's also assume that the following holds:

$$\begin{gathered}\exists h_u, h'_u, \Phi_u, \Phi'_u, S_u, S'_u, \mathbb{P}_u, \mathbb{P}'_u. \\ (h'', \Phi'', S'', \mathbb{P}'') \rightarrow^* (h_u, \Phi_u, S_u, \mathbb{P}_u) \xrightarrow{\mathsf{unlock}(\iota,k)} (h'_u, \Phi'_u, S'_u, \mathbb{P}'_u)\end{gathered} \tag{115}$$

From (115) we know that it must be the case that $\Phi_u(k) = (\{\iota\} \uplus I', \kappa')$ for some $I' \in \mathcal{P}(\mathsf{Tid})$ and $\kappa' \geq \mathrm{S}$. Then, from (113) we obtain that there must be an action $\alpha = \mathsf{lock}(\iota, k, \kappa')$ in the following sequence of reductions:

$$(h'', \Phi'', S'', \mathbb{P}'') \rightarrow^* (h_u, \Phi_u, S_u, \mathbb{P}_u) \tag{116}$$

From (114) we know that at the start of the reductions in (116), ι is in the shrinking phase and cannot acquire any lock. This means that no such α exists and the reduction in (115) is impossible. By contradiction we obtain that the following must hold:

$$\begin{gathered}\neg\exists h_u, h'_u, \Phi_u, \Phi'_u, S_u, S'_u, \mathbb{P}_u, \mathbb{P}'_u. \\ (h'', \Phi'', S'', \mathbb{P}'') \rightarrow^* (h_u, \Phi_u, S_u, \mathbb{P}_u) \xrightarrow{\mathsf{unlock}(\iota,k)} (h'_u, \Phi'_u, S'_u, \mathbb{P}'_u)\end{gathered}$$

Lemma C.5.

$$\mathsf{cAtom}(\mathbb{C}) \triangleq \begin{array}{l}\forall \iota, s, s', p, p', \Phi, h, h', \mathbb{C}', \alpha. \\ (s, p, \mathbb{C}) \xrightarrow{\alpha}_\iota (s', p', \mathbb{C}') \wedge h' = (\llbracket \alpha \rrbracket (h, \Phi) \downarrow_1) \\ \quad \implies (h, s, \mathbb{C}) \rightarrow_{\text{ATOM}} (h', s', \mathbb{C}')\end{array}$$

Proof. $\forall \mathbb{C} \in \mathsf{Cmd}. \mathsf{cAtom}(\mathbb{C})$ by induction on the structure of commands Cmd.

Base case 1: $\mathtt{skip} \in \mathsf{Cmd}$
To show: $\mathsf{cAtom}(\mathtt{skip})$
The `skip` case follows trivially since there is no possible one-step α reduction that involves it.

Base case 2: $\mathtt{x} := \mathbb{E} \in \mathsf{Cmd}$
To show: $\mathsf{cAtom}(\mathtt{x} := \mathbb{E})$
For arbitrary $\iota, s, s', p, p', \Phi, h, h', \mathbb{C}', \alpha$ we assume that $(s, p, \mathtt{x} := \mathbb{E}) \xrightarrow{\alpha}_\iota (s', p', \mathbb{C}') \wedge h' = (\llbracket \alpha \rrbracket (h, \Phi) \downarrow_1)$ holds. The only way to reduce $\mathtt{x} := \mathbb{E}$ is through the ASSIGN rule which makes

$\alpha = \mathsf{id}(\iota), \mathbb{C}' = \mathtt{skip}, s' = s[\mathtt{x} \mapsto v], p' = p, h' = h$ where $v = [\![\mathbb{E}]\!]^{\mathrm{E}}_s$. It follows that we have all the requisites to reduce $(h, s, \mathtt{x} := \mathbb{E}) \rightarrow_{\text{ATOM}} (h, s[\mathtt{x} \mapsto v], \mathtt{skip})$ through the AtAssign rule.

Base case 3: $\mathtt{x} := [\mathbb{E}] \in \mathsf{Cmd}$
To show: $\mathsf{cAtom}(\mathtt{x} := [\mathbb{E}])$
For arbitrary $\iota, s, s', p, p', \Phi, h, h', \mathbb{C}', \alpha$ we assume that $(s, p, \mathtt{x} := [\mathbb{E}]) \xrightarrow{\alpha} (s', p', \mathbb{C}') \wedge h' = ([\![\alpha]\!](h, \Phi) \downarrow_1)$ holds. The only way to reduce $\mathtt{x} := [\mathbb{E}]$ is through the Read rule which makes $\mathbb{C}' = \mathtt{skip}, h' = h, s' = s[\mathtt{x} \mapsto v], \alpha = \mathsf{read}(\iota, k, v)$ where $k = [\![\mathbb{E}]\!]^{\mathrm{E}}_s$ and $v = h(k)$. It follows that we have all the requisites to reduce $(h, s, \mathtt{x} := [\mathbb{E}]) \rightarrow_{\text{ATOM}} (h, s[\mathtt{x} \mapsto v], \mathtt{skip})$ through the AtRead rule.

Base case 4: $\mathtt{x} := \mathtt{alloc}(\mathbb{E}) \in \mathsf{Cmd}$
To show: $\mathsf{cAtom}(\mathtt{x} := \mathtt{alloc}(\mathbb{E}))$
For arbitrary $\iota, s, s', p, p', \Phi, h, h', \mathbb{C}', \alpha$ we assume that $(s, p, \mathtt{x} := \mathtt{alloc}(\mathbb{E})) \xrightarrow{\alpha} (s', p', \mathbb{C}') \wedge h' = ([\![\alpha]\!](h, \Phi) \downarrow_1)$ holds. The only way to reduce $\mathtt{x} := \mathtt{alloc}(\mathbb{E})$ is through the Alloc rule which makes $\mathbb{C}' = \mathtt{skip}, h' = h[l \mapsto 0] \ldots [l + n - 1 \mapsto 0], s' = s[\mathtt{x} \mapsto l], \alpha = \mathsf{alloc}(\iota, n, l)$ where $n = [\![\mathbb{E}]\!]^{\mathrm{E}}_s, n > 0$ and $\{l, \ldots, l + n - 1\} \cap \mathsf{dom}(h) \equiv \emptyset$. It follows that we have all the requisites to reduce $(h, s, \mathtt{x} := \mathtt{alloc}(\mathbb{E})) \rightarrow_{\text{ATOM}} (h[l \mapsto 0] \ldots [l + n - 1 \mapsto 0], s[\mathtt{x} \mapsto l], \mathtt{skip})$ through the AtAlloc rule.

