Weak Consistency in Transactional Systems: a Multi-version Based Operational Approach

Contents of this set of notes: History heaps. Semantics of Programs running under weak consistency models using history heaps as states. Simulation technique for comparing weak consistency models defined using history heaps. Verification of implementations. **Points following Dagstuhl: Viktor seemed positive about the history heap work. His question was whether the framework is generic enough to capture the protocols that they are developing with Azalea. Alexey's opinion is that the framework may have some use if we manage to prove implementations of protocols correct. I would also like to have Azalea's opinion on a semantics based on history heaps.**

Additional Key Words and Phrases: keyword1, keyword2, keyword3

1 INTRODUCTION

AC: Obviously this will be rewritten again and again, but it should give an idea of how the paper is going to be structured.

Modern distributed systems s often rely on databases that achieve scalability by weakening consistency guarantees of distributed transaction processing. These databases are said to implement weak consistency models. Such weakly consistent databases allow for faster transaction processing, but exhibit anomalous behaviours, which do not arise under a database with a strong consistency guarantee, such as *serialisability*.

Recently, the research community has made an effort to give formal specifications of weak consistency models for transactional distributed databases [Burckhardt et al. 2012; Cerone et al. 2015, 2017; Crooks et al. 2017; Kaki et al. 2017; Shapiro et al. 2016]. However, the problem of giving the semantics describing the operational behaviour of clients interacting with a weakly consistent databases, has been largely neglected. The only work that we are aware of in this field is given by [Kaki et al. 2017]. There the authors study the behaviour of functional programs interacting with a relational database, and develop a program logics for proving invariants of programs executed by clients of the database.

AC: Insert sentence to swiftly kill their paper without being an asshole here.

In this paper we focus on the semantics of clients of distributed key-value stores which provide weak consistency guarantees. We propose a *coarse-grained* approach to evaluating transactions, where transactions are executed by clients in a single, atomic step; furthermore, transactions of concurrent clients are executed in an interleaving fashion. This is In contrast with the work of [Kaki et al. 2017], which takes a fine-grained approach in which transactions are evaluated one step at the time on a local copy of the database, and thus it is possible to interleave the execution of transaction; we find this complication to be unnecessary, at least in a setting where transactions enjoy *atomic visibility*: either none or all the updates of a transaction are made visible to another transaction.

We take a multi-version approach to model the state of a key-value store. Each key is mapped to a set of versions. A Version consists of a value and the meta-data of the transactions that wrote and read the version. At any given instant of time, concurrent clients can observe different versions for the same key. This approach is necessary to capture the non-serialisable behaviour of programs, while still retaining interleaving concurrency and atomic reduction of transactions.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

Because we want to model different consistency models, in our semantics transactions are executed only prior to passing a particular execution test. For example, to constrain a program to only exhibit serialisable behaviours, we require that transactions can be executed by a client only if it observes the most recent available version for each key. By tweaking the execution test of transactions, we tweak the consistency model under which programs are executed. In 5 we give examples of execution tests that can be used to model all the consistency models formalised in [Cerone et al. 2015]. The notion of execution test is inspired by [Crooks et al. 2017]. There a notion of *commit test* is introduced to determine whether a transaction can be executed safely. However, the notion of commit test requires the complete knowledge of how the system of the key-value store evolved from its initial state (i.e. it requires knowing the total order in which transactions executed in the computation); execution tests, on the other hand, only requires knowing the list of versions stored by each key, and the information about which version of each key is observed by the client executing the transaction.

AC: Paragraph about the thoroughness of the semantics and correspondence of our specifications and the declarative ones from the CONCUR'15 paper. Paragraph about the logic. Here we should stress that clients that execute correctly under serialisability, are not necessarily correct under a weaker consistency model. There should be a sentence explaining that thoroughness of the semantics is necessary to obtain soundness of the logic.

Contributions of the paper:

(1) An operational semantics of programs interacting with a key-value store,
(2) Definition of different execution tests for capturing several consistency models, and a proof that the consistency models captured by the execution tests we propose are equivalent to the ones obtained through the respective declarative specification given in[Cerone et al. 2015],
(3) A proof that the operational semantics is *thorough*: For each of the consistency model we propose, and for any program P, our semantics captures all the behaviours that P can display under said consistency model,
(4) A separation logic based on our semantics to reason about properties of clients interacting with a weakly consistent key-value store.

AC: Got bored of writing the introduction, in any case it will need to be changed a lot in the future.

2 SEMANTICS

SX: program vs transaction

We focus on an abstract computational model where multiple client programs can access and update keys in a key-value store using atomic transactions. Transactions in our model execute atomically, though they have different effect on the key-value stores depending on the *consistency model*. A consistency model controls how the key-value store evolves. A common model is *serialisability*, where transactions appear one after another, yet it is not necessary for many weaker model, which means that, at the moment of executing, a transaction may not observe the most up-to-date values of keys.

To overcome this issue, we first model the state of the system using *multi-version key-value stores (MKVSs)* (Section 2.1). It keeps track of all the versions written for any key, as well as the information about the transactions that read and wrote such versions. To model the potential out-of-date observation, we intrude *views*. A view decides the observable versions of keys for a client. Therefore, when a client want to commit a transaction, it first takes *a snapshot* of the system

with the view, executes internally with respect to the snapshot (Section 2.2.2) and afterwards commits the effect of the transaction if the change is allowed by the consistency model (Section 2.2.3).

2.1 Multi-version Key-value Stores and Views

AC:

SX: Partial is better for logic

Maybe it's better to keep k fixed and say that we look at only a fragment of the key value store. Alternatively, we can go for partial mappings to represent MKVSs, but still avoiding allocation and deallocation of keys.

.

We model the state of system using *multi-version key-value stores (MKVSs)*, is a mapping from keys to lists of versions. Each version $(n, t, \mathcal{T})$ contains a natural number as the value n and a transaction identifier t who writes it and a set of transaction identifiers $\mathcal{T}$ who read it (Def. 2.1). One example of MKVS is given in Fig. 1a (ignore the lines labelled cl_1 and cl_2). There are two keys k_1 and k_2 and each of them has two versions with value 0 and value 1 respectively. We use a box to represent a version where the left is the value and the top right corner is the transaction that wrote the version and bottom right corner is the set of transaction that read such a version. The versions are listed from the left, the earlier version, to right, the later version.

Given a version $\vartheta = (n, t, \mathcal{T})$, let $\mathsf{Value}(\vartheta) = n$, $\mathsf{Write}(\vartheta) = t$, $\mathsf{Reads}(\vartheta) = \mathcal{T}$. Given a list of versions $[\vartheta_0, \cdots, \vartheta_{n-1}]$, let $|\vartheta_0 \cdots \vartheta_{n-1}| = n$ be its length. Given a MKVS κ and a key k, let $\kappa(k)$ be the list of versions associated with the key and $\kappa(k, i)$ denotes the *i-th* version associated with key k. We assume the index starts from 0, so $\kappa(k)(|\kappa(k)| - 1)$ is the latest versions of the key k.

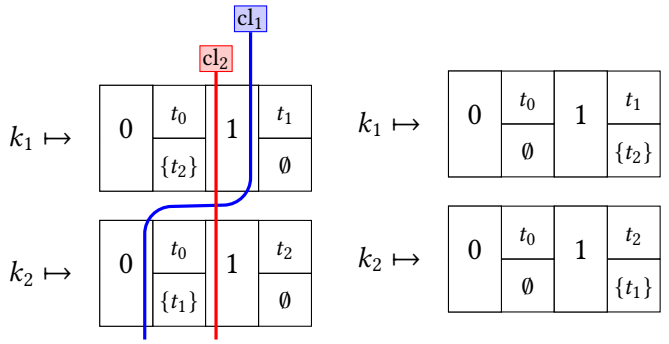
Definition 2.1 (Multi-version key-value stores). Assuming a countably infinite set of keys $\textsc{Keys} = \{k_1, \cdots\}$, *transactions identifiers* $\textsc{TransID} \triangleq \{t_1, \cdots\}$ and natural numbers $n \in \mathbb{N}$, a *multi-version key-value store (MKVS)*, $\kappa \in \mathrm{MKVSs}$, is a partial finite function from keys to lists of *versions*. Each *version* is a triple containing a natural number n, a transaction identifier t and a set of transactions identifiers $\mathcal{T}$:

$$\vartheta \in \textsc{Versions} \triangleq \left\{(n, t, \mathcal{T}) \;\middle|\; n \in \mathbb{N} \wedge t \in \textsc{TransID} \wedge \mathcal{T} \subseteq \textsc{TransID} \wedge t \notin \mathcal{T}\right\}$$
$$\kappa \in \mathrm{MKVSs} \triangleq \textsc{Keys} \xrightarrow{fin} \textsc{Versions}^*$$

The well-formed condition for a key-value store asserts a transaction identifier appears in all the versions for an key at most twice, one as the writer and one as a reader. Also there are no circular dependencies in versions. Given two versions ϑ_1, ϑ_2, the former *direct depends* on later, written $\vartheta_1 \xrightarrow{\mathrm{ddep}} \vartheta_2$, if $\mathsf{Write}(\vartheta_2) \in \mathsf{Reads}(\vartheta_1)$; that is, some transaction t wrote the version ϑ_2 after reading ϑ_1. The $\vartheta_1 \xrightarrow{\mathrm{ddep}(\kappa)} \vartheta_2$ denotes that ϑ_1, ϑ_2 appear as versions from the some KVMS κ and ϑ_1 depends on ϑ_2. Thus, the second well-formedness condition for KVMS is the *transitive relation of direct dependency* $\left(\xrightarrow{\mathrm{ddep}(\kappa)}\right)^+$ is acyclic:

$$\mathtt{wfHH}(\kappa) \triangleq \mathtt{acyclic}\left(\left(\xrightarrow{\mathrm{ddep}(\kappa)}\right)^+\right) \wedge \forall k, t, i, j.\ \begin{pmatrix}\mathsf{Write}(\kappa(k,i)) = \mathsf{Write}(\kappa(k,j)) \vee \\ (t \in \mathsf{Reads}(\kappa(k,i)) \wedge t \in \mathsf{Reads}(\kappa(k,j))\end{pmatrix} \implies i = j$$

The composition of two MSKVs is disjointed union when the domains are disjointed $\kappa \bullet_\kappa \kappa' \triangleq \kappa \uplus \kappa'$ and the unit element is $\mathbf{1}_\kappa \triangleq \emptyset$, which together form *the partial commutative monoid of MSKVs.*



(a) A configuration with a well-formed MSKV and two views

(b) A ill-formed MKVS

Fig. 1. Multi-version key-value stores

AC: A point that this does not ensure a real causal dependency between the two versions, yet it is consistent with the notion of causality employed in databases, should be made

.

SX: Snapshot has not be explained yet

Let us discuss the last well-formedness constraint for a MKVS κ in Def. 2.1, which states that there is no circularity in dependencies relation. This ensures that no versions are created *out of thin-air*. An example of the out of thin-air anomaly is given by Fig. 1b: T transaction t_2 that read the value of k_1 written by t_1; conversely, t_1 read the value of k_2 written by t_2. Because we assume that transactions read a state of the key-value store from an atomic snapshot fixed at the moment they execute, this situation cannot happen. For t_2 to read the version installed by t_1, transaction t_2 must start after t_1, *i.e.* $\kappa_1(k_2, 2) \xrightarrow{\text{ddep}(\kappa_1)} \kappa(k_1, 2)$. Similarly, t_1 must starts after t_2, *i.e.* $\kappa_1(k_1, 2) \xrightarrow{\text{ddep}(hh_1)} \kappa_1(k_2, 2)$. The desired dependent relation violates the well-formedness of MKVSs that $\xrightarrow{\text{ddep}(\kappa_1)}$ is acyclic.

SX: configuration is bad term? as we often use a configuration of the semantics

MKVSs is the overall state of the system, but different *clients* may observe different versions of the same key. To model this, we introduce the notion of *views* and *configurations* (Def. 2.2). A view V defines the particular version of each key that a client will observe when executing a transaction. An example of views is given in Fig. 1a. There are two views, cl_1 in red and cl_2 in blue. Given the figure, formally the view for cl_1 is $u_1 = \{\text{k}_1 \mapsto 1, \text{k}_2 \mapsto 0\}$. This means the client observes the second version of key k_1, carrying value 1 and the first version of key k_2 carrying value 0.

Definition 2.2 (views). *A view* is a partial finite function from keys to indexes,

$$u \in \textsc{Views} \triangleq \textsc{Keys} \xrightarrow{fin} \mathbb{N}$$

The composition is disjointed union $u \bullet_u u' \triangleq u \uplus u'$ and the unit is $\mathbf{1}_u \triangleq \emptyset$. The order between two views with the same domain is defined by the order of the indexes,

$$u \leq u' \overset{\text{def}}{\iff} \text{dom}(u) = \text{dom}(u') \wedge \forall k.\, u(k) \leq u'(k)$$

A configuration consists of a MKVS, and the views associated with clients. An example of configuration is given in Fig. 1a including the key-value store and two clients associated with their own views. For brevity, let $\mathsf{versionOf}(\kappa, k, u) = \kappa(k, u(k))$; we commit an abuse of notation and often write $\mathsf{Value}(\kappa, k, u)$ in lieu of $\mathsf{Value}(\mathsf{versionOf}(\kappa, \mathrm{k}, V))$, and similarly for Write, Reads. If $\vartheta = \mathsf{versionOf}(\kappa, k, u)$, we say that *$u$ k-points to ϑ in κ*. If $\vartheta = \kappa(k, i)$ for some $0 \leq i \leq u(k)$, we say that *u k-includes ϑ in κ*. Last, we always assume that key-value stores, views, and configurations are well-formed, unless otherwise stated.

Definition 2.3 (configurations). A view u is well-formed with respect to a key-value store κ if the have the same domain and every index from the view is within the range of the length of the corresponding version:

$$\mathrm{dom}(\kappa) = \mathrm{dom}(u) \wedge \forall k \in \mathrm{dom}(u).\ 0 \leq u(k) \leq |\kappa(k)|$$

A *configuration* is a pair $C = (\kappa, \mathcal{U})$, where $\mathcal{U} : \textsc{ClientID} \xrightarrow{fin} \textsc{Views}$ is a partial finite function from clients to views. A configuration $C = (\kappa, \mathcal{U})$ is well-formed if for any client any $\mathrm{cl} \in \mathrm{dom}(\mathcal{U})$, the view $\mathcal{U}(t)$ is well-formed with respect to the key-value store κ.

Intuitive, when a client executes a transaction, it extracts *a snapshot* $ss \in \textsc{Snapshot}$ via $\mathsf{snapshot}\ (\kappa, u)$ function which extracts the values corresponding to the versions indexed by the view u (Def. 3.4). For example, in Fig. 1a, the client cl_1 will observes a state where first key k_0 carries value 1 and second key k_2 carries value 0 if it takes a snapshot.

Definition 2.4 (Snapshots). Given the sets of program values $\textsc{Val}$ and keys $\textsc{Keys}$ (Def.2.1), the set of *snapshots* is:

$$ss \in \textsc{Snapshot} \triangleq \textsc{Keys} \xrightarrow{fin} \textsc{Val}$$

The *snapshot composition function*, $\bullet_{ss} : \textsc{Snapshot} \times \textsc{Snapshot} \rightharpoonup \textsc{Snapshot}$, is defined as $\bullet_{ss} \triangleq \uplus$, where $\uplus$ denotes the standard disjoint function union. The *snapshot unit element* is $\mathbf{1}_{ss} \triangleq \emptyset$, denoting a function with an empty domain. The *partial commutative monoid of snapshots* is $(\textsc{Snapshot}, \bullet_{ss}, \{\mathbf{1}_{ss}\})$. Then given a MKVS κ, and a view Vu, the snapshot of u in κ is defined as:

$$\mathsf{snapshot}(\kappa, u) \triangleq \lambda k.\, \mathsf{Value}(\kappa, k, u).$$

AC: General Comment on this Section: it is too abstract. We should give either here or in the introduction an example of computation - the write skew program should be okay that helps the reader understanding what's going on. Also, it could be also good to illustrate the notions of execution tests and consistency models.

2.2 Programming Language and Operational Semantics

In this section we define a simple programming language for client programs interacting with a key-value store where clients can only interact with the key-value store using transactions. We abstract from aborting transactions: rather than assuming that a transaction may abort due to a violation of the consistency guarantees given by the key-value store, we only allow the execution of a transaction when its effects are guaranteed to not violate the consistency model of the key-value store. This approach is equivalent to a setting where clients always restart a transaction after it aborts.

SX: The sentence is too long. I think before the ":" is enough

2.2.1 Programming Language. A program $\mathtt{P}$ contains fixed numbers of clients associated with their *commands* (Def. 2.5). Each client has a unique identifier $t \in \textsc{ClientID}$. For better presentation, we often write $\mathtt{C}_1 \parallel \mathtt{C}_2 \parallel \cdots \parallel \mathtt{C}_n$ as a syntactic sugar for a program $\mathtt{P}$ with implicit unique client identifiers $\mathtt{P} = \{\mathrm{cl}_1 \mapsto \mathtt{C}_1, \mathrm{cl}_2 \mapsto \mathtt{C}_2, \ldots, \mathrm{cl}_n \mapsto \mathtt{C}_n\}$, where $\mathtt{C}_i$ denotes commands. The *commands*, ranged over by $\mathtt{C}$, are defined by an inductive grammar comprising the standard constructs of $\mathtt{skip}$, sequential composition $(\mathtt{C};\mathtt{C})$, non-deterministic choice $(\mathtt{C}+\mathtt{C})$ and iterations $(\mathtt{C}^*)$. To simulate conditional branching and iterations, we have primitive commands: assume $(\mathtt{assume}(\mathtt{E}))$ and assignment $(\mathtt{x} := \mathtt{E})$, where $\mathtt{x}$ denotes stack variable and $\mathtt{E}$ denotes arithmetic expressions which have no side effect. Additionally, the programming language contains the *transaction* construct $\lceil \mathtt{T} \rceil$ denoting the *atomic* execution of the transaction $\mathtt{T}$. The atomicity guarantees the execution are dictated by the underlying consistency model. *Transactions*, ranged over by $\mathtt{T}$, are defined by a similar inductive grammar comprising $\mathtt{skip}$, the primitive command $\mathtt{T}_p$, non-deterministic choice, iteration and sequential composition. The primitive commands include assignment $(\mathtt{x} := \mathtt{E})$, lookup $(\mathtt{E} := [\mathtt{E}])$, mutation $([\mathtt{E}] := \mathtt{E})$ and assume $(\mathtt{assume}(\mathtt{E}))$. The arithmetic expression is interpreted to a value (Def. 2.7) via a stack $s \in \textsc{Stacks}$ (Def. 2.6) and there is no side-effect. Transactions do *not* contain the *parallel* composition construct $(\parallel)$ as they are to be executed atomically.