Base case 5: $[\mathbb{E}_1] := \mathbb{E}_2 \in \mathsf{Cmd}$
To show: $\mathsf{cAtom}([\mathbb{E}_1] := \mathbb{E}_2)$
For arbitrary $\iota, s, s', p, p', \Phi, h, h', \mathbb{C}', \alpha$ we assume that $(s, p, [\mathbb{E}_1] := \mathbb{E}_2) \xrightarrow{\alpha} (s', p', \mathbb{C}') \wedge h' = ([\![\alpha]\!](h, \Phi) \downarrow_1)$ holds. The only way to reduce $[\mathbb{E}_1] := \mathbb{E}_2$ is through the Write rule which makes $\mathbb{C}' = \mathtt{skip}, h' = h[k \mapsto v], s' = s, \alpha = \mathsf{write}(\iota, k, v)$ where $k = [\![\mathbb{E}_1]\!]^{\mathrm{E}}_s, v = [\![\mathbb{E}_2]\!]^{\mathrm{E}}_s$ and $k \in \mathsf{dom}(h)$. It follows that we have all the requisites to reduce $(h, s, [\mathbb{E}_1] := \mathbb{E}_2) \rightarrow_{\text{ATOM}} (h[k \mapsto v], s, \mathtt{skip})$ through the AtWrite rule.

Inductive case 1: $\mathbb{C}_1; \mathbb{C}_2 \in \mathsf{Cmd}$
Inductive hypothesis: $\mathsf{cAtom}(\mathbb{C}_1) \wedge \mathsf{cAtom}(\mathbb{C}_2)$
To show: $\mathsf{cAtom}(\mathbb{C}_1; \mathbb{C}_2)$
For arbitrary $\iota, s, s', p, p', \Phi, h, h', \mathbb{C}', \alpha$ we assume that $(s, p, \mathbb{C}_1; \mathbb{C}_2) \xrightarrow{\alpha} (s', p', \mathbb{C}') \wedge h' = ([\![\alpha]\!](h, \Phi) \downarrow_1)$ holds. There are two possible ways to reduce $\mathbb{C}_1; \mathbb{C}_2$.

1. When $\mathbb{C}_1 = \mathtt{skip}$, we can reduce $(s, p, \mathtt{skip}; \mathbb{C}_2) \xrightarrow{\mathsf{id}(\iota)} (s, p, \mathbb{C}_2)$ through the SeqSkip rule, therefore $\mathbb{C}' = \mathbb{C}_2$. We can do the same using the AtSeqSkip rule by reducing $(h, s, \mathtt{skip}; \mathbb{C}_2) \rightarrow_{\text{ATOM}} (h, s, \mathbb{C}_2)$.

2. When $\mathbb{C}_1 \neq \mathtt{skip}$, we can reduce $(s, p, \mathbb{C}_1; \mathbb{C}_2) \xrightarrow{\alpha} (s', p', \mathbb{C}'_1; \mathbb{C}_2)$ through the SeqSkip rule, making $\mathbb{C}' = \mathbb{C}'_1; \mathbb{C}_2$, by running $(s, p, \mathbb{C}_1) \xrightarrow{\alpha} (s', p', \mathbb{C}'_1)$. From inductive hypothesis on $\mathbb{C}_1$ we obtain that $(h, s, \mathbb{C}_1) \rightarrow_{\text{ATOM}} (h', s', \mathbb{C}'_1)$ which is the premiss of the AtSeq rule to reduce $(h, s, \mathbb{C}_1; \mathbb{C}_2) \rightarrow_{\text{ATOM}} (h', s', \mathbb{C}'_1; \mathbb{C}_2)$.

Inductive case 2: $\mathtt{if}\ (\mathbb{B})\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2 \in \mathsf{Cmd}$
Inductive hypothesis: $\mathsf{cAtom}(\mathbb{C}_1) \wedge \mathsf{cAtom}(\mathbb{C}_2)$
To show: $\mathsf{cAtom}(\mathtt{if}\ (\mathbb{B})\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2)$
For arbitrary $\iota, s, s', p, p', \Phi, h, h', \mathbb{C}', \alpha$ we assume that $(s, p, \mathtt{if}\ (\mathbb{B})\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2) \xrightarrow{\alpha} (s', p', \mathbb{C}') \wedge h' = ([\![\alpha]\!](h, \Phi) \downarrow_1)$ holds. There are two possible ways to reduce $\mathtt{if}\ (\mathbb{B})\ \mathbb{C}_1\ \mathtt{else}\ \mathbb{C}_2$ for $b = [\![\mathbb{B}]\!]^{\mathrm{B}}_s$.

1. When $b = \top$, we can reduce $(s, p, \texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2) \xrightarrow{\mathsf{id}(\iota)} (s, p, \mathbb{C}_1)$ through the CondT rule. It is possible to do the same using the AtCondT rule in order to obtain:

$$(h, s, \texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2) \rightarrow_{\text{Atom}} (h, s, \mathbb{C}_1)$$

2. When $b = \bot$, we can reduce $(s, p, \texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2) \xrightarrow{\mathsf{id}(\iota)} (s, p, \mathbb{C}_2)$ through the CondF rule. It is possible to do the same using the AtCondF rule in order to obtain:

$$(h, s, \texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2) \rightarrow_{\text{Atom}} (h, s, \mathbb{C}_2)$$

Inductive case 3: $\texttt{while}\,(\mathbb{B})\ \mathbb{C} \in \mathsf{Cmd}$
Inductive hypothesis: $\mathsf{cAtom}(\mathbb{C})$
To show: $\mathsf{cAtom}(\texttt{while}\,(\mathbb{B})\ \mathbb{C})$
For arbitrary $\iota, s, s', p, p', \Phi, h, h', \mathbb{C}', \alpha$ we assume that $(s, p, \texttt{while}\,(\mathbb{B})\ \mathbb{C}) \xrightarrow{\alpha} (s', p', \mathbb{C}') \wedge h' = (\llbracket \alpha \rrbracket(h, \Phi) \downarrow_1)$ holds. There are two possible ways to reduce $\texttt{while}\,(\mathbb{B})\ \mathbb{C}$ for $b = \llbracket \mathbb{B} \rrbracket^{\mathbb{B}}_s$.

1. When $b = \top$, we can reduce $(s, p, \texttt{while}\,(\mathbb{B})\ \mathbb{C}) \xrightarrow{\mathsf{id}(\iota)} (s, p, \mathbb{C}; \texttt{while}\,(\mathbb{B})\ \mathbb{C})$ through the LoopT. In a similar way we can reduce $(h, s, \texttt{while}\,(\mathbb{B})\ \mathbb{C}) \rightarrow_{\text{Atom}} (h, s, \mathbb{C}; \texttt{while}\,(\mathbb{B})\ \mathbb{C})$ via the AtLoopT rule.

2. When $b = \bot$, we can reduce $(s, p, \texttt{while}\,(\mathbb{B})\ \mathbb{C}) \xrightarrow{\mathsf{id}(\iota)} (s, p, \texttt{skip})$ through the LoopF. In a similar way we can reduce $(h, s, \texttt{while}\,(\mathbb{B})\ \mathbb{C}) \rightarrow_{\text{Atom}} (h, s, \texttt{skip})$ via the AtLoopF rule.

□

Lemma C.6.

$$\forall h, h', \mathbb{P}_1, \mathbb{P}_2.\, (h, \mathbb{P}_1) \rightarrow^*_{\text{Atom}} (h', \texttt{skip}) \implies (h, \mathbb{P}_1; \mathbb{P}_2) \rightarrow^*_{\text{Atom}} (h', \texttt{skip}; \mathbb{P}_2)$$

Proof. We proceed by induction on n, i.e. the number of reduction steps in $\rightarrow^*_{\text{Atom}}$.

Base case: $n = 0$
To show:

$$\forall h, h', \mathbb{P}_1, \mathbb{P}_2.\, (h, \mathbb{P}_1) \rightarrow^0_{\text{Atom}} (h', \texttt{skip}) \implies (h, \mathbb{P}_1; \mathbb{P}_2) \rightarrow^*_{\text{Atom}} (h', \texttt{skip}; \mathbb{P}_2)$$

We assume $(h, \mathbb{P}_1) \rightarrow^0_{\text{Atom}} (h', \texttt{skip})$ holds and given it is a zero-step reduction, the only possible case is for $\mathbb{P}_1 = \texttt{skip}$. Therefore $(h, \texttt{skip}) \rightarrow^0_{\text{Atom}} (h', \texttt{skip})$ where $h = h'$. Now we have $(h, \texttt{skip}; \mathbb{P}_2) \rightarrow^0_{\text{Atom}} (h', \texttt{skip}; \mathbb{P}_2)$ given that $\rightarrow^0_{\text{Atom}}$ is a reflexive relation where no reduction occurs, making $h = h'$ again.

Inductive case: For some arbitrary $n > 0$
Inductive hypothesis: Assume the property holds for all programs $\mathbb{P}$ that terminate in n steps.

$$\forall h, h', \mathbb{P}, \mathbb{P}_2.\, (h, \mathbb{P}) \rightarrow^n_{\text{Atom}} (h', \texttt{skip}) \implies (h, \mathbb{P}; \mathbb{P}_2) \rightarrow^*_{\text{Atom}} (h', \texttt{skip}; \mathbb{P}_2)$$

To show:

$$\forall h, h', \mathbb{P}_1, \mathbb{P}_2.\, (h, \mathbb{P}_1) \rightarrow^{n+1}_{\text{Atom}} (h', \texttt{skip}) \implies (h, \mathbb{P}_1; \mathbb{P}_2) \rightarrow^*_{\text{Atom}} (h', \texttt{skip}; \mathbb{P}_2)$$

We assume $(h, \mathbb{P}_1) \rightarrow^{n+1}_{\text{Atom}} (h', \texttt{skip})$ holds which implies that there exists some program $\mathbb{P}_n$ and storage h_n such that $(h, \mathbb{P}_1) \rightarrow_{\text{Atom}} (h_n, \mathbb{P}_n)$ and $(h_n, \mathbb{P}_n) \rightarrow^n_{\text{Atom}} (h', \texttt{skip})$. The former is the premiss of rule PSeq for the conclusion $(h, \mathbb{P}_1; \mathbb{P}_2) \rightarrow_{\text{Atom}} (h_n, \mathbb{P}_n; \mathbb{P}_2)$ in one step of reduction, while the latter gives us $(h_n, \mathbb{P}_n; \mathbb{P}_2) \rightarrow^*_{\text{Atom}} (h', \texttt{skip}; \mathbb{P}_2)$ by inductive hypothesis. From the two combined we conclude that the property holds for $n + 1$ steps. □

Lemma C.7.

$$\forall h, h', S, S', \Phi, \mathbb{P}_1, \mathbb{P}_2.$$
$$(h, \Phi, S, \mathbb{P}_1; \mathbb{P}_2) \to^* (h', \emptyset, S', \texttt{skip}; \mathbb{P}_2) \implies (h, \Phi, S, \mathbb{P}_1) \to^* (h', \emptyset, S', \texttt{skip})$$

Proof. The proof is done by induction on n, i.e. the number of reduction steps in $\to^*$.

Base case: $n = 0$
To show:

$$\forall h, h', S, S', \Phi, \mathbb{P}_1, \mathbb{P}_2.$$
$$(h, \Phi, S, \mathbb{P}_1; \mathbb{P}_2) \to^0 (h', \emptyset, S', \texttt{skip}; \mathbb{P}_2) \implies (h, \Phi, S, \mathbb{P}_1) \to^* (h', \emptyset, S', \texttt{skip})$$

We assume that $(h, \Phi, S, \mathbb{P}_1; \mathbb{P}_2) \to^0 (h', \emptyset, S', \texttt{skip}; \mathbb{P}_2)$ holds. A zero-step reduction means that it must be the case that $\mathbb{P}_1 = \texttt{skip}$. Therefore $(h, \Phi, S, \texttt{skip}; \mathbb{P}_2) \to^0 (h', \emptyset, S', \texttt{skip}; \mathbb{P}_2)$ for $h = h', \Phi = \emptyset, S = S'$.