Definition 2.5 (Programming language). A *program*, $\mathtt{P} \in \textsc{Prog}$, is a partial finite function from thread identifiers to commands. Assuming the set of *variables* $\mathtt{x} \in \textsc{Vars}$, the commands $\mathtt{C} \in \textsc{Cmd}$ are defined by the following grammar:

$$\mathtt{C} ::= \mathtt{skip} \mid \mathtt{x} := \mathtt{E} \mid \mathtt{assume}(\mathtt{E}) \mid \lceil \mathtt{T} \rceil \mid \mathtt{C}\,;\mathtt{C} \mid \mathtt{C}+\mathtt{C} \mid \mathtt{C}^*$$

The $\mathtt{T} \in \textsc{Trans}$ in the grammar above denotes a *transaction* defined by the following grammar including the primitive transactional commands $\mathtt{T}_p$:

$$\begin{aligned} \mathtt{T}_p &::= \mathtt{a} := \mathtt{E} \mid \mathtt{a} := [\mathtt{E}] \mid [\mathtt{E}] := \mathtt{E} \mid \mathtt{assume}(\mathtt{E}) \mid \\ \mathtt{T} &::= \mathtt{skip} \mid \mathtt{T}_p \mid \mathtt{T}\,;\mathtt{T} \mid \mathtt{T}+\mathtt{T} \mid \mathtt{T}^* \end{aligned}$$

Given the set of *keys*, $k \in \textsc{Keys}$ (Def. 2.1), the set of *program values* is $v \in \textsc{Val} \triangleq \mathbb{N} \cup \textsc{Keys}$. The $\mathtt{E} \in \textsc{Expr}$ denotes an *arithmetic expression* defined by the grammar:

$$\mathtt{E} ::= v \mid \mathtt{x} \mid \mathtt{E}+\mathtt{E} \mid \mathtt{E} \times \mathtt{E} \mid \ldots$$

Definition 2.6 (Stacks). A *stack* is a partial finite function from variables $\textsc{Vars}$ (Def. 2.5) to program values $\textsc{Val}$ (Def. 2.5): $s \in \textsc{Stacks} \triangleq \textsc{Vars} \xrightarrow{fin} \textsc{Val}$.

Definition 2.7 (Evaluation of expression). Given a stack $s \in \textsc{Stacks}$ (Def.2.6), the *arithmetic expression evaluation* function, $[\![.]\!]_{(.)} : \textsc{Expr} \times \textsc{Stacks} \rightharpoonup \textsc{Val}$, is defined inductively over the structure of expressions as follows:

$$\begin{aligned} [\![v]\!]_s &\triangleq v \\ [\![\mathtt{x}]\!]_s &\triangleq s(\mathtt{x}) \\ [\![\mathtt{E}_1+\mathtt{E}_2]\!]_s &\triangleq [\![\mathtt{E}_1]\!]_s + [\![\mathtt{E}_2]\!]_s \\ [\![\mathtt{E}_1 \times \mathtt{E}_2]\!]_s &\triangleq [\![\mathtt{E}_1]\!]_s \times [\![\mathtt{E}_2]\!]_s \\ \ldots &\triangleq \ldots \end{aligned}$$

2.2.2 Semantics for transactions. When a transaction starts, it determines a local snapshot from the current state of the system and a view, which we will explain the process in Section 2.2.3. The transaction also installed with *fingerprints* which is initially empty. The fingerprints of a transaction corresponds to the set of all its interaction with the key-value store. Each time the transaction executes a primitive command $\mathtt{T}_p$ internal, the fingerprints get updated to trace the effect of the

command. At the end, the transaction throw away the local snapshot, but commit the fingerprints to the system.

The fingerprints include the *first read preceding a write* and *last write* for each key. This is because a transaction is executed atomically, all the intermediate steps are not observable from the outside world. The *fingerprints* formally is a set of *operations* Ops which are either read $(\mathtt{R}, k, v)$ from the k with the value v, or write $(\mathtt{W}, k, v)$ to the key k with the value v (Def. 2.8). Note that In the Def. 2.8, the $(.)\lfloor_{(.)}$ denotes projection. For a tuple, for example $o\lfloor_i$, it gives the *i-th* element of the tuple. It is lifted to a set of tuples, for example $\mathcal{F}\lfloor_i$, which gives a set of all the *i-th* elements. The well-formedness condition for fingerprints asserts it is a set of operations in which there are at most one read and one write for each key. The composition, then, is defined as set disjointed union as long as the result is well-formed.

Definition 2.8 (operation and fingerprints). A *transaction operation* $o \in \textsc{Ops}$ is a tuple of *an operation tag* that is either read or write, an key and a value.

$$o \in \textsc{Ops} \triangleq \mathcal{P}\left(\{\mathtt{R}, \mathtt{W}\}\right) \times \textsc{Keys} \times \textsc{Val}$$

A well-formed fingerprint, $\mathcal{F} \in \textsc{Fp}$, is a subset of Ops in which any two elements contain either different tags or different key.

$$\textsc{Fp} \triangleq \left\{\mathcal{F} \;\middle|\; \mathcal{F} \subseteq \textsc{Ops} \wedge \forall o, o' \in \mathcal{F}.\, o\lfloor_1 \neq o'\lfloor_1 \vee o\lfloor_2 \neq o'\lfloor_2\right\}$$

The unit element is $\mathbf{1}_{\mathcal{F}} \triangleq \emptyset$ and the composition of two fingerprints is the disjointed union when the two sets contain disjointed keyes,

$$\mathcal{F} \bullet_{\mathcal{F}} \mathcal{F}' \triangleq \begin{cases} \mathcal{F} \uplus \mathcal{F}' & \text{if } \mathcal{F}\lfloor_2 \cap \mathcal{F}'\lfloor_2 = \emptyset \\ \text{undefined} & \text{otherwise} \end{cases}$$

PG: $\mathcal{F} \lll \mathcal{F}'$?.

SX: We can define this but not sure it is useful

Lemma 2.9. *The well-formedness of fingerprints is closed under* $\lll$.

The operational semantics for transactions T (Fig. 2) is defined with respect to a transactional state of the form $(s, ss, \mathcal{F})$ comprising a stack, a snapshot and *fingerprints*. We first define a state transformers on pairs of stacks and snapshots for the primitive commands $\mathtt{T}_p$. We also define its fingerprint by fp function, which denotes the contribution of the primitive command that might observed by the external environment, *i.e.* transactions from other threads. The fp extracts the read or write operation from loop-up and mutation respectively, otherwise ϵ. We also define a binary operator $\mathcal{F} \lll o$ that specifies the effects of adding a new operation o to the fingerprints $\mathcal{F}$. If the new operation is a read, for example $(\mathtt{R}, k, v)$ where k is the key and v is the associated value, and there is no other operation related to the same key, this new read operation will be included in the result. Meanwhile, if the new operation is a write, it will overwrite all preview write operations to the same key. This ensures the fingerprints contains only the first read preceding a write, and only the last write for each key. This choice is motivated by the fact that we only focus on atomically visible transactions: keys are read from a snapshot of the database, and new version are written only at the moment the transaction commits. For technical reasons, if the right hand side is a special token ϵ corresponding to a command does not result in an interaction with key-value store. Therefore, the semantics for primitive command TPrimitive updates the stack and heap by the transformers relation and updates the operation set by first extracting the operation and adding it via $\lll$ operator. The semantics for non-deterministic choices TChoice, sequential compositions TSeqSkip and TSeq, and iteration TIter have the expected behaviours.

The state transformers on pairs of stacks and snapshots for the primitive commands T_p (left), and the op for extracting the operation from the primitive commands (right):

$$(s, ss) \xrightarrow{\mathsf{x:=E}} (s[\mathsf{x} \mapsto [\![\mathsf{E}]\!]_s], ss)$$
$$(s, ss) \xrightarrow{\mathsf{x:=[E]}} (s[\mathsf{x} \mapsto ss([\![\mathsf{E}]\!]_s)], ss)$$
$$(s, ss) \xrightarrow{[\mathsf{E}_1]:=\mathsf{E}_2} (s, ss[[\![\mathsf{E}_1]\!]_s \mapsto [\![\mathsf{E}_2]\!]_s])$$
$$(s, ss) \xrightarrow{\mathsf{assume(E)}} (s, ss) \text{ where } [\![\mathsf{E}]\!]_s \neq 0$$

$$\mathsf{fp}\,(s, ss, \mathsf{x} := \mathsf{E}) \triangleq \epsilon$$
$$\mathsf{fp}\,(s, ss, \mathsf{x} := [\mathsf{E}]) \triangleq (\mathsf{R}, [\![\mathsf{E}]\!]_s, ss([\![\mathsf{E}]\!]_s))$$
$$\mathsf{fp}\,(s, ss, [\mathsf{E}_1] := \mathsf{E}_2) \triangleq (\mathsf{W}, [\![\mathsf{E}_1]\!]_s, [\![\mathsf{E}_2]\!]_s)$$
$$\mathsf{fp}\,(s, ss, \mathtt{assume(E)}) \triangleq \epsilon$$
$$\mathsf{fp}\,(s, ss, \mathtt{return(E)}) \triangleq \epsilon$$

The binary operator $\mathcal{F} \lll o$ that specifies the effects of adding a new operation o to the set $\mathcal{F}$:

$$\mathcal{F} \lll (\mathsf{R}, k, v) \triangleq \begin{cases} \mathcal{F} \uplus \{(\mathsf{R}, k, v)\} & (-, k, -) \notin \mathcal{F} \\ \mathcal{F} & \text{otherwise} \end{cases} \qquad \mathcal{F} \lll (\mathsf{W}, k, v) \triangleq (\mathcal{F} \setminus \{(\mathsf{W}, k, -)\}) \uplus \{(\mathsf{W}, k, v)\}$$
$$\mathcal{F} \lll \epsilon \triangleq \mathcal{F}$$

Given the set of stacks Stacks (Def. 2.6), heaps Snapshot (Def.2.4) and transactions Trans (Def.2.5) and the arithmetic expression evaluation $[\![\mathsf{E}]\!]_s$ (Def. 2.5), the *operational semantics of transactions*:

$$\rightsquigarrow : ((\textsc{Stacks} \times \textsc{Snapshot} \times \textsc{Fp}) \times \textsc{Trans}) \times ((\textsc{Stacks} \times \textsc{Snapshot} \times \textsc{Fp}) \times \textsc{Trans})$$

$$\frac{(s, ss) \xrightarrow{\mathsf{T}_p} (s', ss') \qquad o = \mathsf{fp}\,(s, ss, \mathsf{T}_p)}{\langle (s, ss, \mathcal{F}), \mathsf{T}_p \rangle \rightsquigarrow \langle (s', ss', \mathcal{F} \lll o), \mathtt{skip} \rangle}\ \textsc{TPrimitive}$$

$$\frac{i \in \{1, 2\}}{(s, ss, \mathcal{F}), \mathsf{T}_1 + \mathsf{T}_2 \rightsquigarrow (s, ss, \mathcal{F}), \mathsf{T}_i}\ \textsc{TChoice} \qquad \frac{}{(s, ss, \mathcal{F}), \mathsf{T}^* \rightsquigarrow (s, ss, \mathcal{F}), \mathtt{skip} + (\mathsf{T}\,;\mathsf{T}^*)}\ \textsc{TIter}$$

$$\frac{}{(s, ss, \mathcal{F}), \mathtt{skip}\,;\mathsf{T} \rightsquigarrow (s, ss, \mathcal{F}), \mathsf{T}}\ \textsc{TSeqSkip} \qquad \frac{(s, ss, \mathcal{F}), \mathsf{T}_1 \rightsquigarrow (s', ss', \mathcal{F}'), \mathsf{T}_1'}{(s, ss, \mathcal{F}), \mathsf{T}_1\,;\mathsf{T}_2 \rightsquigarrow (s', ss', \mathcal{F}'), \mathsf{T}_1'\,;\mathsf{T}_2}\ \textsc{TSeq}$$

Fig. 2. Operational semantics for transactions

2.2.3 Program Semantics. The semantics for commands is in Fig. 3, taking the form:

$$(\kappa, s, u), \mathsf{C} \xrightarrow{\mathsf{ET}} (\kappa', s', u'), \mathsf{C}'$$

where κ, κ' are multi-version key-value stores (Def. 2.1), s, s' are the stacks (Def. 2.6), u, u' are views (Def. 3.3), and ET is *an execution test* (Def. 2.10), i.e. a condition that must be satisfied in order for a client to execute a transaction safely. We parametrise the execution test in the operational semantics of commands (Fig. 3), while we will give examples of different execution tests in Section 5.

Execution tests is a set of quadruples $(\kappa, u, \mathcal{F}, u')$ consisting of a key-value store, a view before execution, a operation set and a view after committing of the operation set. We often write $(\kappa, u) \triangleright_{\mathsf{ET}} \mathcal{F} : u'$ in lieu of $(\kappa, u, \mathcal{F}, u') \in \mathsf{ET}$. The quadruple describes that when the state of the key-value store is κ, a client who has view u is allowed to execute a single transaction that has the fingerprints $\mathcal{F}$, and then after the commit the thread view must be updated to at least u'.

AC: There we also note that by tweaking the execution test used by the semantics, we capture different consistency models of key-value stores.

Definition 2.10 (Execution Tests). Given the set of key-value stores $\kappa \in$ MKVSs (Def. 2.1), fingerprints $\mathcal{F} \in$ Fp (Def. 2.8) and views $u, u' \in$ Views (Def. 2.2), *execution tests* $\mathsf{ET} \in$ ET is a set of quadruples in the form of $(\kappa, u, \mathcal{F}, u')$:

$$\mathsf{ET} \in \mathrm{ET} \triangleq \mathcal{P}\,(\text{MKVSs} \times \text{Views} \times \text{Fp} \times \text{Views})$$

Well-formed execution tests ET require the domain of views and the key-value store are the same, and the fingerprints only have keys included in the previous domain:

$$\forall \kappa, u, u', \mathcal{F}.\ (\kappa, u, \mathcal{F}, u') \in \mathsf{ET} \implies \mathcal{F}|_2 \subseteq \mathrm{dom}(u) = \mathrm{dom}(u') = \mathrm{dom}(\kappa)$$

SX: Maybe also some version of composition requirement. For example, the composition of two should be also included in the consistency model.

$$\forall m, m'.\ m \in \mathsf{ET} \wedge m' \in \mathsf{ET} \implies m \bullet m' \in \mathsf{ET}$$

where $\bullet \triangleq (\bullet_\kappa, \bullet_u, \bullet_\mathcal{F}, \bullet_u)$.

The main rule in the operational semantics for commands is the one corresponding to the execution of a transaction PCommit. First, the view can shift to later versions before executing the transaction to model the client might gain more information about the key-value store since its last commit. To recall The order between two views with the same domain, for example $u'' \geq u$ in PCommit, is defined by the order of the indexes (Def. 2.2). This new local view u'' should also be consistent with the key-value stores, *i.e.* it leads to a situation where the current client is allowed to execute the transaction. The transaction code T is executed locally given an initial snapshot $ss = \mathrm{snapshot}(\kappa, u'')$ (Def. 3.4) decided by the current state of key-value store κ and the local view u''. After local execution via the semantics for transactions (Fig. 2), we propagate the stack s' and more importantly obtain the fingerprints $\mathcal{F}$, while the snapshot ss' will be throw away. Then the transaction picks a fresh identifier t, *i.e.* one that does not appear in the key-value store, and commits the fingerprint $\mathcal{F}$, which will update the key-value store and local view. The updateMKVS function updates the history heap using the fingerprint, $\kappa' = \mathrm{updateMKVS}\,(\kappa, u'', t, \mathcal{F})$. For read operations, it includes the new identifier to the read set of the version the is pointed by the local view u''. For write operations of the form $(\mathtt{W}, k, n)$, it extends a new version written by the new transaction $(n, t, \emptyset)$ to the tail of $\kappa(k)$. For updating the view, we set a lower bound for the new local view by updateView function. Assuming the commands executed by clients are wrapped with in a single session, the lower bound of the view corresponds to the strong session guarantees introduced by [?]. This function shifts the view to the up-to-date version in the new key-value store if the version is installed by the current transaction. This guarantees strong program order, meaning the following transaction will at least read its own write. Finally, the actual new local view u' is any view greater than the lower bound $u' \geq \mathrm{updateView}\,(\kappa', u'', \mathcal{F})$. The overall execution satisfied the execution tests, *i.e.* $(\kappa, u'') \triangleright_{\mathrm{ET}} \mathcal{F} : u'$.

AC: The paragraph below should probably go when discussing the rules of the semantics:
Note that the way in which MKVSs and views are updated ensure the following: • a client always reads its own preceding writes; • clients always read from an increasingly up-to-date state of the database; • the order in which clients update a key k is consistent with the order of the versions for such keys in the MKVS; • writes take place after reads on which they depend.

Lemma 2.11. *The function* updateMKVS *is well-defined over well-formed MKVS, fingerprints and views. Given a well-formed MKVS κ, a view u that is well-formed with κ, well-formed fingerprints $\mathcal{F}$,*

For any functions f, the notation $f[k \mapsto v]$ means the result by replacing or extending the element associated with k to v. For any lists or tuples l, the notation $l[i \mapsto k]$ means the result by replacing the i-*th* element to k. To record the index of list starts from 0. The $+\!\!+$ denotes list concatenation.

$$\begin{aligned}
\mathsf{updateMKVS}\,(\kappa, u, t, \mathbf{1}_{\mathcal{F}}) &\triangleq \kappa \\
\mathsf{updateMKVS}\,(\kappa, u, t, \mathcal{F} \uplus \{(\mathtt{R}, k, -)\}) &\triangleq \mathtt{let}\ (n, t', \mathcal{T}) = \kappa(k, u(k)) \\
&\quad\ \mathsf{and}\ \kappa' = \kappa\big[k \mapsto \kappa(k)\big[u(k) \mapsto (n, t', \mathcal{T} \uplus \{t\})\big]\big] \\
&\quad\ \mathtt{in}\ \mathsf{updateMKVS}\,(\kappa', u, t, \mathcal{F}) \\
\mathsf{updateMKVS}\,(\kappa, u, t, \mathcal{F} \uplus \{(\mathtt{W}, k, n)\}) &\triangleq \mathtt{let}\ \kappa' = \kappa[k \mapsto (\kappa(k) +\!\!+ [(n, t, \emptyset)])] \\
&\quad\ \mathtt{in}\ \mathsf{updateMKVS}\,(\kappa', u, t, \mathcal{F}) \\
\mathsf{updateView}\,(\kappa, u, \mathbf{1}_{\mathcal{F}}) &\triangleq u \\
\mathsf{updateView}\,(\kappa, u, \mathcal{F} \uplus \{(\mathtt{R}, k, -)\}) &\triangleq \mathsf{updateView}\,(\kappa, u, \mathcal{F}) \\
\mathsf{updateView}\,(\kappa, u, \mathcal{F} \uplus \{(\mathtt{W}, k, -)\}) &\triangleq \mathsf{updateView}\,(\kappa, u[k \mapsto (|\kappa(k)| - 1)], \mathcal{F}) \\
\mathsf{fresh}\,(\kappa) &\triangleq \{t \mid t \in \textsc{TransID} \wedge \forall k, i.\, t \neq \mathsf{Write}(\kappa(k, i)) \wedge t \notin \mathsf{Reads}(\mathcal{H}^{\mathtt{r}}(k, i))\}
\end{aligned}$$