Inductive case: For some arbitrary $n > 0$
Inductive hypothesis: Assume the property holds for all programs $\mathbb{P}; \mathbb{P}_2$ that reduce to skip in n steps.

$$\forall h, h', S, S', \Phi, \mathbb{P}, \mathbb{P}_2.$$
$$(h, \Phi, S, \mathbb{P}; \mathbb{P}_2) \to^n (h', \emptyset, S', \texttt{skip}; \mathbb{P}_2) \implies (h, \Phi, S, \mathbb{P}) \to^* (h', \emptyset, S', \texttt{skip})$$

To show:

$$\forall h, h', S, S', \Phi, \mathbb{P}_1, \mathbb{P}_2.$$
$$(h, \Phi, S, \mathbb{P}_1; \mathbb{P}_2) \to^{n+1} (h', \emptyset, S', \texttt{skip}; \mathbb{P}_2) \implies (h, \Phi, S, \mathbb{P}_1) \to^* (h', \emptyset, S', \texttt{skip})$$

We assume that $(h, \Phi, S, \mathbb{P}_1; \mathbb{P}_2) \to^{n+1} (h', \emptyset, S', \texttt{skip}; \mathbb{P}_2)$ holds which means there must be a program $\mathbb{P}_n$ and state h_n, Φ_n, S_n such that $(h, \Phi, S, \mathbb{P}_1; \mathbb{P}_2) \xrightarrow{\alpha} (h_n, \Phi_n, S_n, \mathbb{P}_n; \mathbb{P}_2)$ and $(h_n, \Phi_n, S_n, \mathbb{P}_n; \mathbb{P}_2) \to^n (h', \emptyset, S', \texttt{skip}; \mathbb{P}_2)$. From the former and rule PSeq we obtain that $(h, \Phi, S, \mathbb{P}_1) \xrightarrow{\alpha} (h_n, \Phi_n, S_n, \mathbb{P}_n)$ and from the latter we get that $(h_n, \Phi_n, S_n, \mathbb{P}_n) \to^* (h', \emptyset, S', \texttt{skip})$ by inductive hypothesis. From the two combined we conclude that the property holds for $n+1$ steps. □

Lemma C.8.

$$\forall h, h', S, S', \Phi', \mathbb{P}. (h, \emptyset, S, \mathbb{P}) \to^* (h', \Phi', S', \texttt{skip}) \implies \Phi' = \emptyset$$

Proof. Let's pick arbitrary $h, h' \in \mathsf{Storage}, S, S' \in \mathsf{TState}, \Phi' \in \mathsf{LMan}, \mathbb{P} \in \mathsf{Prog}$ and assume that the following holds:

$$(h, \emptyset, S, \mathbb{P}) \to^* (h', \Phi', S', \texttt{skip}) \tag{117}$$

We now also assume that the final lock manager, Φ', is empty, meaning that it all of keys in its domain are associated to locks in the unlocked mode.

$$\Phi' \neq \emptyset \tag{118}$$

From (117) and (118) we know that in the full reduction that happens as described in (117) there must be at least an action α which modifies the lock manager and gives lock ownership to a particular transaction:

$$(h, \emptyset, S, \mathbb{P}) \to^* (h_l, \Phi_l, S_l, \mathbb{P}_l) \xrightarrow{\alpha} (h'_l, \Phi'_l, S'_l, \mathbb{P}'_l) \to^* (h', \Phi', S', \texttt{skip}) \tag{119}$$

- If $\alpha = \mathsf{lock}(\iota, k, \kappa)$ then from (119) and the semantic interpretation of α it must be the case that $\Phi'_l(k) = (\{\iota\} \uplus I, \kappa)$ for $(I, \kappa') = \Phi_l(k), \kappa' \leq \mathrm{s}$.
- If $\alpha = \mathsf{alloc}(\iota, n, l)$ then (119) and the semantic interpretation of α it must be the case that $\Phi'_l(l) = \ldots = \Phi'(l + n - 1) = (\{\iota\}, \mathrm{x})$.

In both cases $\Phi'_l \neq \emptyset$. From (118) we obtain that there must be at least one cell in the domain of Φ' such that its associated lock is not unlocked and therefore owned by a transaction. It follows that for one of the α in (119), there is no matching $\mathsf{unlock}(\iota, k)$ (where in the case of $\alpha = \mathsf{alloc}(\iota, n, l)$ we have $l \leq k < l + n$) operation done by the same transaction to release the lock it holds. Formally this means that:

$$k \in \mathsf{dom}(\Phi') \wedge \iota \in \left(\Phi'(k) \downarrow_1\right) \tag{120}$$

From (117) we know that the whole program $\mathbb{P}$ reduced to `skip` meaning that the transaction $\mathbb{T}_\iota$ must have also reduced to `skip` in the same reduction. The only way for a transaction to do so, is through the PSkip in a reduction of the following shape:

$$(h'_l, \Phi'_l, S'_l, \mathbb{P}'_l) \rightarrow^* (h_e, \Phi_e, S_e, \mathbb{P}_e) \xrightarrow{\mathsf{id}(\iota)} (h_e, \Phi_e, S_e, \mathbb{P}'_e) \rightarrow^* (h', \Phi', S', \mathtt{skip}) \tag{121}$$

From (120) we know that $\iota \in \Phi_e(k) \downarrow_1$ since there is no unlock action to release ι's lock on k. The PSkip rule's premiss instead requires that:

$$\forall k.\, k \in \mathsf{dom}(\Phi_e) \wedge \iota \notin (\Phi_e(k) \downarrow_1) \tag{122}$$

From (121) and (122) we obtain a contradiction, therefore we can conclude that $\Phi' = \emptyset$ as needed. ☐

Lemma C.9.

$$\forall h, \Phi, S, \mathbb{P}, \mathbb{P}'.\, (h, \Phi, S, \mathbb{P}) \xrightarrow{\mathsf{sys}} (h', \Phi', S', \mathbb{P}') \implies$$
$$(h, \mathbb{P}) \rightarrow_{\mathrm{ATOM}} (h', \mathbb{P}') \wedge h' = h \wedge \Phi' = \Phi \wedge S' = S$$

Proof. Let's pick arbitrary $h \in \mathsf{Storage}, \Phi \in \mathsf{LMan}, S \in \mathsf{TState}, \mathbb{P}, \mathbb{P}' \in \mathsf{Prog}$. We now assume that the following holds:

$$(h, \Phi, S, \mathbb{P}) \xrightarrow{\mathsf{sys}} (h, \Phi, S, \mathbb{P}') \tag{123}$$

Given that from (123) we know that $\mathbb{P}$ reduced to $\mathbb{P}'$ through sys we will proceed with a case-by-case analysis on the structure of $\mathbb{P}$.

- If $\mathbb{P} = \mathtt{skip}; \mathbb{P}''$ for some $\mathbb{P}'' \in \mathsf{Prog}$, then from the PSeqSkip rule we get that $\mathbb{P}' = \mathbb{P}'', h' = h, \Phi' = \Phi, S' = S$. From the AtPSeqSkip rule we know that as part of the atomic semantics, $(h, \mathbb{P})$ will also reduce to $(h, \mathbb{P}'')$.
- If $\mathbb{P} = \mathtt{skip} \| \mathtt{skip}$ then from the ParEnd rule we get that $\mathbb{P}' = \mathtt{skip}, h' = h, \Phi' = \Phi, S' = S$. From the AtParEnd rule we know that as part of the atomic semantics, $(h, \mathbb{P})$ will also reduce to $(h, \mathtt{skip})$.
- If $\mathbb{P} = \mathbb{P}_0^*$ for some $\mathbb{P}_0 \in \mathsf{Prog}$, then from the Loop rule we get that $\mathbb{P}' = \mathtt{skip} + (\mathbb{P}_0; \mathbb{P}_0^*), h' = h, \Phi' = \Phi, S' = S$. From the AtLoop rule we know that as part of the atomic semantics, $(h, \mathbb{P})$ will also reduce to $(h, \mathtt{skip} + (\mathbb{P}_0; \mathbb{P}_0^*))$.
- $\mathbb{P} = \mathbb{P}_1 + \mathbb{P}_2$ for some $\mathbb{P}_1, \mathbb{P}_2 \in \mathsf{Prog}$, then we can apply one of two reduction rules:
  - If sys was generated by the ChoiceL rule then $\mathbb{P}' = \mathbb{P}_1, h' = h, \Phi' = \Phi, S' = S$ and through the AtChoiceL rule $(h, \mathbb{P})$ will also reduce to $(h, \mathbb{P}_1)$.

– If sys was generated by the ChoiceR rule then $\mathbb{P}' = \mathbb{P}_2, h' = h, \Phi' = \Phi, S' = S$ and through the AtChoiceR rule $(h, \mathbb{P})$ will also reduce to $(h, \mathbb{P}_2)$.

□

Lemma C.10. Any system action label followed by a transaction label, $\alpha(\iota)$, can be swapped as long as α does not come from $\mathbb{T}_\iota$'s first reduction.

$$\forall h, \underline{h}, \Phi, S, \mathbb{P}, x, y, n, \tau, \tau', \alpha, \iota. \tag{124}$$

$$\mathsf{tgen}(\tau, h, \underline{h}, \Phi, S, \mathbb{P}) \land \alpha = \alpha(\iota) \land x = (\mathsf{sys}, n) \land y = (\alpha, n+1) \land x \in \tau \land y \in \tau \land \tag{125}$$

$$\exists \alpha' = \alpha(\iota).\, \tau \vDash \alpha' < x \land \tau' = \mathsf{swap}(\tau, x, y) \implies \mathsf{tgen}(\tau', h, \underline{h}, \Phi, S, \mathbb{P}) \tag{126}$$

Proof. Let's pick arbitrary $h, \underline{h} \in \mathsf{Storage}, \Phi \in \mathsf{LMan}, S \in \mathsf{TState}, \mathbb{P} \in \mathsf{Prog}, x, y \in \mathsf{Act} \times \mathbb{N}, n \in \mathbb{N}, \tau, \tau' \in [\mathsf{Act} \times \mathbb{P}], \iota \in \mathsf{Tid}$ and assume that the following holds:

$$\mathsf{tgen}(\tau, h, \underline{h}, \Phi, S, \mathbb{P}) \land \alpha = \alpha(\iota) \land x = (\mathsf{sys}, n) \land y = (\alpha, n+1) \land x \in \tau \land y \in \tau \land \tag{127}$$

$$\exists \alpha' = \alpha(\iota).\, \tau \vDash \alpha' < x \land \tau' = \mathsf{swap}(\tau, x, y) \tag{128}$$

The above means that τ generates $\underline{h}$ starting from $h, \Phi, S, \mathbb{P}$ and as part of its operations, it contains a system transition, $x = (\mathsf{sys}, n)$, immediately followed by an operation, $y = (\alpha, n+1)$, performed by transaction ι. Also, α is not the first reduction of $\mathbb{T}_\iota$ since by assumption there exists another action $\alpha' = \alpha(\iota)$ which happens before x in τ, i.e. $\tau \vDash \alpha' < x$ holds. We now assume the existence of another trace, τ', which is equivalent to τ with x and y swapped. We are now required to show that $\mathsf{tgen}(\tau', h, \underline{h}, \Phi, S, \mathbb{P})$ holds.

From the definition tgen and the semantic interpretation of sys we know that the following must hold:

$$\begin{aligned}(h, \Phi, S, \mathbb{P}) \to^* (h_1, \Phi_1, S_1, \mathbb{P}_1) \xrightarrow{\mathsf{sys}} (h_1, \Phi_1, S_1, \mathbb{P}_1') \xrightarrow{\alpha} (h_2, \Phi_2, S_2, \mathbb{P}_2) \\ \to^* (\underline{h}, \emptyset, S', \mathtt{skip})\end{aligned} \tag{129}$$

Given that from our assumption, α is not ι's starting action then from (129) without loss of generality we can assume that $\mathbb{P}_1$ is of the following shape:

$$(\mathbb{T}_\iota; \mathbb{P}_a) \parallel \mathbb{P}_b \tag{130}$$

meaning that the sys transition happened as part of $\mathbb{P}_b$. From (129) and (130) we know that we can always find a program $\mathbb{P}_1''$ with the shape $(\mathbb{T}_\iota'; \mathbb{P}_a) \parallel \mathbb{P}_b$, such that:

$$\begin{aligned}(h, \Phi, S, \mathbb{P}) \to^* (h_1, \Phi_1, S_1, \mathbb{P}_1) \xrightarrow{\alpha} (h_2, \Phi_2, S_2, \mathbb{P}_1'') \xrightarrow{\mathsf{sys}} (h_2, \Phi_2, S_2, \mathbb{P}_2) \\ \to^* (\underline{h}, \emptyset, S', \mathtt{skip})\end{aligned} \tag{131}$$

From (131) and the definition of tgen we can conclude that $\mathsf{tgen}(h, \underline{h}, \Phi, S, \mathbb{P})$ holds. □

Lemma C.11.

$$\begin{gathered}\forall h, h', \mathbb{T}.\, (h, \mathbb{T}) \to^*_{\text{Atom}} (h', \mathtt{skip}) \implies \\ \left(\exists \sigma, \sigma'.\, \mathbb{T}, \sigma \to^*_{\mathsf{Views}} \mathtt{skip}, \sigma' \land h = \sigma \downarrow_1 \land h' = \sigma' \downarrow_1\right)\end{gathered}$$

Proof. We prove the statement by induction on the structure of $\mathbb{T} \in \mathsf{Trans}$.