The function snapshot is defined in Def. 3.4 and operational semantics for transactions $\rightsquigarrow$ in Fig. 2. Given the set of executions tests ET (Def. 2.10), stacks Stacks (Def. 2.6), multi-version key-value stores MKVSs (Def. 2.1) and views Views (Def. 2.2), the *operational semantics for commands*:

$$\rightarrow : ((\text{MKVSs} \times \textsc{Stacks} \times \textsc{Views}) \times \textsc{Cmd}) \times \text{ET} \times ((\text{MKVSs} \times \textsc{Stacks} \times \textsc{Views}) \times \textsc{Cmd})$$

$$\frac{\begin{array}{l} u'' \geq u \quad t \in \mathsf{fresh}\,(\kappa) \quad ss = \mathsf{snapshot}(\kappa, u'') \quad (s, ss, \mathbf{1}_{\mathcal{F}}), \mathtt{T} \rightsquigarrow^{*} (s', ss', \mathcal{F}), \mathtt{skip} \\ \kappa' = \mathsf{updateMKVS}\,(\kappa, u'', t, \mathcal{F}) \quad u' \geq \mathsf{updateView}\,(\kappa', u'', \mathcal{F}) \quad (\kappa, u) \triangleright_{\text{ET}} \mathcal{F} : u' \end{array}}{(\kappa, s, u), \big[\mathtt{T}\big] \xrightarrow{\text{ET}} (\kappa', s', u'), \mathtt{skip}}\ \textsc{PCommit}$$

$$\frac{v = [\![\mathtt{E}]\!]_s}{(\kappa, s, u), \mathtt{x} := \mathtt{E} \xrightarrow{\text{ET}} (\kappa, s[\mathtt{x} \mapsto v], u), \mathtt{skip}}\ \textsc{PAssign} \qquad \frac{[\![\mathtt{E}]\!]_\sigma \neq 0}{(\kappa, s, u), \mathtt{assume(E)} \xrightarrow{\text{ET}} (\kappa, s, u), \mathtt{skip}}\ \textsc{PAssume}$$

$$\frac{i \in \{1, 2\}}{(\kappa, s, u), \mathtt{C}_1 + \mathtt{C}_2 \xrightarrow{\text{ET}} (\kappa, s, u), \mathtt{C}_i}\ \textsc{PChoice} \qquad \frac{}{(\kappa, s, u), \mathtt{C}^* \xrightarrow{\text{ET}} (\kappa, s, u), \mathtt{skip} + (\mathtt{C}\,;\mathtt{C}^*)}\ \textsc{PIter}$$

$$\frac{}{(\kappa, s, u), \mathtt{skip}\,;\mathtt{C} \xrightarrow{\text{ET}} (\kappa, s, u), \mathtt{C}}\ \textsc{PSeqSkip} \qquad \frac{(\kappa, s, u), \mathtt{C}_1 \xrightarrow{\text{ET}} (\kappa, s', u'), \mathtt{C}_1{}'}{(\kappa, s, u), \mathtt{C}_1\,;\mathtt{C}_2 \xrightarrow{\text{ET}} (\kappa, s', u'), \mathtt{C}_1{}'\,;\mathtt{C}_2}\ \textsc{PSeq}$$

The thread environment is a partial function from thread identifiers to pairs of stacks and views $Env \in \textsc{EnvEnv} \triangleq \textsc{ClientID} \xrightarrow{fin} \textsc{Stacks} \times \textsc{Views}$. Given the set of execution tests ET (Def. 2.10) and key-value stores MKVSs (Def. 2.1), the *semantics for programs*:

$$. \Rightarrow_{.} . : (\text{MKVSs} \times \textsc{EnvEnv} \times \textsc{Prog}) \times \text{ET} \times (\text{MKVSs} \times \textsc{EnvEnv} \times \textsc{Prog})$$

$$\frac{(s, u) = Env(\mathrm{cl}) \qquad \mathtt{C} = \mathtt{P}(\mathrm{cl}) \qquad (\kappa, s, u), \mathtt{C}, \xrightarrow{\text{ET}} (s', \kappa', u'), \mathtt{C}'}{(\kappa, Env, \mathtt{P}) \Rightarrow_{\text{ET}} (\kappa', Env[\mathrm{cl} \mapsto (s', u')], \mathtt{P}[\mathrm{cl} \mapsto \mathtt{C}'])}\ \textsc{PSingleThread}$$

Fig. 3. operational semantics for threads and programs

and a t that does not appear in κ, then $\mathsf{UpdateMKVS}(\kappa, \mathcal{F}, t, u)$ *is uniquely determined and yields a well-formed MKVS.*

Lemma 2.12. *The function* UpdateView *is well-defined. Given a well-formed MKVS* κ*, a view* u *that is well-formed with* κ*, well-formed fingerprints* $\mathcal{F}$*, then* $\mathsf{UpdateView}(\kappa, \mathcal{F}, u)$ *is uniquely determined a view and the view is well-formed with the MKVS.*

Last, the program has standard interleaving semantics by picking a client and then progressing one step (Fig. 3). To achieve that a thread environment holds the stacks and views associated with all active clients $Env \in \textsc{ThdEnv}$. We assume the client identifiers from client environment match with those in the program P. We also assume all the stacks and views initially are the same respectively.

2.2.4 Example of Operational Semantics. To conclude our discussion on the operational semantics, we show in detail one possible computation of a program P_1 consisting of two transactions executing in parallel:

$$\mathsf{P}_1 \equiv t_1 : \begin{bmatrix} \mathsf{[x]} := 1; \\ \mathsf{[y]} := 1; \end{bmatrix} \Bigg\| \; t_2 : \begin{bmatrix} \mathsf{a} := \mathsf{[x]}; \\ \mathsf{b} := \mathsf{[y]}; \\ \mathtt{if}\,(\mathsf{a} = 1 \wedge \mathsf{b} = 0)\{ \\ \quad \mathsf{ret} := \text{☹}\} \end{bmatrix}$$

The $\mathtt{if}\,(\mathsf{E})\{\mathsf{C}_1\}\mathtt{else}\{\mathsf{C}_2\}$ is encoded as $(\mathtt{assume}(\mathsf{E})\,;\,\mathsf{C}_1) + (\neg\mathtt{assume}(\mathsf{E})\,;\,\mathsf{C}_2)$. To recall, we often write $\mathsf{C}_1 \parallel \mathsf{C}_2 \parallel \cdots \parallel \mathsf{C}_n$ as a syntactic sugar for a program P with implicit unique thread identifiers $\mathsf{P} = \{\mathrm{cl}_1 \mapsto \mathsf{C}_1, \mathrm{cl}_2 \mapsto \mathsf{C}_2, \ldots, \mathrm{cl}_n \mapsto \mathsf{C}_n\}$. For better presentation, we annotated transactions with unique identifiers, yet they are allocated dynamically in the semantics. We also treat the value assigned to the ret variable as *returned value*. Assume the variables x and y refer to two key, and a and b are local variables to threads.

The special symbol ☹, for example the returned value by the transaction t_2, is to emphasise some undesirable behaviour of a transaction. In this case, the undesirable behaviour corresponds to the transaction to the right t_2 observing only one of the updates from t_1. Intuitively, this behaviour violates the constraints that transactions should be executed atomically (further discussed in Section **??**), we want to show that if no restrictions are placed on the consistency model specification, it is possible for P_1 to reach a configuration where the second transaction t_2 returns ☹. To illustrate this and also explain the semantics, we instantiate the operation semantics with the most permissive execution tests $\triangleright_\top$, *i.e.* the view after u' at least observes its own writes and no other constraint:

$$(\kappa, u) \triangleright_\top \mathcal{F} : u' \triangleq \mathsf{updView}(\kappa, u, \mathcal{F}) \leq u'$$

AC: The condition on V' is not really needed.

Before any computation, the initial configuration for P_1 is the one in which there are two keys k_x and k_y where each key is associated with an initial version with value zero written by an initialisation transaction t_0, $\kappa_0 = \{k_\mathsf{x} \mapsto [(0, t, \emptyset)], k_\mathsf{y} \mapsto [(0, t, \emptyset)]\}$. The initial view of each thread points to the initial version of each key, $u_0^1 = u_0^2 = \{k_\mathsf{x} \mapsto 1, k_\mathsf{y} \mapsto 1\}$. The two threads have the same initial stack containing two variables x and y referring to the only keys in the key-value store respectively, *i.e.* $s_0^1 = s_0^2 = \{\mathsf{x} \mapsto k_\mathsf{x}, \mathsf{y} \mapsto k_\mathsf{y}\}$. Therefore the initial configuration is $(\kappa_0, Env_0, \mathsf{P}_1)$ where $Env_0 = \{\mathrm{cl}_1 \mapsto (s_0^1, u_0^1), \mathrm{cl}_2 \mapsto (s_0^2, u_0^2)\}$. Fig. 4a gives a graphical representation of the initial configuration.

We are now ready to show how to derive a computation of P_1 that violates *atomic visibility* and it will be explained formally in Section **??**. In the specific computation, we choose to let the transaction cl_1 commit first. According to rule PCommit, we need to perform the following steps:

SX: Not sure the bullet points work here? Just typesetting.

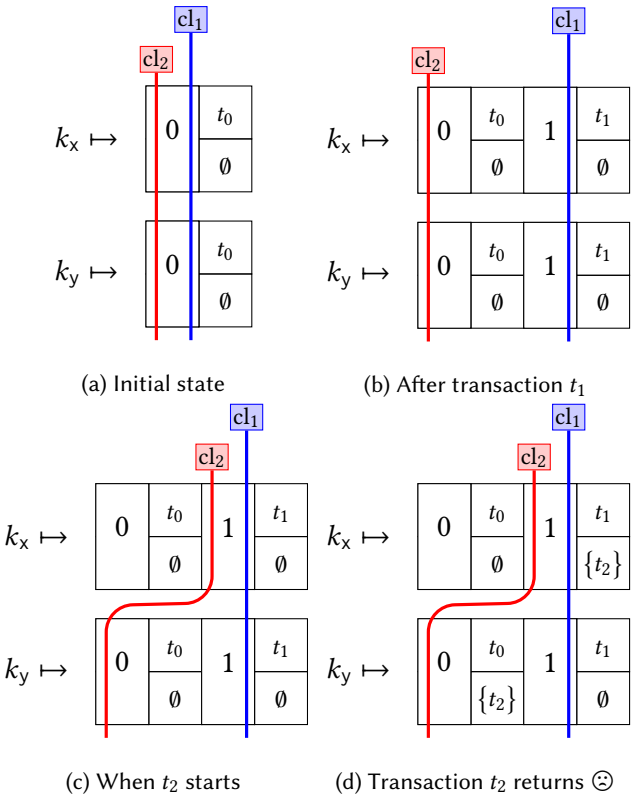


Fig. 4. Graphical Representation of configurations obtained through the execution of P_1.

- Arbitrarily shift the view u_0^1 for thread cl_1 to the right as long as it is within the bound of key-value store and obtain a view u'' such that $u'' \geq u_0^1$. Because κ_0 contains only one version per key, here the only possibility is that $u'' = u_0^1$.
- Determine the initial snapshot $ss = \mathrm{snapshot}(\kappa_0, u'')$ for the transaction cl_1. In this case, we have that $ss = \{k_x \mapsto 0, k_y \mapsto 0\}$.
- Given the initial snapshot ss, initially empty fingerprint $\mathbf{1}_{\mathcal{F}}$ and the stack s_0^1, by the operational semantics for transaction (Fig. 2), after executing the transactional codes $[\mathtt{x}] := 1; [\mathtt{y}] := 1$, the finial fingerprints include two write operations as $\mathcal{F} = \{(\mathtt{W}, k_\mathtt{x}, 1), (\mathtt{W}, k_\mathtt{y}, 1)\}$.

AC:

SX: Not sure those details are necessary.

This amounts to execute the transaction in isolation from the external environment, using the rules in the operational semantics for transactions. Because this execution must match the premiss of Rule $(C - Tx)$, The code is run using h as the initial heap, σ_0 as the initial thread stack, $\tau_0 = \lambda_a.0$ as the initial transaction stack, and $\emptyset$ as the initial fingerprint. We only need to apply Rule $(Tx - prim)$ twice, in which case we obtain

$$
\begin{array}{rcl}
 & \sigma_0 \vdash \left\langle h_0, \_, \emptyset, \begin{array}{l}[k_x] := 1; \\ [k_y] := 1\end{array} \right\rangle & \rightarrow \\
\rightarrow & \left\langle h_0[[k_x] \mapsto 1], \_, (\emptyset \oplus \mathbf{WR}\ [k_x] : 1), [k_y] := 1 \right\rangle & = \\
= & \left\langle h_0[[k_x] \mapsto 1], \_, \{\mathbf{WR}\ [k_x] : 1\}, [k_y] := 1 \right\rangle & \rightarrow \\
\rightarrow & \left\langle h_0[[k_x] \mapsto_1, [k_y] \mapsto 1], \_, (\{\mathbf{WR}\ [k_x] : 1\} \oplus (\mathbf{WR}\ [k_y] : 1)), - \right\rangle & = \\
= & \left\langle \_, \_, \{\mathbf{WR}\ [k_x] : 1, \mathbf{WR}\ [k_y] : 1\}, - \right\rangle &
\end{array} \quad (1)
$$

Therefore, we conclude $O = \{\mathbf{WR}\ [k_x] : 1, \mathbf{WR}\ [k_y] : 1\}$.

- The transaction throws away the local snapshot and commits the fingerprints to the key-value store. A fresh transaction identifier t_1 is picked. The new key-value store κ_1 is determined by the functions $\kappa_1 = \mathsf{updHisHp}\,(\kappa_0, u'', t_1, \mathcal{F})$ and the lower bound of the new view is given by the function $\mathsf{updView}\,(\kappa_1, u'', \mathcal{F})$.
- Last, the local view shift to the right so that it satisfies the execution tests, $u' \geq \mathsf{updView}\,(\kappa_1, u'', \mathcal{F}) \wedge (\kappa_0, u'') \triangleright_\top u' : \mathcal{F}$. In this case, the permissive model does not constraint the view at all. Therefore the overall final state is in Fig. 4b.

Next, thread cl_2 executes its own transaction. Note that the view for cl_2 still points to initial versions and the semantics allows the view get updated arbitrarily before executing the transaction. Because the key-value store now has two versions for each key, there are exactly four possible views for the transaction t_2. In particular, assume it updates the view for k_x but not k_y, *i.e.* $\{k_x \mapsto 1, k_x \mapsto 0\}$ (Fig. 4c). Given the view, the transaction t_2 will assign ☹ to the `ret` and this transaction is allowed to commit since the commit test does not stop this (Fig. 4d).

AC: ORDINAL VERNON starts from here.

3 HISTORY HEAPS AND VIEWS

We focus on a computational model where multiple client programs can access and update locations in a key-value store using atomic transactions. Transactions in our model execute atomically, though the consistency guarantees that they provide do not necessarily correspond to *serialisability*. This means that, at the moment of executing, a transaction may not observe the most up-to-date value of a location.

To overcome this issue, we model the state of the system using *multi-version key-value stores* (MKVSs) and *views*. A MKVS keeps track of all the versions written for any key, as well as the information about the transactions that read and wrote such versions. Views keep track of the version observed for each key by clients.

Formally speaking, we assume a countably infinite set of keys $\textsc{Keys} = \{\mathrm{k}_1, \cdots\}$, a set of transaction identifiers $\textsc{TransID} = \{t_1, \cdots\}$, a set of clients $\textsc{Clients} = \{cl_1, \cdots\}$. We also assume values to be natural numbers from the set n.

AC: from now on, the sort font is used for sets.

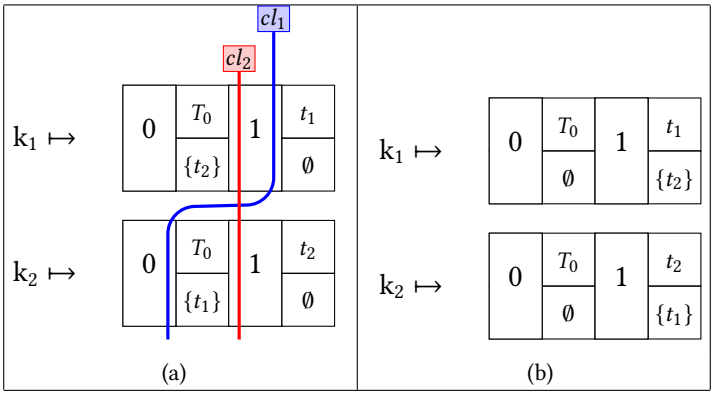


Fig. 5. (a) - A well-formed configuration (b) - An ill-formed MKVS.

Definition 3.1. A *version* is a triple $v = (n, t, \mathcal{T})$, where, n is the value of the version, t is the identifier of the transaction that wrote the version, and $\mathcal{T}$ is a (possibly empty) set of identifiers of the transactions that read the version. Given a version $v = (n, t, \mathcal{T})$, we let $\mathsf{Value}(v) = n$, $\mathsf{Write}(v) = t$, $\mathsf{Reads}(v) = \mathcal{T}$. The set of versions is denoted as $\textsc{Versions}$.

A *Multi-version Key-value Store*, or MKVS, is a mapping $\kappa : \textsc{Keys} \to \textsc{Versions}^*$ from keys to lists of versions.

Given a list of versions $v_1 \cdots v_n$, we let $|v_1 \cdots v_n| = n$ be its length. Also, let κ be a MKVS, k be a key such that $\kappa(\mathrm{k}) = v_1 \cdots v_n$, and $i \leq n$ be a strictly positive natural number; then we let $\kappa(\mathrm{k}, i) = v_i$.

We often depict MKVSs graphically. One example is given by the MKVS κ_0 of Figure 5(a) (ignore for the moment the straight lines labelled cl_1 and cl_2).

AC: Maybe it's better to keep $\textsc{Keys}$ fixed and say that we look at only a fragment of the key value store. Alternatively, we can go for partial mappings to represent MKVSs, but still avoiding allocation and deallocation of keys.

. To the left we have the set of keys stored y κ_0, in this case k_1 and k_2. To the right, on the same line of a key, a matrix containing the list of versions stored by such a key in κ_0. Starting from the first column, each version is represented by two adjacent columns in the matrix: the column to the left gives the value of the version, while the column to the right contains the identifier of the transaction that wrote the version to the top, and the identifiers of the transactions that read such a version to the bottom. In the case of κ_0, there are two versions stored for k_1: the first one with value 0, written by t_0 and read by t_2; and the second one with value 1, written by t_2 and read by no transaction.

Throughout this paper, we focus on MKVSs that can be obtained in databases whose consistency guarantees enjoy atomic visibility [Cerone et al. 2015; Cerone and Gotsman 2016; Cerone et al. 2017]. To this end, we impose some well-formedness constraints on the MKVSs.

Definition 3.2. A MKVS κ is *well-formed* if and only if

- a transaction does not write two different versions for the same key: $\forall \mathrm{k} \in \textsc{Keys}.\ \forall i, j = 1, \cdots, |\kappa(\mathrm{k})|.\mathsf{Write}(\kappa(\mathrm{k}, i)) = \mathsf{Write}(\kappa(\mathrm{k}, j)) \implies i = j$,
- a transaction does not read two different versions for the same key: $\forall \mathrm{k} \in \textsc{Keys}.\ \forall i, j = 1, \cdots, |\kappa(\mathrm{k})|.(\mathsf{Reads}(\kappa(\mathrm{k}, i) \cap \mathsf{Reads}(\kappa(\mathrm{k}, j)) \neq \emptyset) \implies i = j$.

- There are no circular dependencies in versions. Given two versions v_1, v_2, we say that v_2 *direct dependency* from v_1, written $v_1 \xrightarrow{\text{ddep}} v_2$, if $\text{Write}(nu_2) \in \text{Reads}(v_1)$; that is, some transaction t wrote the version v_2 after reading v_1

  AC: A point that this does not ensure a real causal dependency between the two versions, yet it is consistent with the notion of causality employed in databases, should be made

  . If v_1, v_2 appear as versions of some object in κ, then we write $v_1 \xrightarrow{\text{ddep}(\kappa)} v_2$. Then the relation $\left(\xrightarrow{\text{ddep}(\kappa)}\right)^+$ is acyclic in κ, i.e. $\left(\xrightarrow{\text{ddep}(\kappa)}\right)^+ \cap \text{Id} = \emptyset$ (where Id is the identity relation).