Base case 1: $\mathtt{begin}\ \mathbb{C}\ \mathtt{end} \in \mathsf{Trans}$
To show:

$$\begin{gathered}\forall h, h', \mathbb{C}.\, (h, \mathtt{begin}\ \mathbb{C}\ \mathtt{end}) \to^*_{\text{Atom}} (h', \mathtt{skip}) \implies \\ \left(\exists \sigma, \sigma'.\, \mathtt{begin}\ \mathbb{C}\ \mathtt{end}, \sigma \xrightarrow{\mathtt{begin}\ \mathbb{C}\ \mathtt{end}}{}^*_{\mathsf{Views}}\ \mathtt{skip}, \sigma' \land h = \sigma \downarrow_1 \land h' = \sigma' \downarrow_1\right)\end{gathered}$$

For arbitrary $h, h', \mathbb{C}$ let's assume that $(h, \texttt{begin } \mathbb{C} \texttt{ end}) \rightarrow^*_{\text{ATOM}} (h', \texttt{skip})$ holds. From rule AtTrans we obtain that the following holds:

$$\exists s'. (h, \emptyset, \mathbb{C}) \rightarrow^*_{\text{ATOM}} (h', s', \texttt{skip}) \tag{132}$$

From (132) and Lemma C.12 we get that:

$$(h', s') \in [\![\mathbb{C}]\!]_{\mathsf{C}}(h, \emptyset) \tag{133}$$

The statement in (133) is the premiss from which we can conclude that:

$$\texttt{begin } \mathbb{C} \texttt{ end}, (h, \emptyset) \xrightarrow{\texttt{begin } \mathbb{C} \texttt{ end}}{}^*_{\mathsf{Views}} \texttt{skip}, (h', s')$$

must hold. It follows that we found $\sigma = (h, \emptyset)$ and $\sigma' = (h', s')$, which concludes our proof.

Base case 2: $\texttt{begin } \mathbb{C} \texttt{ end}_\iota \in \mathsf{Trans}$
System transactions are not reduced by the Atom semantics given that there is no rule capable of doing so. It follows that we can skip this case since $(h, \texttt{begin } \mathbb{C} \texttt{ end}_\iota) \rightarrow^*_{\text{ATOM}} (h', \texttt{skip})$ will never hold. □

Lemma C.12.

$$\forall h, h', s, s', \mathbb{C}. (h, s, \mathbb{C}) \rightarrow^*_{\text{ATOM}} (h', s', \texttt{skip}) \implies (h', s') \in [\![\mathbb{C}]\!]_{\mathsf{C}}(h, s)$$

Proof. We prove the statement by induction on the structure of $\mathbb{C} \in \mathsf{Cmd}$.

Base Case 1: $\texttt{skip} \in \mathsf{Cmd}$
For arbitrary h, h', s, s' let's assume that $(h, s, \texttt{skip}) \rightarrow^*_{\text{ATOM}} (h', s', \texttt{skip})$ holds. We must have a zero-step reduction which makes $h' = h$ and $s' = s$. Now, from the definition of $[\![-]\!]_{\mathsf{C}}$ we obtain that $[\![\texttt{skip}]\!]_{\mathsf{C}}(h, s) = \{(h, s)\}$. It follows that $(h', s') \in [\![\texttt{skip}]\!]_{\mathsf{C}}(h, s)$.

Base Case 2: $\hat{\mathbb{C}} \in \mathsf{Cmd}$ The required result follows directly from Lemma C.13.

Inductive Case 1: $\mathbb{C}_1; \mathbb{C}_2 \in \mathsf{Cmd}$
Inductive hypothesis:

$$\begin{gathered}\forall h, h', s, s'. (h, s, \mathbb{C}_1) \rightarrow^*_{\text{ATOM}} (h', s', \texttt{skip}) \implies (h', s') \in [\![\mathbb{C}_1]\!]_{\mathsf{C}}(h, s) \\ \wedge \\ \forall h, h', s, s'. (h, s, \mathbb{C}_2) \rightarrow^*_{\text{ATOM}} (h', s', \texttt{skip}) \implies (h', s') \in [\![\mathbb{C}_2]\!]_{\mathsf{C}}(h, s)\end{gathered}$$

For arbitrary h, h', s, s' let's assume that $(h, s, \mathbb{C}_1; \mathbb{C}_2) \rightarrow^*_{\text{ATOM}} (h', s', \texttt{skip})$ holds. As a consequence we know that from the repeated use of rule AtSeq, followed by rule AtSkip we obtain this overall reduction:

$$\exists h'', s''. (h, s, \mathbb{C}_1; \mathbb{C}_2) \rightarrow^*_{\text{ATOM}} (h'', s'', \texttt{skip}; \mathbb{C}_2) \tag{134}$$

$$\rightarrow_{\text{ATOM}} (h'', s'', \mathbb{C}_2) \rightarrow^*_{\text{ATOM}} (h', s', \texttt{skip}) \tag{135}$$

From (134) we know that $(h, s, \mathbb{C}_1) \rightarrow^*_{\text{ATOM}} (h'', s'', \texttt{skip})$ and from the reduction in (135) that $(h'', s'', \mathbb{C}_2) \rightarrow^*_{\text{ATOM}} (h', s', \texttt{skip})$. From the I.H. on both $\mathbb{C}_1$ and on $\mathbb{C}_2$ we obtain that:

$$(h'', s'') \in [\![\mathbb{C}_1]\!]_{\mathsf{C}}(h, s) \ \wedge \ (h', s') \in [\![\mathbb{C}_2]\!]_{\mathsf{C}}(h'', s'') \tag{136}$$

Now, from (136) and the definition of $[\![-]\!]_{\mathsf{C}}$ it must be the case that $(h', s') \in [\![\mathbb{C}_1; \mathbb{C}_2]\!]_{\mathsf{C}}(h, s)$ which concludes the proof.