Let us discuss for a moment the last constraint in Definition 3.2, which states that there is no circularity in dependencies stored by a MKVS. This is to ensure that no versions are created *out of thin-air*. An example of the out of thin-air anomaly is given by the MKVS κ_1 of Figure 5(b): there we have a transaction t_2 that read the value of k_1 written by t_1; conversely, t_1 read the value of k_2 written by t_2. Because we assume that transactions read a state of the key-value store from an atomic snapshot fixed at the moment they execute, this situation cannot happen. For t_2 to read the value installed by t_1, then t_2 must have been executed after t_1. Simlarly, t_1 must have been executed after t_2. Formally, we have that $\kappa_1(\text{k}_2, 2) \xrightarrow{\text{ddep}(\kappa_1)} \kappa(\text{k}_1, 2)$, and $\kappa_1(\text{k}_1, 2) \xrightarrow{\text{ddep}(hh_1)} \kappa_1(\text{k}_2, 2)$, violating the constraint of well-formed MKVSs that states that $\xrightarrow{\text{ddep}(\kappa_1)}$ is acyclic. When introducing our semantics of clients in §4, we show that (under reasonable conditions) it generates only well-formed MKVSs.

Because MKVSs store multiple versions for each key, different clients may observe different versions of the same key, at a single instant of time. To model this fact, we introduce the notion of *views* and *configurations*.

Definition 3.3. A *View* is a function $V : \textsc{Keys} \to n$. The set of all views is denoted as $\textsc{Views}$.

A *configuration* is a pair $C = (\kappa, \mathcal{U})$, where $\mathcal{U} : \textsc{Clients} \rightharpoonup \textsc{Views}$ is a partial mapping from clients to views.

A view V defines the particular version of each key that a client will observe when executing a transaction. A configuration consists of a MKVS, and the views that a set of clients have each on the state of the MKVS. An example of configuration is given in Figure 5(a). There are two clients, cl_1 and cl_2, each with their own view (represented in the Figure by labelled lines crossing the MKVS at each location). According to the view of cl_1, formally defined as $V_1 = [\text{k}_1 \mapsto 2], \text{k}_2 \mapsto 1]$, this client observes in κ the second version of key k_1, carrying value 1, and the first version of k_2, carrying value 0. Similarly, according to its view $V_2 = [\text{k}_1 \mapsto 2, \text{k}_2 \mapsto 2]$, the client cl_2 observes in $\kappa$the second and most up-to-date version for both k_1 and k_2.

A view V is well-formed with respect to a MKVS κ iff, for any $\text{k} \in \textsc{Keys}$. $0 < V(\text{k}) \leq |\kappa(\text{k})|$. A configuration $C = (\kappa, \mathcal{U})$ is well-formed if, for any $cl \in \text{dom}(\mathcal{U})$, the view $\mathcal{U}(cl)$ is well-formed with respect to κ. Henceforth, we always assume that MKVSs, views and configurations are well-formed, unless otherwise stated. and a MKVS κ, we let $\text{versionOf}(\kappa, \text{k}, V) = \kappa(\text{k}, V(\text{k}))$; we commit an abuse of notation and often write $\text{Value}(\kappa, \text{k}, V)$ in lieu of $\text{Value}(\text{versionOf}(\kappa, \text{k}, V))$, and similarly for Write, Reads. if $v = \text{versionOf}(\kappa, \text{k}, V)$, we say that V k-points to v in κ. If $v = \kappa(\text{k}, i)$ for some $i \leq V(\text{k})$, we say that V k-includes v in κ.

When a client executes a transaction, it extracts a concrete state of the key-value store by using its view to select a version for each key in the database. The concrete state extracted in this way takes the name of the *snapshot* of the transaction. In general, the process of determining the view

of a client, hence the snapshot in which such a client executes transactions, is non-deterministic. A snapshot $ss \in \textsc{Snapshot}$ consists of a function $ss : \textsc{Keys} \to \mathbb{N}$. Given a MKVS κ and a view V, we can always determine a snapshot in the obvious way.

Definition 3.4. Let κ be a MKVS, and V be a view. The snapshot of V in κ is defined as

$$\text{snapshot}(\kappa, V) = \lambda \mathsf{k}.\text{Value}(\kappa, \mathsf{k}, V).$$

AC: General Comment on this Section: it is too abstract. We should give either here or in the introduction an example of computation - the write skew program should be okay that helps the reader understanding what's going on. Also, it could be also good to illustrate the notions of execution tests and consistency models.

We assume that each client has its own stack, where data for performing local computations is stored. The set of thread-local stack variables is denoted by $\textsc{ThdVars} \triangleq \{\mathsf{x}, \mathsf{y}, \cdots\}$, while the set of transaction local variables is denoted by $\textsc{TxVars} \triangleq \{\mathsf{a}, \mathsf{b}, \cdots\}$. We use $\textsc{ThdStks}$ to range over thread-local stacks in the set $\textsc{ThdStks} \triangleq \textsc{ThdVars} \to \textsc{Val}$, and $\textsc{TxStks}$ to range over transaction-local stacks in the set $\textsc{TxStks} \triangleq \textsc{TxVars} \to \textsc{Val}$.

4 OPERATIONAL SEMANTICS

In this section we define a simple programming language for clients of programs interacting with a key-value store. In this semantics, clients can only interact with the key-value store by using transactions. We abstract from aborting transactions: rather than assuming that a transaction may abort due to a violation of the consistency guarantees given by the key-value store, we only allow the execution of a transaction when its effects are guaranteed to not violate the consistency model of the key-value store. This approach is equivalent to a setting where clients always restart a transaction after it aborts.

Programs are mapping from clients to commands. Each client is equipped with a set of local variables $\textsc{ThdVars} \triangleq \{\mathsf{x}, \mathsf{y}, \cdots\}$ that they can use for local computations. Each client is equipped with a stack $\sigma : \textsc{ThdVars} \to n$. The set of stacks is denoted by $\textsc{ThdStks}$. The state of a system consists of a configuration $C = (\kappa, \mathcal{U})$, and a partial mapping $\mathsf{Env} : \textsc{Clients} \rightharpoonup \textsc{ThdStks}$ from client identifiers to stacks, such that $\text{dom}(\mathcal{U}) = \text{dom}(\textsc{ThdStks})$.

Syntax of Programs. We assume a set of (primitive) transactional commands $t, t', \cdots$, which we leave unspecified. Each transactional command t is associated with a *state transformer* $\mathcal{S}_t \subseteq (\textsc{ThdStks} \times \textsc{Snapshot}) \times (\textsc{ThdStks} \times \textsc{Snapshot})$.

AC: This is slightly confusing: from the point of view of programs, the snapshot does not change while a transaction is executing.

We use the notation $(\sigma, h) \xrightarrow{t} (\sigma', h')$ in lieu of $(\sigma, \tau, ss), (\sigma', ss')) \in \mathcal{S}_t$. We also assume a set of primitive non-transactional commands $c, c', \cdots$ that can be executed by a command outside transactions. Each primitive non-transactional command c is associated with a state transformer $\mathcal{S}_c \subseteq (\textsc{ThdStks} \times \textsc{ThdStks})$, and again we adopt the notation $\sigma \xrightarrow{c} \sigma'$ in lieu of $(\sigma, \sigma') \in \mathcal{S}_c$. This definition ensures that non-transactional primitive commands do not access versions stored in the key-value stores.

Often, we will assume a language of expressions at the base of primitive (transactional and non-transactional) commands. This language is defined by the grammar below:

$$\mathsf{E} ::= v \mid \mathsf{x} \mid \mathsf{a} \mid \mathsf{E} + \mathsf{E} \mid \mathsf{E} \cdot \mathsf{E} \mid \cdots$$

The set of all expressions is denoted by $\textsc{Expr}$.

The set of primitive non-transactional and transactional commands we will use is given by

$$c ::= \mathsf{x} := \mathsf{E} \mid \mathsf{assume(E)}$$
$$t ::= c \mid [\mathsf{E}] := \mathsf{E} \mid \mathsf{a} := [\mathsf{E}]$$

Below we define the syntax of programs allowed by our language.

$$\mathsf{P} ::= cl : \mathsf{C} \,\|\, \mathsf{P} \,\|\, \mathsf{P}$$
$$\mathsf{C} ::= \mathbf{0} \mid c \mid \lceil \mathsf{T} \rceil \mid \mathsf{C};\mathsf{C} \mid \mathsf{C} + \mathsf{C} \mid \mathsf{C}^*$$
$$\mathsf{T} ::= \mathbf{0} \mid t \mid \mathsf{T};\mathsf{T} \mid \mathsf{T} + \mathsf{T} \mid \mathsf{T}^*$$

A program corresponds therefore to a set of commands executing in parallel. Each command in a command is annotated with a thread identifier cl.

AC: Following a discussion on the semantics, nobody likes client identifiers. It may simply be the case that we model a program as a set of commands, each of which is augmented with a MKVS and the views of clients on such MKVS.

Interpretation of Expressions and Primitive Commands. Expressions are going to be evaluated in values from n in the usual way.

$$\begin{aligned} [\![\cdot]\!]\cdot &: \textsc{Expr} \times \textsc{ThdStks} \to \mathbb{N} \\ [\![v]\!]\sigma &= v \\ [\![\mathsf{x}]\!]\sigma &= \sigma(\mathsf{x}) \\ [\![E_1 + E_2]\!]\sigma &\triangleq [\![E_1]\!]\sigma + [\![E_2]\!]\sigma \\ \vdots &= \vdots \end{aligned}$$

We now proceed to define the state transformers associated to transactional and non-transactional primitive commands. To specify the transformer of mutations, i.e. commands of the form $[\mathsf{E}_1] := \mathsf{E}_2$, and dereference, i.e. commands of the form $\mathsf{x} := [\mathsf{E}]$, we fix a bijection $\mathsf{keyOf} : n \to \textsc{Keys}$. This bijection exists because we are assuming that Keys is countably infinite. For the sake of clarity, we often avoid writing $\mathsf{keyOf}(n)$ and use the symbol k_n as a shorthand for $\mathsf{keyOf}(n)$ instead. For transactional and non-transactional primitive commands we have

$$(\sigma, ss) \xrightarrow{\mathsf{x} := \mathsf{E}} (\tau[\mathsf{x} \mapsto [\![\mathsf{E}]\!]\sigma], ss)$$
$$(\sigma, ss) \xrightarrow{\mathsf{x} := [\mathsf{E}]} (\tau[\mathsf{x} \mapsto ss(\mathsf{k}_{[\![\mathsf{E}]\!]\sigma})], ss)$$
$$(\sigma, ss) \xrightarrow{[\mathsf{E}_1] := \mathsf{E}_2} (\tau, ss[\mathsf{k}_{[\![\mathsf{E}_1]\!]\sigma} \mapsto [\![\mathsf{E}_2]\!]\sigma])$$
$$(\sigma, ss) \xrightarrow{\mathsf{assume(E)}} (\sigma, h) \text{ if } [\![E]\!]\sigma \neq 0$$

where we recall that, given an arbitrary function $f : X \to Y$, and two elements $x \in X, y \in Y$, then $f[x \mapsto y]$ denotes the function f' such that $f'(x) = y$, and $f'(x') = f(x')$ for all $x' \neq x$.

For non-transactional primitive commands we have

$$\sigma \overset{\mathsf{x}:=\mathsf{E}}{\rightsquigarrow_c} \sigma[\mathsf{x} \mapsto [\![\mathsf{E}]\!](\sigma)]$$
$$\sigma \overset{\mathsf{assume}(E)}{\rightsquigarrow_c} \sigma \text{ where} [\![\mathsf{E}]\!](\sigma) \neq 0$$

For transactional primitive commands t, we also defined a *fingerprint*, which denotes the contribution of the primitive command t in terms of operations that may be potentially observed by the external environment (i.e. other transactions). Formally, we assume a set of operations $\mathsf{Op} = \{\mathsf{RD\ k} : n, \mathsf{WR\ k} : n \mid \mathsf{k} \in \textsc{Keys} \wedge n \in n\}$. For technical reasons, we also define a special operation ε, called the empty operation, that is not included in Op, and we let $\mathsf{Op}_\varepsilon = \mathsf{Op} \cup \{\varepsilon\}$. Intuitively, ε corresponds to the fingerprint of transactional commands that do not result in an interaction with the key-value store.

Definition 4.1. For our language of primitive transactional commands, we define the function $\mathsf{Fprint} : \mathsf{C} \times \textsc{ThdStks} \times \textsc{Snapshot} \to \mathrm{Op}$ as follows:

$$\begin{array}{lcll} \mathsf{Fprint}(\mathsf{a} := \mathsf{E}, -, -) & \triangleq & \varepsilon & \\ \mathsf{Fprint}(\mathsf{a} := [\mathsf{E}], \sigma, ss) & \triangleq & \mathtt{RD}\ [n] : ss([n]) & \text{where } n := [\![\mathsf{E}]\!](\sigma)(\tau) \\ \mathsf{Fprint}([\mathsf{E}_1] := \mathsf{E}_2, \sigma, ss) & \triangleq & \mathtt{WR}\ [n_1] : n_2 & \text{where } n_i := [\![\mathsf{E}_i]\!](\sigma), i = 1, 2 \end{array}$$

Semantics of Transactions. For the operational semantics of transactions, judgements take the form $\langle \sigma, h, O, \mathsf{T} \rangle \to \langle \sigma', h', O', \mathsf{T}' \rangle$. Here $O, O' \subseteq \mathrm{Op}$ keep track of the fingerprint of the transaction being executed. Because transactions only work with a local snapshot of the key-value store, at this level MKVSs and views are not involved.

The fingerprint of a transaction corresponds to the set of all its interactions (i.e. read and write operations over keys) with the key-value store. To keep track of such interactions, we introduce an operator $\oplus$ that specifies the effects of adding a new operation inside a set of previously defined operations.

Definition 4.2. The operator $\oplus : \mathcal{P}(\mathrm{Op}) \times \mathrm{Op}_\varepsilon \to \mathcal{P}(\mathrm{Op})$ is defined as follows:

$$\begin{array}{lcl} O \oplus \varepsilon & = & O \\ O \oplus (\mathtt{RD}\ [n] : m) & = & \begin{cases} O \cup \{\mathtt{RD}\ [n] : m)\} & \Longleftarrow\ O \cap \{\mathtt{RD}\ [n] : m, \mathtt{WR}[n] : m \mid m \in \textsc{Val}\} = \emptyset \\ O & \Longleftarrow\ \text{otherwise} \end{cases} \\ O \oplus (\mathtt{WR}\ [n] : m) & = & (O \setminus \{\mathtt{WR}; [n] : m \mid m \in \textsc{Val}\}) \cup \{\mathtt{WR}\ [n] : m\} \end{array}$$

Intuitively, when executing a transaction we start with the empty fingerprint. Initially, the transaction has not interacted with the key-value store. Then, every time that a primitive command is executed by the transaction, we update its set of interactions by applying the operator $\oplus$ to the fingerprint of the primitive command executed. The definition of the operator $\oplus$ ensures that, for each key, only the first read preceding a write, and only the last write to that key, are recorded into the fingerprint of a transaction. This choice is motivated by the fact that we only focus on atomically visible transactions: keys are read from a snapshot of the database, and new version are written only at the moment the transaction commits.

Definition 4.3. A set of operations O is well-defined if,

$$\forall \mathrm{k} \in \textsc{Keys}. \forall O \in \{\mathtt{RD}, \mathtt{WR}\}. \forall n, m \in n.\ (O\ \mathrm{k} : n) \in O \wedge (O\ \mathrm{k} : m) \in O \implies n = m.$$

Lemma 4.4. *The set of well-defined operations is closed under $\oplus$:*

$$\forall O \in \mathcal{P}(\mathrm{Op})\ .\ \forall op \in \mathrm{Op}_\varepsilon.\ O \text{ is well-formed} \implies O \oplus op \text{ is well-formed.}$$

The rules of the operational semantics for transactions are given in Figure 6.

AC: I changed the language so that we have classical sequential composition and Kleene star. However, these choices lead to a more complicated operational semantics (6 rules against 4). The advantage of this syntax is that we do not have program variables anymore, which in turn makes proofs easier (especially when proving properties of recursive programs). However, I'm not a fan of this approach, since we are basically getting rid of an internal complication - i.e. technical details that will only play a role in proofs - by introducing external ones - i.e. that are plainly visible for the reader.

$$\frac{(\sigma, ss) \xrightarrow{t} (\sigma', h') \qquad \mathsf{Fprint}(t, \sigma, ss) = op}{\langle \sigma, ss, O, t\rangle \longrightarrow \langle \tau', ss', O \oplus op, \mathsf{T}\rangle}\ (\mathsf{T}\text{-}prim)$$

$$\frac{\langle \sigma, ss, O, \mathsf{T}_1\rangle \longrightarrow \langle \sigma', ss', O', \mathsf{T}_1'\rangle}{\langle \sigma, ss, O, \mathsf{T}_1; \mathsf{T}_2\rangle \longrightarrow \langle \sigma', ss', O', \mathsf{T}_1'; \mathsf{T}_2\rangle}\ (\mathsf{T}\text{-}comp\text{-}1) \qquad \frac{}{\langle \sigma, ss, O, \mathbf{0}; \mathsf{T}_2\rangle \longrightarrow \langle \sigma, ss, O, \mathsf{T}_2\rangle}\ (\mathsf{T}\text{-}comp\text{-}1)$$

$$\frac{}{\langle \sigma, ss, O, \mathsf{T}_1 + \mathsf{T}_2\rangle \longrightarrow \langle \sigma, ss, O, \mathsf{T}_1\rangle}\ (\mathsf{T}\text{-}choice\text{-}L) \qquad \frac{}{\langle \sigma, ss, O, \mathsf{T}_1 + \mathsf{T}_2\rangle \longrightarrow \langle \sigma, ss, O, \mathsf{T}_2\rangle}\ (\mathsf{T}\text{-}choice\text{-}R)$$

$$\frac{}{\langle \sigma, ss, O, \mathsf{T}^*\rangle \longrightarrow \langle \sigma, ss, O, \mathbf{0} + (\mathsf{T}; \mathsf{T}^*)\rangle}\ (\mathsf{T}\text{-}star)$$

Fig. 6. Rules for the operational semantics of transactions.

Semantics of Commands. Judgements in the semantics of commands take the form

$$\langle \kappa, \sigma, V : C\rangle \xrightarrow{\mathsf{ET}} \langle \kappa', \sigma', V' : \mathsf{C}'\rangle$$

Where ET is an execution test, i.e. a condition that must be satisfied in order for a client to execute a transaction safely. Execution Tests are introduced formally later in this Section, while we give different examples of execution tests in Section 5. There we also note that by tweaking the execution test used by the semantics, we capture different consistency models of key-value stores.

The main rule in the operational semantics of commands is the one corresponding to the execution of a transaction, i.e. $\mathsf{C} = [\mathsf{T}]$. For this rule, we must specify the following:

- How the execution of the transaction $[\mathsf{T}]$ affects a MKVS, and
- How the view of the client executing the command is changed, after the transaction has been executed.

Let us first discuss how the MKVS is changed when executing a transaction. Recall that a MKVS κ and the view V determine a snapshot via the function snapshot. By executing the transaction $[\mathsf{T}]$ using such a snapshot as its initial state, we obtain the fingerprint of the transaction as a set of operations O. We then choose a fresh transaction identifier t (i.e. that does not appear in κ already) that we associate with the execution of $[\mathsf{T}]$. Each read operation $\mathtt{RD}\ \mathrm{k} : n$, refers to the version v to which the view V pointed before the transaction $[\mathsf{T}]$ was executed: when we update the history heap, we must update the set of read transactions of such a version $\mathsf{Reads}(v)$ to include the transaction t. For write operations of the form $\mathtt{WR}\ \mathrm{k} : m$, we create a new version $v = (m, t, \emptyset)$ and we append it at the tail of $\kappa(\mathrm{k})$.

Definition 4.5. The function $\mathsf{UpdateMKVS}_t$ is defined as follows:

$$\begin{array}{lcl}
\mathsf{UpdateMKVS}_t(\kappa, V, \{\mathtt{RD}\ \mathrm{k} : n\}) & = & \mathsf{let}\ (m, t', \mathcal{T}) = \mathsf{snapshot}(\kappa, V)(\mathrm{k})\ \mathsf{in} \\
 & & \kappa\Big[\mathrm{k} \mapsto \Big(\kappa(\mathrm{k})[V(\mathrm{k}) \mapsto (m, t', \mathcal{T} \cup \{t\})]\Big)\Big] \\
\mathsf{HHupdate}_t(\kappa, _, \{\mathtt{WR}\ \mathrm{k} : n\}) & = & \kappa\Big[[n] \mapsto (\kappa(\mathrm{k}) \cdot \langle n, t, \emptyset\rangle)\Big] \\
\mathsf{UpdateMKVS}_t(\kappa, V, O_1 \cup O_2) & = & \mathsf{UpdateMKVS}_t(\mathsf{HHupdate}_t(\kappa, V, O_1), V, O_2)
\end{array}$$

Lemma 4.6. *The function* $\mathsf{UpdateMKVS}_t$ *is well-defined over well-formed MKVS, sets of operations and views. If* κ *is a well-formed history heap,* V *is a well-formed view with respect to* κ*,* O *is a well-formed set of operations and* t *does not appear in* κ*, then* $\mathsf{UpdateMKVS}_t(\kappa, O, V)$ *is uniquely determined and equivalent to a well-formed MKVS.*

AC: The definition above should be well-formed, but we must prove this.

Next, we discuss how the view of a command is changed after executing a transaction. In this case, we only update the view of the locations that have been written by a transaction to be up-to-date with the version installed by the transaction that has been executed.

Definition 4.7. The function UpdateView is defined below:

$$\begin{array}{lcl} \mathsf{UpdateView}(\kappa, V, \{\mathtt{RD\ k} : n\}) & = & V \\ \mathsf{UpdateView}(\kappa, V, \{\mathtt{WR\ k} : n\}) & = & V[\mathrm{k} \mapsto (|\kappa(\mathrm{k})| + 1)] \\ \mathsf{UpdateView}(\kappa, V, \mathcal{O}_1 \cup \mathcal{O}_2) & = & \mathsf{UpdateView}(\kappa, \mathsf{UpdateView}(\kappa, V, \mathcal{O}_1), \mathcal{O}_2) \end{array}$$

Lemma 4.8. *The function* UpdateView *is well-defined. Furthermore, let* $\kappa, V, \mathcal{O}$ *and* t *be a well-formed MKVS, view, set of operations, and a transction identifier that does not appear in* κ*, respectively. Then* $\mathsf{UpdateView}(\kappa, V, \mathcal{O})$ *is well-formed with respect to* $\mathsf{UpdateMKVS}_t(\kappa, V, \mathcal{O})$.

AC: The paragraph below should probably go when discussing the rules of the semantics

Note that the way in which MKVSs and views are updated ensure the following: • a client always reads its own preceding writes; • clients always read from an increasingly up-to-date state of the database; • the order in which clients update a key k is consistent with the order of the versions for such keys in the MKVS; • writes take place after reads on which they depend.

AC: (need to check again what this exactly means, but I'm working from home and I cannot access the session guarantee paper)

Assuming that commands executed by clients represent are wrapped within a single session, these four bullets correspond to the strong session guarantees introduced by Terry et al. in [**?**].

Finally, we introduce the notion of *execution tests*, that will be used in the semantics to determine whether a client can safely execute a transaction.

Definition 4.9. An execution test is a set ET of tuples of the form $(\kappa, V, \mathcal{O}, V')$.

Given an execution test ET and a tuple $(\kappa, V, \mathcal{O}, V') \in \mathsf{ET}$, we often write $\mathsf{ET} \vdash (\kappa, V) \triangleright \mathcal{O} : V'$. Roughly speaking $\mathsf{ET} \vdash (\kappa, V) \triangleright \mathcal{O} : V'$ means that, under the execution test ET, in the MKVS κ a client with view V can safely commit a transaction with fingerprint $\mathcal{O}$, and push its view to V'.

The rules of the operational semantics of commands are given in Figure 7. The main rule for executing commands is the one that models the execution of one transaction. In the rule below we compare views according to a partial order $\sqsubseteq$, which is defined to be the point-wise comparison of values of views at single locations: $V_1 \sqsubseteq V_2 \triangleq \forall \mathrm{k} \in \textsc{Keys}. V_1(\mathrm{k}) \leq V_2(\mathrm{k})$.

Semantics of Programs.

$$\frac{V \sqsubseteq V' \qquad ss = \mathsf{snapshot}(\kappa, V') \qquad \langle h, \sigma, \emptyset, \mathsf{T}\rangle \rightarrow^* \langle \_, \sigma', \mathcal{O}, \mathbf{0}\rangle \qquad \mathsf{UpdateView}(\kappa, V', \mathcal{O}) \sqsubseteq V'' \qquad \mathsf{ET} \vdash (\kappa, V') \triangleright \mathcal{O} : V'' \qquad t \text{ fresh in } \kappa}{\langle \kappa, \sigma, V, [\mathsf{T}]\rangle \rightarrow_{\mathsf{ET}} \langle \mathsf{UpdateMKVS}_t(\kappa, V', \mathcal{O}), \sigma', V'', \mathbf{0}\rangle} \; (C\text{–}Tx)$$

$$\frac{\sigma \rightsquigarrow_c \sigma'}{\langle \kappa, \sigma, V, c\rangle \rightarrow_{\mathsf{ET}} \langle \kappa, \sigma', V, \mathbf{0}\rangle} \; (prim\text{–}c)$$

$$\frac{\langle \kappa, \sigma, V, \mathsf{C}_1\rangle \rightarrow_{\mathsf{ET}} \langle \kappa', \sigma', V', \mathsf{C}_1'\rangle}{\langle \kappa, \sigma, V, \mathsf{C}_1; \mathsf{C}_2\rangle \rightarrow_{\mathsf{ET}} \langle \kappa', \sigma', V', \mathsf{C}_1'; \mathsf{C}_2\rangle} \; (C\text{–}Seq\text{–}T) \qquad \frac{}{\langle \kappa, \sigma, V, \mathbf{0}; \mathsf{C}_2\rangle \rightarrow_{\mathsf{ET}} \langle \kappa, \sigma, V, \mathsf{C}_2\rangle} \; (C\text{–}Seq\text{–}nil)$$

$$\frac{}{\langle \kappa, \sigma, V, \mathsf{C}_1 + \mathsf{C}_2\rangle \rightarrow_{\mathsf{ET}} \langle \kappa, \sigma, V, \mathsf{C}_1\rangle} \; (C\text{–}Choice\text{–}L) \qquad \frac{}{\langle \kappa, \sigma, V, \mathsf{C}_1 + \mathsf{C}_2\rangle \rightarrow_{\mathsf{ET}} \langle \kappa, \sigma, V, \mathsf{C}_2\rangle} \; (C\text{–}Choice\text{–}R)$$

$$\frac{}{\langle \kappa, \sigma, V, \mathsf{C}^*\rangle \rightarrow \langle \kappa, \sigma, V, \mathbf{0} + (\mathsf{C}; \mathsf{C}^*)\rangle} \; (C\text{–}star)$$

Fig. 7. Operational Semantics of Commands.

AC: I changed (again) the definition of consistency models. I'm leaving the old version in the comment-box for future reference, though it is not clear whether the definition below is expressive enough to capture some consistency models (among others, snapshot isolation.)
In this sense, a consistency model specification CM consists of a set of quadruples $(\kappa, V, \mathcal{V}, O)$, where

- **κ is a history heap, corresponding to a state of the database**
- **V is the view of the history heap of the thread that wants to execute a transaction,**
- **$\mathcal{V}$ is the multi-set of views of the external environment (i.e. other threads that may be interacting with the database),**
- **O is the fingerprint of the transaction to be executed.**

Therefore, tuples of the form $(\kappa, V, \mathcal{V}, O) \in CM$ describe which operations a thread is allowed to execute in a single transaction, without conflicting with other threads. We often write $(\kappa, V, \mathcal{V}) \triangleright_{CM} O$ in lieu of $(\kappa, V, \mathcal{V}, O) \in CM$.

The last step we take is that of defining the operational semantics of programs. To this end, we associate a thread stack and view to each parallel component (or thread) of the program. In practice, we define an environment Env to be a partial mapping $\mathsf{Env} : \textsc{Clients} \rightharpoonup (\mathsf{ThreadStacks} \times \mathsf{Views})$

AC: Set of Views and thread identifiers never introduced before.

Given two environments, $\mathsf{Env}_1, \mathsf{Env}_2$, we let $\mathsf{Env}_1 \uplus \mathsf{Env}_2$ to be defined if $\mathrm{dom}(\mathsf{Env}_1) \cap \mathrm{dom}(\mathsf{Env}_2) = \emptyset$, in which case we have

$$(\mathsf{Env}_1 \uplus \mathsf{Env}_2)(cl) = \begin{cases} \mathsf{Env}_1(cl) & \Longleftarrow \; cl \in \mathrm{dom}(\mathsf{Env}_1) \\ \mathsf{Env}_2(cl) & \Longleftarrow \; cl \in \mathrm{dom}(\mathsf{Env}_2) \\ \text{undefined} & \Longleftarrow \; \text{otherwise} \end{cases}$$

Judgements in the operational semantics take the form

$$\frac{}{\langle \kappa, \mathsf{Env}, \mathsf{P}\rangle \xrightarrow{CM} \langle \kappa', \mathsf{Env}', \mathsf{P}'\rangle}$$

$$\frac{\mathsf{Env} = [cl \mapsto (\sigma, V)] \qquad \langle \kappa, \sigma, V, \mathsf{C}\rangle \rightarrow_{\mathsf{ET}} \langle \kappa', \sigma', V', \mathsf{C}'\rangle \qquad \mathsf{Env}' = [cl \mapsto (\sigma', V')]}{\langle \kappa, \mathsf{Env}, cl : \mathsf{C}\rangle \rightarrow_{\mathsf{ET}} \langle \kappa', \mathsf{Env}', cl : \mathsf{C}'\rangle} \; (P\text{–}thd\text{–}exec)$$

$$\frac{\langle \kappa, \mathsf{Env}_1, \mathsf{P}_1\rangle \rightarrow_{\mathsf{ET}} \langle \kappa', \mathsf{Env}_1', \mathsf{P}_1'\rangle \qquad (\mathsf{Env}_1 \uplus \mathsf{Env}_2)\downarrow}{\langle \kappa, \mathsf{Env}_1 \uplus \mathsf{Env}_2, \mathsf{P}_1 \parallel \mathsf{P}_2\rangle \rightarrow_{\mathsf{ET}} \langle \kappa', \mathsf{Env}_1' \uplus \mathsf{Env}_2, \mathsf{P}_1' \parallel \mathsf{P}_2\rangle} \; (P\text{–}par\text{–}L)$$

$$\frac{\langle \kappa, \mathsf{Env}_2, \mathsf{P}_2\rangle \rightarrow_{\mathsf{ET}} \langle \kappa', \mathsf{Env}_2', \mathsf{P}_2'\rangle \qquad (\mathsf{Env}_1 \uplus \mathsf{Env}_2)\downarrow}{\langle \kappa, \mathsf{Env}_1 \uplus \mathsf{Env}_2, \mathsf{P}_1 \parallel \mathsf{P}_2\rangle \rightarrow_{\mathsf{ET}} \langle \kappa', \mathsf{Env}_1' \uplus \mathsf{Env}_2, \mathsf{P}_1' \parallel \mathsf{P}_2\rangle} \; (P\text{–}par\text{–}R)$$

Fig. 8. Semantics of Programs.

AC: The changes made to consistency model specifications simplify the semantics a lot. The rule of parallel composition becomes standard, and the only rule that now needs to take the consistency model into account should be the execution of a transaction in the semantics of commands.

The rules for the operational semantics of programs are standard. We have three rules, depicted in Figure **??**. One rule lifts transitions of commands to single-threaded programs. The remaining two rules model the parallel composition of programs in a standard, interleaving fashion.

4.1 Operational Semantics: Example

To conclude our discussion on the operational semantics, we show in detail one possible computation of a program consisting of two transactions executing in parallel. The program P_1 that we consider is illustrated below:

$$\begin{bmatrix}[k_x] := 1;\\ [k_y] := 1;\end{bmatrix} \Bigg\| \begin{bmatrix}\mathsf{a} := [\mathsf{k}_x];\\ \mathsf{b} := [\mathsf{k}_y];\\ \mathtt{if}\,(\mathsf{a} = 1 \wedge \mathsf{b} = 0)\{\\ \quad \mathtt{ret} := \text{☹}\}\end{bmatrix}$$

Note also that we omitted thread identifiers for commands in P_1. In general, we often write $\mathsf{C}_1 \parallel \cdots \parallel \mathsf{C}_n$ as a shorthand for $cl_1 : \mathsf{C}_1 \parallel \cdots \parallel cl_n : \mathsf{C}_n$.

The program P_1 contains a special symbol ☹ which can be returned by the transaction of cl_2. In practice, this can be an arbitrary fixed value that can be returned by a transaction. However, by using the symbol ☹ as the return value for the transaction, we want to emphasize the fact that the transaction exhibited some undesirable behaviour. In this case, the undesirable behaviour corresponds to the transaction on the right hand side observing only one of the updates that can be executed by the transaction on the left hand side.

Although this behaviour clearly violates the constraints that transactions should be executed atomically, we show that if no restrictions are placed on the consistency model specification, it is possible to execute P_1 and reach a configuration where the transaction executed by cl_2 returns ☹. To this end, let $\mathsf{ET}_\top$ be the most permissive execution test, i.e. the one such that $\mathsf{ET}_\top(\kappa, V) \triangleright O : V'$ for any κ, V, O, V' such that $\mathsf{UpdateView}(\kappa, V, O) \sqsubseteq V'$.

AC: The condition on V' is not really needed.

Before showing the computation of P_1 that leads to the transaction of cl_2 to return ☹, we need to introduce some definitions and notation. The initial state in which P_1 is executed is the one in which each location has an initial version written by some initialisation transaction t_0, the view V_0 of each thread points to the initial version of each key, i.e. $V_0 = [\mathsf{k}_1 \mapsto 1, \mathsf{k}_2 \mapsto 1]$, and all the stacks are initialised to 0. We assume that the only key in the database are $\mathsf{k}_1, \mathsf{k}_2$, and that the only

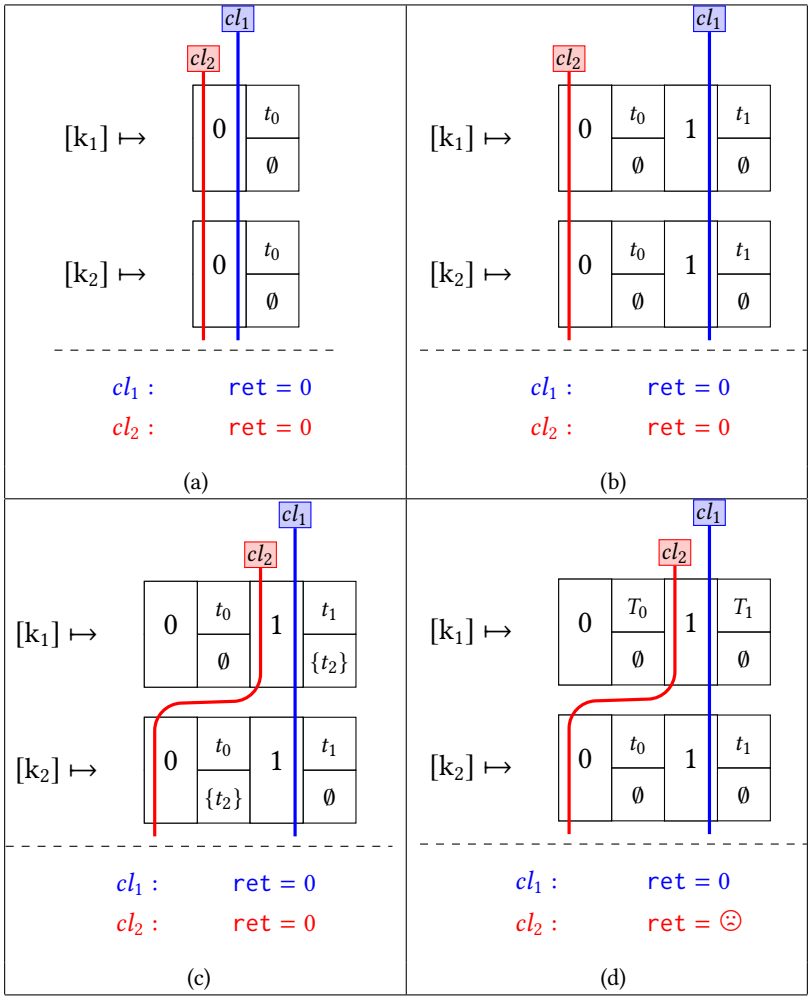


Fig. 9. Graphical Representation of configurations obtained through the execution of P_1.

variable in the local stack of clients is ret. The initial state is then given by $\mathcal{S}_0 = (\kappa_0, \mathsf{Env}_0)$, where $\kappa_0(\mathrm{k}_1) = \kappa_0(\mathrm{k}_2) = (0, t_0, \emptyset)$, $\mathsf{Env}_0(cl_1) = \mathsf{Env}_0(cl_2) = ([\mathtt{ret} \mapsto 0], V_0)$, The initial state is represented graphically in Figure 9(a).

We are now ready to show how to derive the computation of P_1 that violates atomic visibility. First, we reduce the transaction of cl_1. According to rule $(C-Tx)$, we need to perform the following steps:

- shift the view V_0 for client cl_1, in κ_0, to the right of an arbitrary amount of positions for each key, and obtain a view $V_{\mathsf{pre}} : V_0 \sqsubseteq V_{\mathsf{pre}}$. Because κ_0 contains only one version for each location, here the only possibility is that $V_{\mathsf{pre}} = V_0$.
- Determine the snapshot $ss = \mathsf{snapshot}(\kappa_0, V_{\mathsf{pre}})$ in which thread cl_1 executes the transaction. In this case, we have that $ss = [\mathrm{k}_1 \mapsto 0, \mathrm{k}_2 \mapsto 0]$.

- Determining the fingerprint O of the transaction executed by cl_1, whose code is

$$\begin{bmatrix}[k_x] := 1;\\ [k_y] := 1;\end{bmatrix}$$

This amounts to execute the transaction in isolation from the external environment, using the rules in the operational semantics for transactions (Figure 6). Because this execution must match the premiss of Rule $(C - Tx)$, The code is run using ss as the initial snapshot, σ_0 as the initial thread stack, and $\emptyset$ as the initial fingerprint. We only need to apply Rule $(\mathsf{T} - prim)$ twice, in which case we obtain

$$\begin{array}{rcrcl} & & \left\langle ss_0, \_, \emptyset, \begin{array}{l}[\mathrm{k}_1] := 1;\\ [\mathrm{k}_2] := 1\end{array} \right\rangle & \rightarrow \\ \rightarrow & \left\langle ss_0[key k_1 \mapsto 1], \_, (\emptyset \oplus \mathtt{WR}\ \mathrm{k}_1 : 1), [\mathrm{k}_2] := 1 \right\rangle & = \\ = & \langle ss_0[[\mathrm{k}_2] \mapsto 1], \_, \{\mathtt{WR}\ \mathrm{k}_1 : 1\}, [\mathrm{k}_2] := 1\rangle & \rightarrow \\ \rightarrow & \left\langle ss_0[\mathrm{k}_1 \mapsto_1, \mathrm{k}_2 \mapsto 1], \_, (\{\mathtt{WR}\ \mathrm{k}_1 : 1\} \oplus (\mathtt{WR}\ \mathrm{k}_2 : 1)), \mathbf{0} \right\rangle & = \\ = & \langle \_, \_, \{\mathtt{WR}\ \mathrm{k}_1 : 1, \mathtt{WR}\ \mathrm{k}_2 : 1\}, \mathbf{0}\rangle & \end{array} \tag{2}$$

Therefore, we conclude $O = \{\mathtt{WR}\ \mathrm{k}_1 : 1, \mathtt{WR}\ \mathrm{k}_2 : 1\}$.