Inductive Case 2: $\texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2 \in \mathsf{Cmd}$
Inductive hypothesis:

$$\forall h, h', s, s'.\,(h, s, \mathbb{C}_1) \to^*_{\text{ATOM}} (h', s', \texttt{skip}) \implies (h', s') \in [\![\mathbb{C}_1]\!]_{\mathsf{C}}(h, s)$$
$$\wedge$$
$$\forall h, h', s, s'.\,(h, s, \mathbb{C}_2) \to^*_{\text{ATOM}} (h', s', \texttt{skip}) \implies (h', s') \in [\![\mathbb{C}_2]\!]_{\mathsf{C}}(h, s)$$

For arbitrary h, h', s, s' let's assume that $(h, s, \texttt{if}\,(\mathbb{B})\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2) \to^*_{\text{ATOM}} (h', s', \texttt{skip})$ holds. There are two cases to consider based on how $b = [\![\mathbb{B}]\!]^{\mathrm{B}}_{(h,s)}$ evaluates.

- If $b = \top$ then from rule AtCondT we reduce:

$$(h, s, \texttt{if}\,(\top)\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2) \to^*_{\text{ATOM}} (h, s, \mathbb{C}_1)$$

  From the definition of $[\![-]\!]_{\mathsf{C}}$ we obtain that:

$$[\![\texttt{if}\,(\top)\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2]\!]_{\mathsf{C}}(h, s) = [\![\mathbb{C}_1]\!]_{\mathsf{C}}(h, s)$$

  From I.H. on $\mathbb{C}_1$ we know that $(h', s') \in [\![\mathbb{C}_1]\!]_{\mathsf{C}}(h, s)$ and we can conclude that:

$$(h', s') \in [\![\texttt{if}\,(\top)\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2]\!]_{\mathsf{C}}(h, s)$$

- If $b = \bot$ then from rule AtCondT we reduce:

$$(h, s, \texttt{if}\,(\bot)\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2) \to^*_{\text{ATOM}} (h, s, \mathbb{C}_2)$$

  From the definition of $[\![-]\!]_{\mathsf{C}}$ we obtain that:

$$[\![\texttt{if}\,(\bot)\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2]\!]_{\mathsf{C}}(h, s) = [\![\mathbb{C}_2]\!]_{\mathsf{C}}(h, s)$$

  From I.H. on $\mathbb{C}_2$ we know that $(h', s') \in [\![\mathbb{C}_2]\!]_{\mathsf{C}}(h, s)$ and we can conclude that:

$$(h', s') \in [\![\texttt{if}\,(\bot)\ \mathbb{C}_1\ \texttt{else}\ \mathbb{C}_2]\!]_{\mathsf{C}}(h, s)$$

Inductive Case 3: $\texttt{while}\,(\mathbb{B})\ \mathbb{C} \in \mathsf{Cmd}$
Inductive hypothesis:

$$\forall h, h', s, s'.\,(h, s, \mathbb{C}) \to^*_{\text{ATOM}} (h', s', \texttt{skip}) \implies (h', s') \in [\![\mathbb{C}]\!]_{\mathsf{C}}(h, s)$$

We will prove this case by mathematical induction on n, the number of $\to^*_{\text{ATOM}}$ reduction steps.
Base case 3.1: $n = 1$
For arbitrary h, h', s, s' let's assume that $(h, s, \texttt{while}\,(\mathbb{B})\ \mathbb{C}) \to_{\text{ATOM}} (h', s', \texttt{skip})$ holds. Given that $n = 1$ we know that the loop terminated immediately therefore $b = [\![\mathbb{B}]\!]^{\mathrm{B}}_{s} = \bot$. From rule AtLoopF we obtain that $h' = h$ and $s' = s$ and from the definition of $[\![-]\!]_{\mathsf{C}}$ we get that:

$$[\![\texttt{while}\,(\bot)\ \mathbb{C}]\!]_{\mathsf{C}}(h, s) = [\![\texttt{skip}]\!]_{\mathsf{C}}(h, s) = \{(h, s)\}$$

which implies that $(h', s') \in [\![\texttt{while}\,(\bot)\ \mathbb{C}]\!]_{\mathsf{C}}(h, s)$.

Inductive case: $n > 1$
Inductive hypothesis:

$$\forall m \leq n, h, h', s, s'.\,(h, s, \texttt{while}\,(\mathbb{B})\ \mathbb{C}) \to^m_{\text{ATOM}} (h', s', \texttt{skip})$$
$$\implies (h', s') \in [\![\texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}(h, s)$$

To show:

$$\forall h, h', s, s'. (h, s, \texttt{while}\,(\mathbb{B})\ \mathbb{C}) \rightarrow^{n+1}_{\text{ATOM}} (h', s', \texttt{skip}) \implies (h', s') \in [\![\texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}(h, s)$$

For arbitrary h, h', s, s' let's assume that $(h, s, \texttt{while}\,(\mathbb{B})\ \mathbb{C}) \rightarrow^{n+1}_{\text{ATOM}} (h', s', \texttt{skip})$ holds. Given that $n > 1$ we know that the loop has unrolled at least once, making $b = [\![\mathbb{B}]\!]^{\text{B}}_{s} = \top$ and from rule AtLoopT we reduce as follows:

$$(h, s, \texttt{while}\,(\top)\ \mathbb{C}) \rightarrow_{\text{ATOM}} (h, s, \mathbb{C}; \texttt{while}\,(\mathbb{B})\ \mathbb{C})$$

At this point we can proceed in a similar way to the sequential (;) case of this proof. We apply rule AtSeq until $\mathbb{C}$ hits skip:

$$(h, s, \mathbb{C}; \texttt{while}\,(\mathbb{B})\ \mathbb{C}) \rightarrow^{*}_{\text{ATOM}} (h'', s'', \texttt{skip}; \texttt{while}\,(\mathbb{B})\ \mathbb{C})$$