- Determine the MKVS and view $\kappa_1, V_{\text{post}}$ obtained by applying the functions $\mathsf{UpdateMKVS}_{t_1}(\kappa_0, V_{\text{pre}}, O)$ and $\mathsf{UpdateView}(\kappa_0, V_{\text{pre}}, mathcal O)$, respectively. Here t_1 is a fresh transaction identifier. The resulting MKVS κ_1 and view V_{post} are depicted in Figure 9(b); there the view V_{post} is labelled with cl_1.
- Determine a view $V_1 : V_{\text{post}} \sqsubseteq V_1$, such that $\mathsf{ET}_\top \vdash (\kappa_0, V_{\text{pre}}) \triangleright O : V_1$. The only possibility here is by choosing $V_1 = V_{\text{post}}$,
- Combine all the steps above, and infer the transition $\langle \mathcal{S}_0, \mathsf{P}_1\rangle \rightarrow \mathsf{ET}_\top\langle \mathcal{S}_1, \mathsf{P}_1'\rangle$, where $\mathcal{S}_1$ is depicted in Figure 10(b) and

$$\mathsf{P}_1' = \mathbf{0} \;\Big\|\; \begin{bmatrix}\mathtt{a} := [\mathrm{k}_1];\\ \mathtt{b} := [\mathrm{k}_2];\\ \mathtt{if}\,(\mathtt{a} = 1 \wedge \mathtt{b} = 0)\{\\ \quad \mathtt{ret} := \text{☹}\end{bmatrix}$$

Next, client cl_2 executes its own transaction using a similar pattern. Note that in $\mathcal{S}_1$, the view of thread cl_2 is $V_0 = [\mathrm{k}_1 \mapsto 1, \mathrm{k}_2 \mapsto 1]$. Because the MKVS of $\mathcal{S}_1$ contains two versions for each location, there are exactly four views $V' : V_0 \sqsubseteq V'$ that can be chosen prior to executing the code of the transaction

$$\begin{bmatrix}\mathtt{a} := [\mathrm{k}_1];\\ \mathtt{b} := [\mathrm{k}_2];\\ \mathtt{if}\,(\mathtt{a} = 1 \wedge \mathtt{b} = 0)\{\\ \quad \mathtt{ret} := \text{☹}\end{bmatrix}$$

In practice, for this example we choose the view $V' = [\mathrm{k}_1 \mapsto 2, \mathrm{k}_2 \mapsto 1]$, which points to the last version for location k_1, and to the initial version for k_2. This view is the one labelled with cl_2 in Figure 9(c). Executing the transaction results in the transition $\langle \mathcal{S}_1, \mathsf{P}_1'\rangle \rightarrow_{\mathsf{ET}_\top} \langle \mathcal{S}_2, \mathbf{0} \parallel \mathbf{0}\rangle$, where $\mathcal{S}_2$ is the state depicted in Figure 9(d). Note that in $\mathcal{S}_2$ the transaction t_2 executed by cl_2 has read both the initial version for key k_2, and the latest version for k_1. This caused such a transaction to pass the check at its third line of code, so that the value `ret` for thread cl_2 is updated to ☹ (see Figure 9(d)).

5 EXAMPLES OF CONSISTENCY MODELS

AC: This Section is going to become heavy in pictures, which should be organised into figures.

In this Section we present different consistency models specifications. For each of them, we give:

- an informal definition, describing the consistency guarantees that schedules of the database should have in plain English,
- examples of litmus tests that, when executed, give rise to the anomalies that should be forbidden from the consistency model,
- a formal consistency model specification, in the style described in §4,
- an explanation of why the consistency model forbids the litmus tests to exhibit the anomaly that should be forbidden.

Later on in the paper, we will show how to compare our consistency models specifications with those already existing in the literature.

AC: There is still a long-way to go before proving correspondence with axiomatic specifications, but this should be mentioned here.

5.1 Read Atomic

Read atomic [Bailis et al. 2014] is the weakest consistency model among those that enjoy *atomic visibility* [Cerone et al. 2015]. It requires transactions to read an atomic snapshot of the database. It also requires transactions to never observe the partial effects of other transactions. This is also known as the *all-or-nothing* property: A transaction observes either none or all the updates performed by another transaction.

One litmus test that should **not** be failed in RAMP consists of the program P_1 from §4.1, which we already observed to produce a violation of atomic visibility if no constraints on the consistency model are placed.

AC: not be failed. Double negation. Bad English.

Intuitively, in such a program, the violation of atomic visibility happened because we allowed to execute the transaction

$$\mathsf{T}_1^2 = \begin{bmatrix} \mathtt{a} := [k_x]; \\ \mathtt{b} := [k_y]; \\ \mathtt{if}\,(\mathtt{a} = 1 \wedge \mathtt{b} = 0)\{ \\ \quad \mathtt{ret} := \text{☹} \end{bmatrix}$$

in the client-local configuration of C' relative to cl_2, which is obtained by removing all the information about cl_1 (view and stack) in Figure 9(c).

To avoid transactions to only observe the partial effects of other transactions, we must ensure that transactional code cannot be executed by a client whose views is up-to-date with respect to some transaction t for some location k_1, but not for some other location k_2. This leads to the following definition.

Definition 5.1. Let V be view, κ be a MKVS, t be a transaction identifier. We say that V *sees* transaction t in κ, written $\mathsf{Visible}(t, \kappa, V)$, iff

$$\forall \mathrm{k}.\ \forall i = 0, \cdots, |\kappa([k_x])|.\ \mathsf{Write}(\kappa(\mathrm{k}, i)) = t \implies i \leq V(\mathrm{k}).$$

AC: In English: the view is up-to-date with respect to all the updates performed by transaction t.

We say that the view V is *consistent* with respect to atomic visibility and the history heap κ, written $\mathsf{Atomic}(\kappa, V)$, if

$$\forall t.\ (\exists\ \mathrm{k}.\exists i \leq V(\mathrm{k}).\ \mathsf{Write}(\kappa(\mathrm{k}, i)) = t) \implies \mathsf{Visible}(t, \kappa, V).$$

AC: In English: if the view V is up-to-date with some of the updates performed by t, then it must be up-to-date with all the updates performed by t. This is the all-or-nothing property.

The execution test $\mathsf{ET}_{\mathsf{RA}}$ is defined as the smallest set such that

$$\mathsf{Atomic}(\kappa, V) \implies \mathsf{ET}_{\mathsf{RA}} \vdash (\kappa, V) \triangleright \_ : \_$$

AC: In English: Before executing a transaction, either you observe all or none the updates of all other transactions. We may strengthen the consistency model and require that the same property must be satisfied at the end as well, though this is not strictly necessary. In this case the check becomes:

$$\mathsf{Atomic}(\kappa, V) \wedge \mathsf{Atomic}(\kappa, V') \wedge \mathsf{UpdateView}(\kappa, V, O) \sqsubseteq V' \implies (\kappa, V) \triangleright_{\mathsf{RA}} O : V'.$$

Suppose that we execute the program P_1 using the execution test $\mathsf{ET}_{\mathsf{RA}}$. We can proceed as in Section 4.1 to infer the transition $\langle \mathcal{S}_0, \mathsf{P}_1 \rangle \rightarrow_{\mathsf{ET}_{\mathsf{RA}}} \langle \mathcal{S}_1, \mathsf{P}'_1 \rangle$, where we recall that $\mathcal{S}_0$, $\mathcal{S}_1$ are depicted in Figure **??**(a), 9(b), respectively.

It is immediate to observe that the only way in which the execution of transaction $\lceil \mathsf{T} \rceil$ from cl_2 in P'_1 can return value ☹ is the following:

- first, push the view V of client cl_2 in the configuration C_1 of Figure 9(b) to observe the update of key k_1, but not the update of k_2. This view is the one labelled with cl_2 in Figure 9(c), and we refer to it as V';
- then, execute the transaction $\lceil \mathsf{T} \rceil$ in cl_2.

However, this second step cannot be performed if the execution test $\mathsf{ET}_{\mathsf{RA}}$ is used. In fact, the view V' is not atomic, according to Definition 5.1, and therefore, client cl_2 cannot execute transaction $\lceil \mathsf{T} \rceil$ using the snapshot extracted from view V'. A consequence of this fact is that, under the execution test $\mathsf{ET}_{\mathsf{RA}}$ there exists no execution of P_1 in which the transaction $\lceil \mathsf{T} \rceil$ returns value ☹.

One may wonder whether it is the case that an execution of a program may get stuck because of a ill-defined execution test. For example, for a given execution test ET, it may be possible to reach a state $\mathcal{S}$ of the system, with MKVS κ and view V for client cl V, such that for any $V' : V \sqsubseteq V'$ we have that $\mathsf{ET}_{\mathsf{RA}} \not\vdash (\kappa, V') \triangleright \_ : \_$. We show that this is not the case for the execution test $\mathsf{ET}_{\mathsf{RA}}$.

Proposition 5.2. *The execution test* $\mathsf{ET}_{\mathsf{RA}}$ does not hinder progress*:*

AC: This does not mean that progress is ensured. A transaction may very well not terminate. Rather, this says that the inability of a program to execute a transaction does not depend on the execution test itself.

for any MKVS ss, views V, and set of operations O, there exist two views $V' : V \sqsubseteq V'$, $V'' : \mathsf{UpdateView}(\kappa, V', O) \sqsubseteq V''$ such that $\mathsf{ET}_{\mathsf{RA}} \vdash (\kappa, V) \triangleright O : V'$.

Proof Sketch. For a given MKVS κ, let $\mathsf{up-to-date}(\kappa) = \lambda \mathrm{k}.|\kappa|$ be the view that points to the most recent version of each object. It is immediate to note that $\mathsf{Atomic}(\kappa, \mathsf{up-to-date}(\kappa))$, and therefore $\mathsf{ET}_{\mathsf{RA}} \vdash (\kappa, \mathsf{up-to-date}(\kappa, V)) \triangleright \_ : \_$. □

5.2 Causal Consistency

The next consistency model that we consider is *transactional causal consistency* [Lloyd et al. 2011]. Intuitively, in this consistency model transactions must be ensured that versions read by transactions

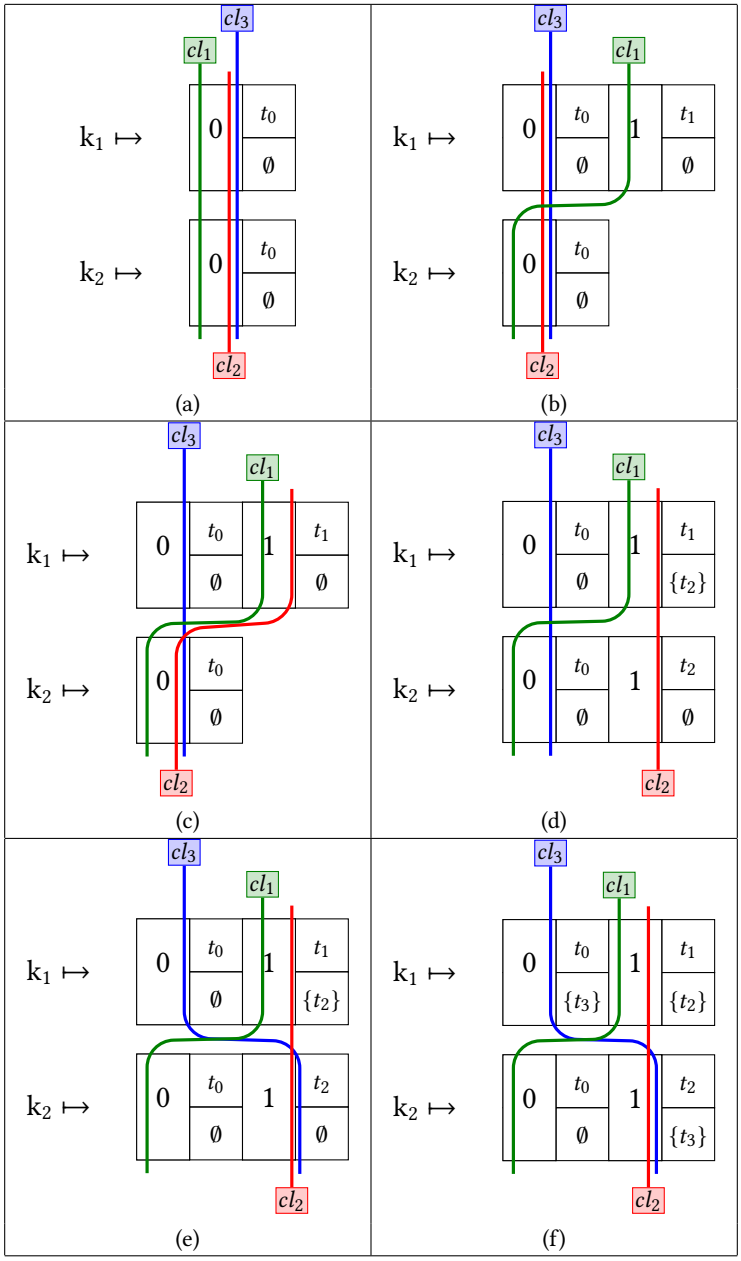


Fig. 10. Configurations obtained in a execution of P_2.

are closed with respect to causal dependencies. Consider for example the following program:

$$P_2 := \left[[k_x] := 1;\right] \left\| \left[\begin{array}{l} a := [k_x]; \\ [k_y] := a; \end{array}\right] \right\| \left[\begin{array}{l} a := [k_x]; \\ b := [k_y]; \\ \texttt{if}\,(a = 0 \wedge b = 1)\{ \\ \quad \texttt{ret} := \text{☹}\} \end{array}\right]$$

For the sake of simplicity, we label the code of the three transactions above ast $\mathsf{T}_1, \mathsf{T}_2, \mathsf{T}_3$, from left to right. It is easy to see that, if no constraints are placed on the execution test (i.e if the execution test $\mathsf{ET}_\top$ from Section **??** is used, then we can have $\lceil \mathsf{T}_3 \rceil$ return ☹. The same is true even if the execution test $\mathsf{ET}_{\mathsf{RA}}$ is used. Informally, the return of value ☹ by $\lceil \mathsf{T}_3 \rceil$ can be obtained from the execution outlined below.

- The initial configuration of this execution is depicted in Figure 10(a).
- $\lceil \mathsf{T}_1 \rceil$ executes with the initial view, which points to the initial (and only) version for each key; after this transaction is executed, a new version $\langle 1, T_1, \emptyset \rangle$ is appended at the end of $\kappa(k_x)$. The resulting configuration is depicted in Figure 10(b).
- next, cl_2 updates its view as to see the version of k_2 installed by cl_1, after which it proceeds to execute $\lceil \mathsf{T}_2 \rceil$. This view is the one labelled with cl_2 in Figure 10(c); according to this view, the snapshot under which $\lceil \mathsf{T}_2 \rceil$ is executed is $[\mathrm{k}_1 \mapsto 1, \mathrm{k}_2 \mapsto 0]$. The code $\lceil \mathsf{T}_2 \rceil$ copies the value of k_1 into k_2, hence after the transaction is executed, a new version for k_2 with value 1 will be created. The configuration obtained after the execution of $\lceil \mathsf{T}_2 \rceil$ is given in Figure 10(d).
- Finally, client cl_3 updates its view to observe the update of location k_2, but not the update of location k_1, before executing transaction $\lceil \mathsf{T}_3 \rceil$. This view is the one labelled cl_3 in Figure 10(e). The transaction $\lceil \mathsf{T}_3 \rceil$ is executed using the snapshot $[\mathrm{k}_1 \mapsto 0, \mathrm{k}_2 \mapsto 1]$, which causes the local variable `ret` of cl_3 to be set to ☹. The configuration obtained after $\lceil \mathsf{T}_3 \rceil$ is executed is depicted in Figure 10(f).

The reason why the execution outlined above is because, in the last step, thread cl_3 executed transaction $\lceil \mathsf{T}_3 \rceil$ in a state where its view observes the update to location k_2, but not the update to location k_1. However, latter directly depends on the former, in the sense of Definition 3.2. In other words, the execution of transaction $\lceil \mathsf{T}_3 \rceil$ resulted in a violation of causality: the update of k_2 is observed, but not the update of k_1 on which it depends.

To specify formally transactional causal consistency, we define inductively the set of views that are consistent with respect to a history heap κ.

AC: Note to self: why not going for a coinductive definition? The set of causally consistent views is the largest set such that whenever V is causally consistent w.r.t. κ, then for each version $v, v' \in \kappa$, such that v' depends on v, then if v' is included in V also v is included in V.

Intuitively, the definition below models the fact that, if we start from a causally consistent view, and we wish to update the view for some location cl_2 to include a newer version v, then we must ensure that all the versions on which v directly depends in κ are already included in the view.

Definition 5.3. Let κ be a MKVS. The set of views that are *causally consistent* with respect to κ, $\mathsf{CCViews}(\kappa)$, is defined as the smallest set such that:

(1) $V_0 := \lambda \mathrm{k}.0 \in \mathsf{CCViews}(\kappa)$,
(2) let $V \in \mathsf{CCViews}(\kappa)$, $V' = V[\mathrm{k} \mapsto (V(x) + 1)]$, for some k such that $V(\mathrm{k}) < |\kappa(\mathrm{k})|$. Suppose that

$$\forall \mathrm{k}'.\ \forall j = 0, \cdots, |\kappa(\mathrm{k}')|.\ \kappa(\mathrm{k}', j) \xrightarrow{\text{ddep}} \mathsf{versionOf}(\kappa, \mathrm{k}, V') \implies j \leq V(\mathrm{k}').$$

Then $V' \in \mathsf{CCViews}(\kappa)$.

Example 5.4.

AC: Note to self: I got this example and the definition wrong several times before getting them right. Which means that inductive definition of causally dependent views is not really that intuitive after all...

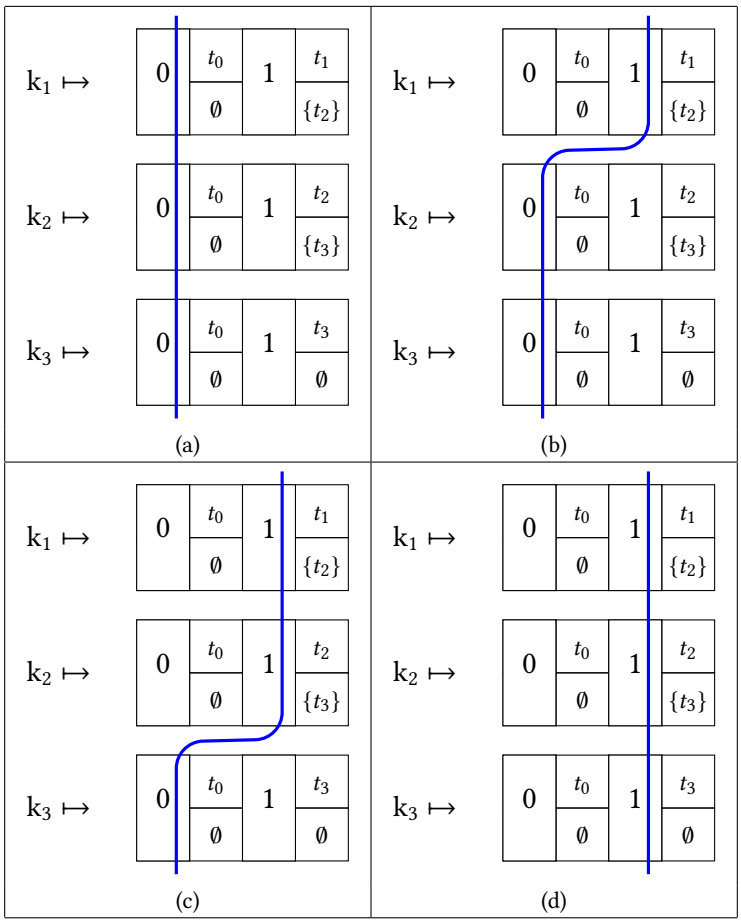


Fig. 11. Building a causally consistent view.

Consider the MKVS κ, depicted in Figure 11(a) - ignore for the moment the view in the Figure. We want to find a view V that points to the latest version of key k_3 in κ, i.e. $\kappa(k_3, 2)$; that is, we require that $V(k_3) = 2$. Furthermore, we want the view V to be causally consistent.

We construct the view V incrementally, as outlined below. Note that for the MKVS κ, we have that $\kappa(k_3, 2) \xrightarrow{\text{ddep}} \kappa(keyk_2, 2)$, and $\kappa(k_2, 2) \xrightarrow{\text{ddep}} \kappa(k_1, 2)$.

We start from the initial view V_0, $V_0(k\_) = 0$. This view is depicted in Figure 11(a), and it is causally consistent by definition. However, it does not point to the last version for k_3: in fact, $V_0(k_3) = 1$. As a first try, one could immediately consider a view V' where the value for location k_3 is updated to 1, that is $V' = V_0[k_3 \mapsto 2]$. However, in this case the view V' is not causally consistent. In fact, we have that $\text{versionOf}(\kappa, k_3, V') = (1, t_3, \emptyset) \xrightarrow{\text{ddep}} (1, t_2, \{t_3\}) = \text{versionOf}(\kappa, k_2, V')$, but $V'(k_2) = 1 < 2$. That is, the version of k_2 observed by V' directly depends on a version of k_2 that is not observed by V'.

As a second attempt, one could then try to update the the view of location k_2 to value 1, resulting in the value $V'' = V_0[k_2 \mapsto 2]$, but similarly we would find that the version $\kappa(k_2, 2) = (\_, t_2, \_)$ directly depends from the version $\kappa(k_1, 2) = (\_, \_, \{t_2\})$, and $V''(k_1) = 1 < 2$.

Finally, we can update the view V_0 to include the update of k_1: this results in the view $V_1 = V_0[\mathrm{k}_1 \mapsto 2]$. Because $\kappa(\mathrm{k}_1, 2) = (\_, t_1, \_)$, and there is no version v in κ on which $\kappa(\mathrm{k}_1, 1)$ directly depends - i.e. such that $v = (\_, \_, \_ \cup \{t_1\})$ - we can conclude that V_1 is causally consistent. The view V_1 is depicted in Figure 11(b).

We can now update the view V_1 to include the version $\kappa(\mathrm{k}_2, 2)$, resulting in the view $V_2 := V_1[\mathrm{k}_2 \mapsto 2]$, depicted in Figure 11(c). Because the view V_2 is causally consistent, and because all the versions on which $\mathrm{versionOf}(\kappa, \mathrm{k}_2, V_2)$ directly depends are observed by V_1, then V_2 is also causally consistent. Similarly, we can define the view $V_3 := V_2[\mathrm{k}_2 \mapsto 2]$, depicted in Figure 11(d), and prove that it casually consistent. Because V_3 points to $\kappa(\mathrm{k}_3, 2)$, there is nothing left to do.

AC: Note: It appears to be that causal consistency is broken (and so is PSI). The problem is that causalities between program dependencies are not tracked down, nor they can be tracked down in the MKVS model as it is now. For example, if we execute the program

$$\begin{bmatrix}[[\mathrm{k}_1] := 1;]\\ [[\mathrm{k}_2] := 1;]\end{bmatrix}; \Big\| \begin{bmatrix}\mathtt{x} := [\mathrm{k}_1];\\ \mathtt{y} := [\mathrm{k}_2];\\ \mathtt{if}\,(\mathtt{x} = 0 \wedge \mathtt{y} = 1)\{\\ \quad \mathtt{ret} := \text{☹}\ :\end{bmatrix}$$

under causal consistency, we can still obtain an execution where the value ☹ is assigned to `ret`. The reason why this happened is a violation of the *monotonic writes* session guarantee (assuming clients are wrapped within sessions.)
I finally managed to get hold to the session guaranteee paper, and it looks that the MKVS semantics only ensures monotonic reads and read my writes. Monotonic writes and write follows reads are not ensured by default. It also appear that we cannot model these last two session guarantees without recording the program order in clients. Note that if we change the model so that MKVSs keep track of the program order of clients, then the isomorphism with dependency graphs is back, and we should point out that any dependency graph specification can be converted into an execution test easily.

Causal consistency ensures that transactions can only observe a causally consistent state of the database, i.e. they can only run only by threads with a causally consistent view of the history heap. We model causal consistency by introducing the execution test $\mathsf{ET}_{\mathsf{CC}}$, defined below.

Definition 5.5. $\mathsf{ET}_{\mathsf{CC}} \vdash (\kappa, V) \triangleright O : V'$ if and only if $\mathsf{ET}_{\mathsf{RA}} \vdash (\kappa, V) \triangleright O : V'$, and $V \in \mathsf{CCViews}(\kappa)$.

AC: Update to note to self: we do need to track down the program order of clients, and introduce strong sessions (maybe just monotonic writes could be enough). See previous note.
Note to self: here there is something subtle going on. We also need to ensure that dependencies caused by the information flow of the program are tracked down. For example, we could have a transaction returning the value of a location $[k_x]$, and then another transaction copy such a value into another location $[k_y]$. There is a notion of dependency between $[k_x], [k_y]$ that is not captured by the notion of *directly depends*. On the other hand, the fact that stacks are thread local, and we have per-thread view monotonicity, should ensure that also program dependencies are preserved. A definitive proof that CC is equivalent to caual consistency specified either in terms of abstract executions or dependency graphs, would settle the argument.

Note that, the view V of cl_3 in Figure 11(e) is not causally consistent. This is because $V(\mathrm{k}_2) = 1$, and $V(\mathrm{k}_1) = 0$. However, $\kappa(\mathrm{k}_2, 2)$ directly depends in $\kappa(\mathrm{k}_1, 2)$, which is not included in the view. In general, the only case in which an execution of P_2 causes transaction $\lceil \mathsf{T}_3 \rceil$ to return value ☹ is when such a transaction is executed using a snapshot determined by a non-causally consistent view. There exists no execution of P_2 using the execution test $\mathsf{ET}_{\mathsf{CC}}$ that causes $\lceil \mathsf{T}_3 \rceil$ to return value ☹.

5.3 Update Atomic

AC: This Consistency Model shows why the notion of consistent views must depend on the set of operations that need to be executed.

The next consistency model that we consider is *Update Atomic*. Although this model has not been implemented, it has been proposed in [Cerone et al. 2015] as a strengthening to Read Atomic to avoid conflicts. The informal specification of Update Atomic states that **(i)** transactions enjoy atomic visibility, and **(ii)** transactions writing to one same location cannot execute concurrently. Update Atomic is also needed to specify more sophisticated consistency models, such as *Parallel Snapshot Isolation* and *Snapshot Isolation*.

AC: Check: Nobi said he was interested in implementing Update Atomic at some point, maybe he ended up doing something.

Programs executing under Update Atomic do not exhibit the *lost update* anomaly: two or more transactions update one location (for example, by incrementing its value by 1), but only the effects of one of them (for example, only one of the increments) will be observed by future transactions.

To illustrate how the lost update anomaly can arise in practice, consider the following program:

$$\mathsf{P}_3 := \begin{bmatrix} [\mathrm{f}_1] := 1; \\ \mathrm{a} := [\mathrm{k}]; \\ [\mathrm{k}] := a + 1; \end{bmatrix} \Bigg\| \begin{bmatrix} [\mathrm{f}_2] := 1; \\ \mathrm{a} := [\mathrm{k}]; \\ [\mathrm{k}] := a + 1; \end{bmatrix} \Bigg\| \begin{bmatrix} \mathrm{a} := [\mathrm{f}_1]; \\ \mathrm{b} := [\mathrm{f}_1]; \\ \mathrm{c} := [\mathrm{k}]; \\ \mathtt{if}\ (\mathrm{a} = 1 \wedge \mathrm{b} = 1 \wedge \mathrm{c} = 1)\{ \\ \quad \mathtt{ret} := ☹\} \end{bmatrix}$$

AC: Intuitive behaviour of the litmus test: two transactions concurrently increment k. A third transaction observes that the first two transactions have been executed. However, it only observes one of the two increments taking place.

We consider an execution in which the transactions contained in the code of clients cl_1, cl_2 both execute w.r.t. the snapshot determined by the initial view. Initially, the configuration of the program coincides with the one given in Figure 12(a). After executing the transaction of cl_1, the resulting configuration is the one depicted in Figure 12(b). Next, after cl_2 executed its own transaction, we obtain the configuration in 12(c). Note that in this configuration, both transactions t_1, t_2 read the initial version for key k. Finally, thread cl_3 updates its view to include the most recent version for all the keys $\mathrm{k}, \mathrm{f}_1, \mathrm{f}_2$. When executing the transaction in its code, all the locations will be found to have value 1, and the return variable will be set to ☹.

The reason why the program P_3 exhibited the lost update anomaly is that the transaction of cl_2 executed using a snapshot obtained from a view which did not include the most up-to-date version for location k. However, the same transaction installed a new version for k. That is, it *lost the update* of the version installed by the transaction executed by cl_1. To forbid this anomaly, we require that if a transaction writes to some key k, then it must have been executed in a snapshot obtained from a view pointing to the most recent version of k.

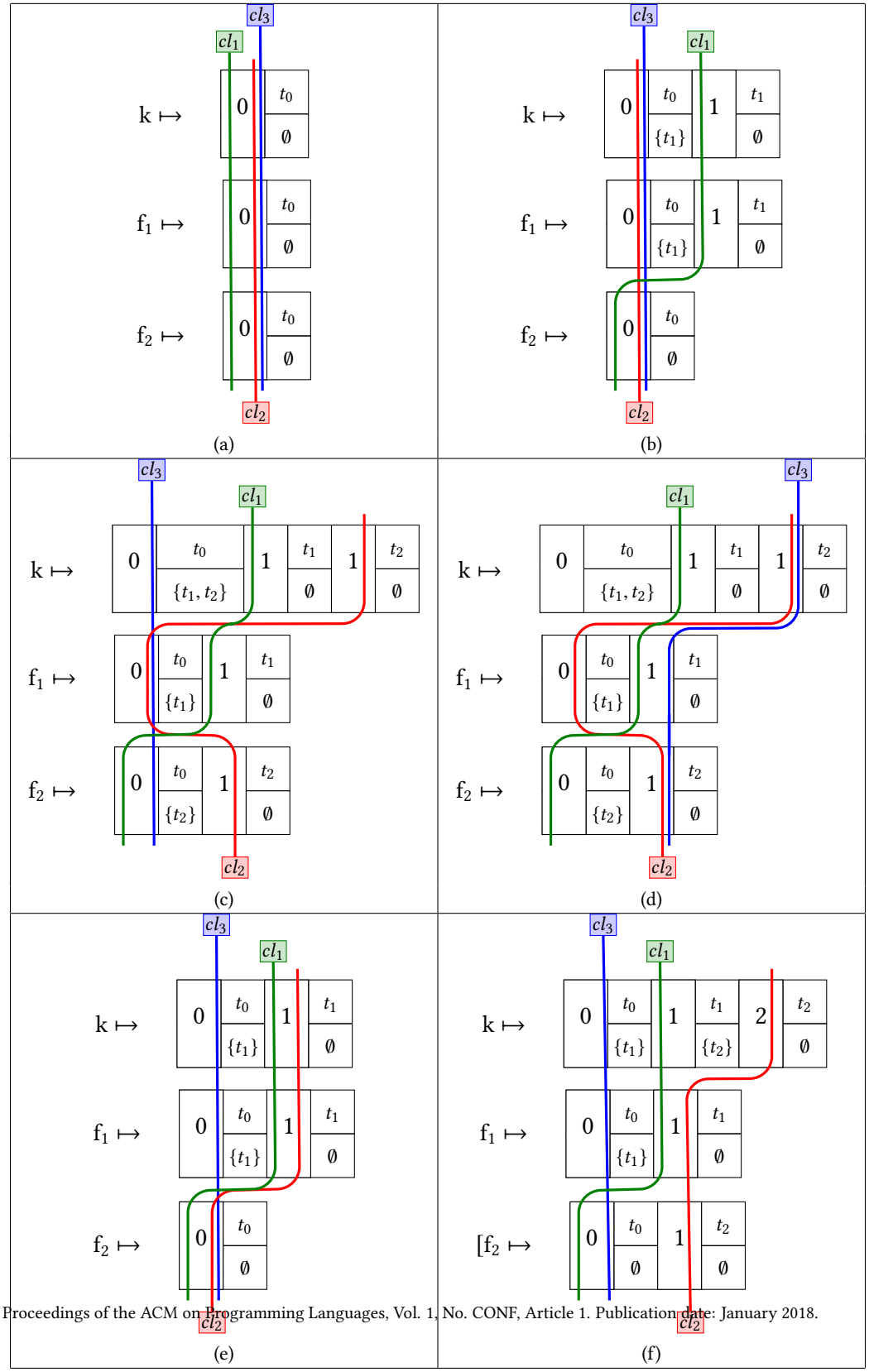


Fig. 12. History heaps obtained in a execution of P

Definition 5.6. $\mathsf{ET}_{\mathsf{UA}} \vdash (\kappa, V) \triangleright O : V'$ iff $\mathsf{ET}_{\mathsf{RA}} \vdash (\kappa, V) \triangleright O : V'$, and for all locations k such that $(\mathtt{WR}\ \mathrm{k} : \_ \in O)$ then $V(\mathrm{k}) = |\kappa(\mathrm{k})|$.

Proposition 5.7. *The execution test* $\mathsf{ET}_{\mathsf{UA}}$ *does not hinder progress. For any* κ, V, O*, there exist* $V' : V \sqsubseteq V'$ *and* $V'' : \mathsf{UpdateView}(\kappa, V', O). \sqsubseteq V''$ *such that* $\mathsf{ET}_{\mathsf{UA}} : (\kappa, V') \triangleright O, V''$.

Note that in $prog_3$, under UA, we cannot execute the transaction of cl_2 starting from the configuration depicted in Figure 12(b). This is because the view of cl_2, in such a configuration, does not include the most recent version for location k. Instead, before executing its transaction cl_2 must update its view to include the most recent version of k. The resulting configuration is depicted in Figure 12(e). After executing its transaction using the snapshot obtained from the view of cl_2 in this configuration, the resulting configuration is the one obtained in Figure 12(f). There are now three different possible configurations in which cl_3 can execute its transaction:

- the initial one, in which case the value 0 will be observed for the three keys k, f_1 and f_2. In this case the transaction will not return value ☹,
- one in which the view of cl_3 for k points to the version $(1, t_1, \{t_2\})$. Because of atomic visibility, it must also be the case that the same view for key k_2 does not include the update performed by t_2 (otherwise, the view for k should point to its most recent version, which was also installed by t_2. Thus in this case the transaction executes w.r.t a snapshot where $\mathrm{f}_2 = 0$, and the value ☹ will not be returned,
- one in which the view of cl_3 for k points to its most recent version $(2, t_2, \emptyset)$; also in this case, the value ☹ will not be returned from the transaction.

5.4 Consistent Prefix

The next consistency model that we illustrate is given by consistent prefix. In this consistency model, we must ensure that once a thread observes the effects of some transaction t, it also observes all the transactions that were executed before t. Another way to express this property is that two different transactions never observe the updates to different locations in different order.

Consider, for example, the program P_4 below.

$$\mathsf{P}_4 := \left(\begin{array}{c} [\mathrm{k}_1 := 1;] \\ \\ \left[\begin{array}{l} \mathtt{a} := [\mathrm{k}_2]; \\ \mathtt{if}\,(\mathtt{a} = 0)\{ \\ \quad \mathtt{ret} := ☹\}; \end{array}\right] \end{array} \middle\| \begin{array}{c} [[\mathrm{k}_2 := 1;] \\ \\ \left[\begin{array}{l} \mathtt{a} := [\mathrm{k}_1]; \\ \mathtt{if}\,(\mathtt{a} = 0)\{ \\ \quad \mathtt{ret} := ☹\}; \end{array}\right] \end{array} \right)$$

In this program, we let

$$[\mathsf{T}_1] = [[\mathrm{k}_1 := 1] \quad [\mathsf{T}_2] = \left[\begin{array}{l} \mathtt{a} := [\mathrm{k}_2]; \\ \mathtt{if}\,(\mathtt{a} = 0)\{ \\ \quad \mathtt{ret} := ☹\}; \end{array}\right]$$

$$[\mathsf{T}_3] = [[\mathrm{k}_2 := 1] \quad [\mathsf{T}_4] = \left[\begin{array}{l} \mathtt{a} := [\mathrm{k}_1]; \\ \mathtt{if}\,(\mathtt{a} = 0)\{ \\ \quad \mathtt{ret} := ☹\}; \end{array}\right]$$

We argue that, using either the execution tests $\mathsf{ET}_{\mathsf{RA}}$ or $\mathsf{ET}_{\mathsf{CC}}$, it is possible to obtain an execution of program P_4 where both $[\mathsf{T}_2]$ and $[\mathsf{T}_4]$ return value ☹. Such an execution can be summarised as follows:

- initially, cl_1 executes transaction $[\mathsf{T}_1]$, leading to the configuration of Figure 13(b), and the program $[\mathsf{T}_2] \parallel [\mathsf{T}_3]\,;\,[\mathsf{T}_4]$ to be executed,

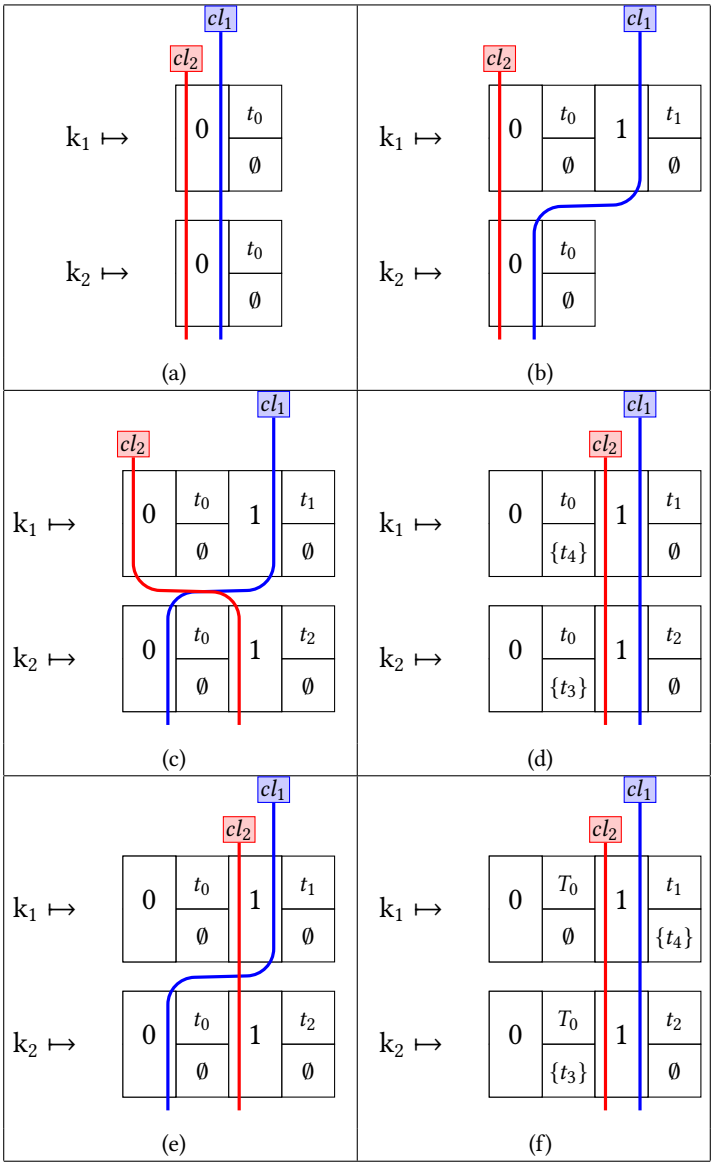


Fig. 13. Configurations obtained throughout an execution of P_4.