It follows that $(h, s, \mathbb{C}) \rightarrow^{*}_{\text{ATOM}} (h'', s'', \texttt{skip})$ and by the top-level I.H. on $\mathbb{C}$ we obtain that $(h'', s'') \in [\![\mathbb{C}]\!]_{\mathsf{C}}(h, s)$. Now, from the definition of $[\![-]\!]_{\mathsf{C}}$ we know that:

$$[\![\texttt{while}\,(\top)\ \mathbb{C}]\!]_{\mathsf{C}}(h, s) = [\![\mathbb{C}; \texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}(h, s) \tag{137}$$

$$= \{(\dot{h}, \dot{s}) \mid S = [\![\mathbb{C}]\!]_{\mathsf{C}}(h, s) \wedge (\dot{h}, \dot{s}) \in [\![\texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}(S)\} \tag{138}$$

Since we know that $(h'', s'') \in [\![\mathbb{C}]\!]_{\mathsf{C}}(h, s)$, we get that from (138) and the I.H. on the number of steps of reduction, it must be the case that one of the $(\dot{h}, \dot{s})$ that are part of the set $[\![\texttt{while}\,(\mathbb{B})\ \mathbb{C}]\!]_{\mathsf{C}}(h'', s'')$ is equivalent to (h', s'), which concludes our proof. □

Lemma C.13.

$$\forall h, h', s, s', \hat{\mathbb{C}}. (h, s, \hat{\mathbb{C}}) \rightarrow^{*}_{\text{ATOM}} (h', s', \texttt{skip}) \implies (h', s') \in [\![\hat{\mathbb{C}}]\!]_{\hat{\mathsf{C}}}(h, s)$$

Proof. We prove the statement by doing a case-by-case analysis on the structure of $\hat{\mathbb{C}} \in \mathsf{ECmd}$.

Case 1: $\texttt{x} := \mathbb{E} \in \mathsf{ECmd}$
For arbitrary h, h', s, s' let's assume that $(h, s, \texttt{x} := \mathbb{E}) \rightarrow^{*}_{\text{ATOM}} (h', s', \texttt{skip})$ holds. From rule AtAssign we know that $h' = h$ and $s' = s[\texttt{x} \mapsto v]$ where $v = [\![\mathbb{E}]\!]^{\text{E}}_{s}$. From the definition of $[\![-]\!]_{\hat{\mathsf{C}}}$ we obtain that $[\![\texttt{x} := \mathbb{E}]\!]_{\hat{\mathsf{C}}} = \{(h, s' = s[\texttt{x} \mapsto v])\}$ which implies that $(h', s') \in [\![\texttt{x} := \mathbb{E}]\!]_{\hat{\mathsf{C}}}$.

Case 2: $\texttt{x} := \texttt{alloc}(\mathbb{E}) \in \mathsf{ECmd}$
For arbitrary h, h', s, s' let's assume that $(h, s, \texttt{x} := \texttt{alloc}(\mathbb{E})) \rightarrow^{*}_{\text{ATOM}} (h', s', \texttt{skip})$ holds. From rule AtAlloc we know that:

$$h' = h[l \mapsto 0] \ldots [l + n - 1 \mapsto 0] \wedge s' = s[\texttt{x} \mapsto l]$$

where $n = [\![\mathbb{E}]\!]^{\text{E}}_{s}$, $n > 0$ and $\{l, \ldots, l + n - 1\} \cap \mathsf{dom}(h) \equiv \emptyset$. Now, from the definition of $[\![-]\!]_{\hat{\mathsf{C}}}$ we obtain that:

$$[\![\texttt{x} := \texttt{alloc}(\mathbb{E})]\!]_{\hat{\mathsf{C}}} = \{(h[\vec{a} \mapsto 0], s' = s[\texttt{x} \mapsto l])\} \text{ for } \vec{a} = (l, \ldots, l + n - 1)$$

which implies that $(h', s') \in [\![\texttt{x} := \texttt{alloc}(\mathbb{E})]\!]_{\hat{\mathsf{C}}}$.

Case 3: $\texttt{x} := [\mathbb{E}] \in \mathsf{ECmd}$
For arbitrary h, h', s, s' let's assume that $(h, s, \texttt{x} := [\mathbb{E}]) \rightarrow^{*}_{\text{ATOM}} (h', s', \texttt{skip})$ holds. From rule AtRead we know that:

$$h' = h \wedge s' = s[\texttt{x} \mapsto v]$$

where $k = [\![\mathbb{E}]\!]^{\text{E}}_{s}, k \in \mathsf{dom}(h)$ and $h(k) = v$. Now from the definition of $[\![-]\!]_{\hat{\mathsf{C}}}$ we obtain that:

$$[\![\texttt{x} := [\mathbb{E}]]\!]_{\hat{\mathsf{C}}} = \{(h, s[\texttt{x} \mapsto v])\}$$

given that $k = [\![\mathbb{E}]\!]^{\text{E}}_{s}, k \in \mathsf{dom}(h)$ and $h(k) = v$. This implies that $(h', s') \in [\![\mathtt{x} := [\mathbb{E}]]\!]_{\hat{\mathsf{C}}}$.

Case 4: $[\mathbb{E}_1] := \mathbb{E}_2 \in \mathsf{ECmd}$
For arbitrary h, h', s, s' let's assume that $(h, s, [\mathbb{E}_1] := \mathbb{E}_2) \rightarrow^{*}_{\text{ATOM}} (h', s', \mathtt{skip})$ holds. From rule AtWrite we know that:

$$h' = h[k \mapsto v] \wedge s' = s$$

where $k = [\![\mathbb{E}_1]\!]^{\text{E}}_{s}, k \in \mathsf{dom}(h)$ and $v = [\![\mathbb{E}_2]\!]^{\text{E}}_{s}$. Now from the definition of $[\![-]\!]_{\hat{\mathsf{C}}}$ we obtain that:

$$[\![[\mathbb{E}_1] := \mathbb{E}_2]\!]_{\hat{\mathsf{C}}} = \{(h[k \mapsto v], s)\}$$

given that $k = [\![\mathbb{E}_1]\!]^{\text{E}}_{s}, k \in \mathsf{dom}(h)$ and $v = [\![\mathbb{E}_2]\!]^{\text{E}}_{s}$. This implies that $(h', s') \in [\![[\mathbb{E}_1] := \mathbb{E}_2]\!]_{\hat{\mathsf{C}}}$. $\square$