- then cl_2 executes transaction $[\![T_3]\!]$, leading to the configuration of Figure 13(c); the remaining program to be executed is $[\![T_3]\!] \parallel [\![T_4]\!]$,
- without changing its view, cl_1 executes transaction $[\![T_2]\!]$. The code T_2 is executed using $[k_1 \mapsto 1, k_2 \mapsto 0]$ as a snapshot; this means that, by executing T_2, the variable `ret` of the thread-local stack of cl_1 is set to ☹. Next, the thread cl_2 executes $[\![T_4]\!]$ without altering its view. Similarly to the execution of $[\![T_2]\!]$ in cl_1, this causes the `ret` variable of the thread-local

stack of cl_2 to be set to ☹. At this point, the final configuration of the program is the one given in Figure 13(d).

To avoid different threads to observe different updates in different orders, we impose a constraint on the consistency model that is known as *consistent prefix*: in a centralised database, where transactions have a start and a commit point, this constraint can be formalised as follows: *If a transaction t_1 observes the effects of another transaction t_2, then it must observe the effects of any transaction that committed before t_2*. In the MKVS framework, transactions are executed in a single step; however, one may think of the order in which transactions execute in our operational semantics to be consistent with the order in which transactions commit (this correspondence will be made precise later in the document, when we will relate executions in our operational semantics to abstract executions used in the declarative style for specifications of consistency models). By requiring that, after a client cl executes a transaction t, it pushes its own view to be up-to-date with the state of the system, we model the fact that any future transaction executed by cl will observe the effects of anything that committed before t (included).

Definition 5.8. We say that $\mathsf{CP} \vdash (hh, V) \triangleright O : V'$ if and only if $\mathsf{RA} \vdash (\kappa, V) \triangleright O : V'$, and for any location k, $V'(\mathrm{k}) = |\mathsf{UpdateMKVS}(\kappa, V, O)(k)|$.

Consider again the program P_4, this time to be executed under CP. We argue that in this case it is not possible to have both threads cl_1 and cl_2 to set the value of `ret` to ☹. The initial configuration in which the program P_4 is executed is the one depicted in Figure 13(a). Initially, either client cl_1 executes the code $\lceil \mathsf{T}_1 \rceil$, or client cl_2 executes transaction $\lceil \mathsf{T}_3 \rceil$; without loss of generality, we consider the former option (the case in which cl_2 executes first is symmetric). Upon executing the code $\lceil \mathsf{T}_1 \rceil$, we obtain the configuration of Figure 13(b), with the program $\lceil \mathsf{T}_2 \rceil \,\|(\lceil \mathsf{T}_3 \rceil ; \lceil \mathsf{T}_4 \rceil)$ to be executed next. At this point, note that if we use the execution test $\mathsf{ET}_{\mathsf{CP}}$ it is not possible for cl_2 to execute $\lceil \mathsf{T}_3 \rceil$ and obtain the configuration of Figure 13(c) as a result. This is because, $\mathsf{ET}_{\mathsf{CP}}$ require the view of cl_2 **after** executing $\lceil \mathsf{T}_3 \rceil$ to be up-to-date, i.e. to point to the last version of each key. That is, assuming that cl_2 executes $\lceil \mathsf{T}_3 \rceil$ next, we obtain the configuration of Figure 13(e). From this point on, whenever cl_2 will execute transaction $\lceil \mathsf{T}_4 \rceil$, it will read value 1 for location k, hence it won't be able to set the value of `ret` to ☹. One possible final configuration for the program is given in Figure 13(f).

5.5 Parallel Snapshot Isolation and Snapshot Isolation

Snapshot Isolation (SI) is a consistency model that has been widely employed in both centralised and distributed databases. Because snapshot isolation does not scale well to geo-replicated and distributed systems [Saeida Ardekani et al. 2013], a weakening of this model called *Parallel Snapshot Isolation* (PSI) [Sovran et al. 2011] has been recently proposed.

Both SI and PSI can be specified in the history heap framework by combining the specification of other consistency models that we have already introduced. In short, SI combines atomic visibility, the snapshot monotonicity property from consistent prefix (if a transaction t_1 observes the effects of another transaction t_2, then it also observes the effects of any transaction that committed before), and the write-conflict detection from update atomic (two committing transactions do not write concurrently to the same location). In contrast, PSI only requires atomic visibility, causal consistency and write-conflict detection. Formally, we have the following:

Definition 5.9. $\mathsf{ET}_{\mathsf{PSI}} = \mathsf{ET}_{\mathsf{CC}} \cap \mathsf{ET}_{\mathsf{UA}}$.
$\mathsf{ET}_{\mathsf{SI}} = \mathsf{ET}_{\mathsf{CP}} \cap \mathsf{ET}_{\mathsf{UA}}$.

5.6 Serialisability

Serialisability is the last and strongest consistency model that we consider. Informally, under serialisability transactions appear to be executed in a sequential order. Consider for example the program

$$\begin{bmatrix} \mathtt{a} := [\mathrm{k}_2]; \\ \mathtt{if}\,(\mathtt{a} = 0)\{; \\ \quad [\mathrm{k}_1] := 1; \\ \quad \mathtt{ret} := \text{☹}; \} \end{bmatrix} \Vert \begin{bmatrix} \mathtt{a} := [\mathrm{k}_1]; \\ \mathtt{if}\,(\mathtt{a} = 0)\{; \\ \quad [\mathrm{k}_2] := 1; \\ \quad \mathtt{ret} := \text{☹}; \} \end{bmatrix}$$

If we execute the program P_5 using any of the execution tests presented so far, we find out that it is possible to infer an execution in which both clients cl_1 and cl_2 set their local variable `ret` to value ☹. For example, under SI, this can happen as follows:

- first, client cl_1 executes its transactions in the configuration of Figure 14(a). The snapshot under which the transaction is executed is given by $[\mathrm{k}_1] \mapsto 0, \mathrm{k}_2 \mapsto 0]$, hence the execution of the transaction results in the thread-local variable `ret` of cl_1 to be set to ☹, and in the configuration of Figure 14(b).
- Next, thread cl_2 executes its transaction in the initial view $[\mathrm{k}_1 \mapsto 0, \mathrm{k}_2 \mapsto 0]$. The snapshot under which the transaction is executed is again $[\mathrm{k}_1 \mapsto 0, \mathrm{k}_2 \mapsto 0]$. After the transaction has been executed, the local variable `ret` of cl_2 is set to ☹, and the final configuration is the one of Figure 14(c). Note that, because we are assuming snapshot isolation as our consistency model, in this configuration we require that the view of thread cl_2 points to the last version of each location.

To avoid the scenario above, known as the *write skew* anomaly, it suffices to ensure that, prior to executing a transaction, the view of a thread is always up-to-date for each location.

Definition 5.10. $\mathsf{ET}_{\mathsf{SER}} \vdash (\kappa, V) \triangleright \mathcal{O} : V'$ iff, for any location k, $V(\mathrm{k}) = |\kappa(\mathrm{k})|$.

Consider again the program P_5, this time to be executed using the execution test $\mathsf{ET}_{\mathsf{SER}}$. Similar as for the execution test $\mathsf{ET}_{\mathsf{SI}}$, after client cl_1 has executed its transaction, we end up with the variable `ret` of such a thread to be set to value ☹, and with the configuration of Figure 14(b). However, at this point we can not execute the transaction of client cl_2 without updating his view beforehand. This is because such a view does not point to the most up-to-date version for key k_1. Instead, before executing its transaction cl_2, updates its view to point to the most up-to-date version of each location, Figure 14(d). Thus, client cl_2 will execute its transaction using the snapshot $[\mathrm{k}_1 \mapsto 1, [\mathrm{k}_2] \mapsto 0]$. A consequence of this fact is that the thread-local variable `ret` of cl_2 will not be set to ☹, and no new version for location k_2 will be created by the execution of the transaction. The final configuration is given in Figure 14(e).

AC: Contents: Read Atomic, Causal Consistency, Update Atomic, Consistent Prefix, Parallel Snapshot Isolation, Snapshot Isolation, Serializability. Well-formedness constraint to be placed on consistency models: progress must always be possible - i.e. it is always possible to execute a transaction if the view of all threads are up-to-date.

6 RELATIONSHIP TO ABSTRACT EXECUTIONS

In this Section we propose an alternative semantics of programs. In this semantics, states correspond to *abstract executions* [Cerone et al. 2015], overloaded with information regarding the transactions that are made visible to individual threads. The abstract execution semantics provides a bridge between specification of consistency models given in terms of history heaps, and the declarative

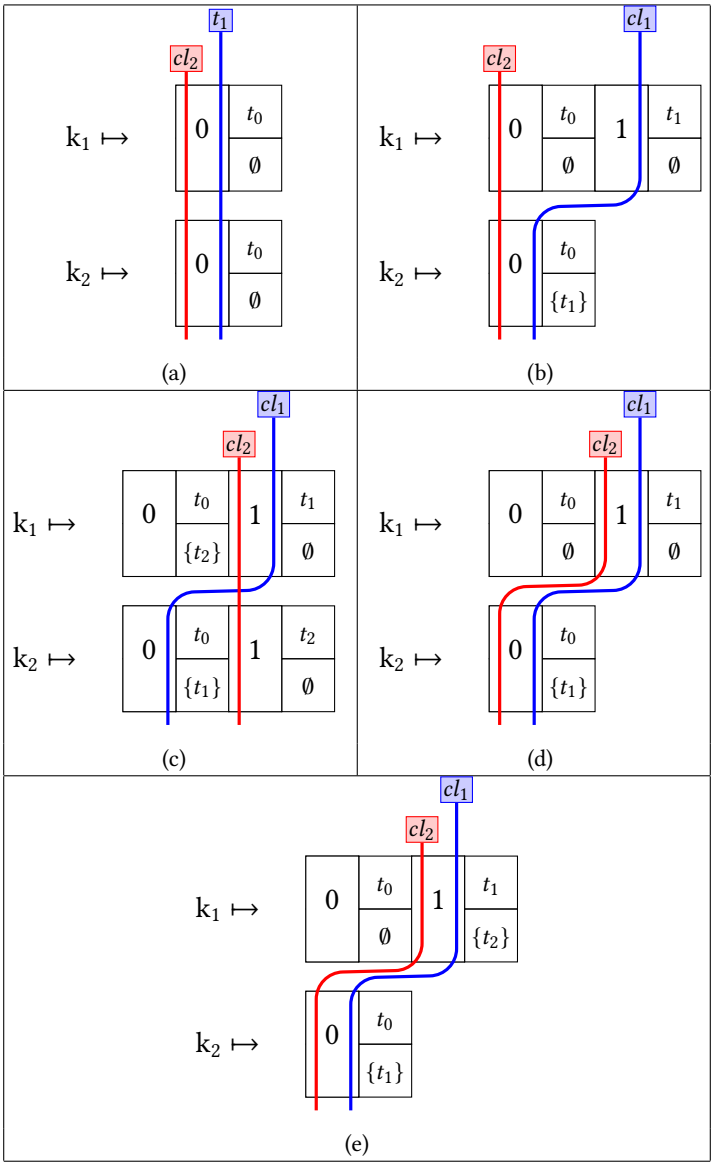


Fig. 14. Graphical representation of configurations obtained by executing P_5.

specification of consistency models, which have been already proved to be equivalent to the operational definitions of such consistency models given in the literature. This section makes the following contributions:

- a definition of (overloaded) abstract executions, and an encoding into history heaps and views,
- an operational semantics based on (overloaded) abstract executions, which is parametric in the specification of a consistency model given using the declarative style of [Cerone et al. 2015; Cerone and Gotsman 2016; Cerone et al. 2017],

- a proof that the semantics is *thorough*: for any program P and consistency model specification x − CM, the abstract execution semantics captures all the potential behaviours that P can exhibit under x − CM. Thoroughness is necessary to validate the soundness of any program analysis techniques, such as program logics, with respect to arbitrary consistency model specifications,

AC: Before I ws referring to what I call thoroughness as completeness. I am now changing this, because it looks like it only created confusion. In short words, this result requires defining the anarchic model of transactions - one in which the database actually reads and writes non-deterministic values despite the client requests, and prove that any abstract execution which is valid w.r.t. x − CM and can be obtained in the anarchic semantics, can also be obtained in the x − CM semantics. More details on this later.

- a proof that the specifications of consistency models in terms of history heaps are sound and complete with respect to the specifications of history heaps given in terms of abstract execution:

  Soundness - for any program P, any execution of P under the history heap semantics and consistency model specification CM can be simulated by an execution of P under the abstract execution semantics and the declarative specification of CM,

  Completeness - for any program P, any execution of P under the abstract execution semantics and the declarative specification of CM can be simulated by an execution of P under the history heap semantics and consistency model specification CM.

6.1 Overloaded Abstract Executions

AC: Note to self - Stop Here! I reaaally need to put everything into a consistent state before proceeding. Inconsistency of the document was fine up to now, but now that I am going into theorems and proofs I cannot be inconsistent anymore. Go back to square one, fix everything, then repeat another time.

Definition 6.1. An overloaded abstract execution (OAE) is a tuple $\hat{\mathcal{X}} = (\mathscr{T}, \mathsf{I}, \mathsf{SO}, \mathsf{VIS}, \mathsf{AR})$ where:

(1) $\mathscr{T} : \textsc{TransID} \rightharpoonup 2^{\mathsf{Op}}$ is a set of transactions, each of which is equipped with a set of operations.
(2) $\mathsf{I} \subseteq \textsc{Clients}$ is a set of thread identifiers.
(3) $\mathsf{SO} \subseteq (\mathrm{dom}(\mathscr{T}) \times \mathrm{dom}(\mathscr{T}) \cup \mathsf{I}$ is the program order, and it corresponds to the union of total orders defined over a partition of $\mathrm{dom}(\mathscr{T}) \cup \mathsf{I}$ that also partitions I into singletons. That is, for any $cl \in \mathsf{I}$ there exists a subset $\mathcal{T}_{cl}$ of $\mathrm{dom}(\mathscr{T})$ such that for any $cl, cl' \in \mathsf{I}$, $\mathcal{T}_{cl} \cap \mathcal{T}_{cl'} \neq \emptyset \implies cl = cl'$; $\bigcup_{cl \in \mathsf{I}} \mathcal{T}_{cl} = \mathrm{dom}(\mathscr{T})$; for each $cl \in \mathsf{I}$ there exists a strict, total order $\mathsf{SO}_i \subseteq \mathcal{T}_{cl} \times (\mathcal{T}_{cl} \cup \{cl\})$, and $\mathsf{SO} = (\bigcup_{cl \in \mathsf{I}} \mathsf{SO}_{cl})$.
(4) $\mathsf{VIS} \subseteq \mathrm{dom}(\mathscr{T}) \times \mathrm{dom}(\mathscr{T})$ is a partial order,
(5) $\mathsf{AR} \subseteq \mathrm{dom}(\mathscr{T}) \times \mathrm{dom}(\mathscr{T})$ is a total order such that $\mathsf{VIS} \subseteq \mathsf{AR}$.

6.2 Completeness of the Semantics

AC: Contents: Abstract execution Semantics and the anarchic model. Encoding of abstract executions into history heaps. Also, traces in the history heaps semantics can be used to recover an abstract execution $\mathcal{X}$. Hence it is possible to convert history heaps specifications into sets of abstract executions. I think this should be the Theorem: for every possible trace of a program P that is allowed by the anarchic model, and that results into an abstract execution χ that is allowed by CM, there exists a trace of the same program under the history heaps CM-semantics, and whose encoding into an abstract execution is exactly $\mathcal{X}$.

6.3 Remarks on the operational semantics

AC: There was a discussion on whether views of threads should always be consistent w.r.t a consistency model specification, or whether they should be consistent only w.r.t. to a consistency model specification only prior to executing a transaction.
In this section I should argue that the first option leads to losing the completeness of the semantics.

REFERENCES

Peter Bailis, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Scalable atomic visibility with RAMP Transactions. In *2014 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 27–38.

Sebastian Burckhardt, Manuel Fahndrich, Daan Leijen, and Mooly Sagiv. 2012. Eventually Consistent Transactions. In *22nd European Symposium on Programming (ESOP)*. 67âĂŞ86. https://www.microsoft.com/en-us/research/publication/eventually-consistent-transactions/

Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. 2015. A framework for transactional consistency models with atomic visibility. In *26th International Conference on Concurrency Theory (CONCUR)*. Dagstuhl, 58–71.

Andrea Cerone and Alexey Gotsman. 2016. Analysing Snapshot Isolation. In *2016 ACM Symposium on Principles of Distributed Computing (PODC)*. 55–64.

Andrea Cerone, Alexey Gotsman, and Hongesok Yang. 2017. Algebraic Laws for Weak Consistency. In *27th International Conference on Concurrency Theory (CONCUR)*. 26:1–22:16.

Natacha Crooks, Youer Pu, Lorenzo Alvisi, and Allen Clement. 2017. Seeing is Believing: A Client-Centric Specification of Database Isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC '17)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/3087801.3087802

Gowtham Kaki, Kartik Nagar, Mahsa Najafzadeh, and Suresh Jagannathan. 2017. Alone Together: Compositional Reasoning and Inference for Weak Isolation. *Proc. ACM Program. Lang.* 2, POPL, Article 27 (Dec. 2017), 34 pages. https://doi.org/10.1145/3158115

Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *23rd ACM Symposium on Operating Systems Principles (SOSP)*. 401–416.

M. Saeida Ardekani, P. Sutra, and M. Shapiro. 2013. Non-monotonic Snapshot Isolation: Scalable and Strong Consistency for Geo-replicated Transactional Systems. In *32nd International Symposium on Reliable Distributed Systems (SRDS)*. 163–172.

Marc Shapiro, Masoud Saeida Ardekani, and Gustavo Petri. 2016. Consistency in 3D. In *Concur (Lipics)*, Josée Desharnais and Radha Jagadeesan (Eds.), Vol. 59. Québec, Québec, Canada, 3:1–3:14.

Y. Sovran, R. Power, M. K. Aguilera, and J. Li. 2011. Transactional storage for geo-replicated systems. In *23rd ACM Symposium on Operating Systems Principles (SOSP)*. 385–400.