A

Analysing Snapshot Isolation

Andrea Cerone, Imperial College London, UK
Alexey Gotsman, IMDEA Software Institute, Spain

Snapshot isolation (SI) is a widely used consistency model for transaction processing, implemented by most major databases and some of transactional memory systems. Unfortunately, its classical definition is given in a low-level operational way, by an idealised concurrency-control algorithm, and this complicates reasoning about the behaviour of applications running under SI. We give an alternative specification to SI that characterises it in terms of transactional dependency graphs of Adya et al., generalising serialization graphs. Unlike previous work, our characterisation does not require adding additional information to dependency graphs about start and commit points of transactions. We then exploit our specification to obtain two kinds of static analyses. The first one checks when a set of transactions running under SI can be chopped into smaller pieces without introducing new behaviours, to improve performance. The other analysis checks whether a set of transactions running under a weakening of SI behaves the same as when running under SI.

1. INTRODUCTION

Transactions simplify concurrent programming by enabling computations on shared data that are isolated from other concurrent computations and resilient to failures. They are commonly provided by databases [Bernstein et al., 1987] and, more recently, by transactional memory systems [Herlihy and Moss, 1993]. Ideally, programmers would like to get strong guarantees about the isolation of transactional computations, formalised by the notion of *serializability* [Bernstein et al., 1987]: the results of concurrently executing a set transactions could be obtained if these transactions executed atomically in some order. Unfortunately, ensuring serializability carries a significant performance penalty. For this reason, transactional systems often provide weaker guarantees about transaction processing, formalised by *weak consistency models*. Snapshot isolation (SI) [Berenson et al., 1995] is one of the most popular such models, implemented by major centralised databases (e.g., MS SQL Sever and Oracle), distributed databases [Daudjee and Salem, 2006; Serrano et al., 2007; Peng and Dabek, 2010] and transactional memory systems [Litz et al., 2014; Bieniusa and Fuhrmann, 2010; Dias et al., 2011; Closure, 2016; Riegel et al., 2006].

Informally, SI is defined by a multi-version concurrency control algorithm as follows. A transaction T reads values of shared objects from a snapshot taken at its start. The transaction commits only if it passes a *write-conflict* detection check: since T started, no other committed transaction has written to any object that T also wrote to. If the check fails, T aborts. Once T commits, its changes become visible to all transactions that take a snapshot afterwards. This concurrency-control algorithm allows unserializable behaviours, called *anomalies*. One of them, *write skew*, is graphically illustrated in Figure 2(f). Each of the transactions T_1 and T_2 checks that the combined balance of two accounts exceeds 100 and, if so, withdraws 100 from one of them. Under SI, both transactions may pass the checks and make the withdrawals from different accounts, resulting in the combined balance going negative. This outcome cannot occur under serializability. Given such anomalies, reasoning

This article is a revised and expanded version of a paper that received the Best Paper Award at the 35th Annual ACM Symposium on Principles of Distributed Computing (PODC).

DOI: http://dx.doi.org/10.1145/0000000.0000000

about the behaviour of applications executing under SI is far from trivial. This task is further complicated by the fact that the specification of SI is given in a low-level operational way, by a concurrency control algorithm. To facilitate reasoning about applications using SI and establishing useful results about this consistency model, we need a more declarative specification that abstracts from implementation-level details as much as possible.

An approach that yields such consistency model specifications was proposed by Adya et al. [Adya, 1999; Adya et al., 2000]. In this approach, an execution of a set of transactions is described by three kinds of *dependencies* between pairs of transactions T_1 and T_2: *read dependencies* record when T_1 reads the value of an object written by T_2; *write dependencies* record when T_1 overwrites the value of an object written by T_2; finally, *anti-dependencies* are derived from read and write dependencies in a certain way (§3). A set of transactions and dependencies between them form a *dependency graph*, generalising classical serialization graphs [Bernstein et al., 1987]. Then the set of executions allowed by a given consistency model is defined by those dependency graphs that lack certain cycles; in particular, serializable executions are characterised by acyclic dependency graphs. This way of specifying consistency models has been shown to be particularly appropriate for designing static analyses [Fekete et al., 2005; Jorwekar et al., 2007; Shasha et al., 1995; Zhang et al., 2013; Cerone et al., 2015b], run-time monitoring [Cahill et al., 2009; Zellag and Kemme, 2014] and proving concurrency-control algorithms correct [Lin et al., 2009; Xie et al., 2015; Diegues and Romano, 2014]. In particular, specifications in terms of dependency graphs facilitate exploring possible program executions in a static analysis, because the analysis can determine which dependencies can possibly exist at run time by looking for pairs of read or write accesses to the same object in the code of different transactions. In contrast, it is hard to predict statically more low-level information about transaction execution, such as the order in which transactions commit.

Specifications in terms of dependency graphs have been proposed for ANSI isolation levels such as serializability, Read Committed and Repeatable Read [Adya, 1999], as well as more recent proposals of consistency models [Bailis et al., 2014; Xie et al., 2015]. But surprisingly, there is no such specification of SI. This is not for the want of trying: Adya did propose a definition of SI that refers to dependency graphs [Adya, 1999]. However, to capture the subtle semantics of SI, this definition extends the graphs by a relation describing low-level information about transaction execution, which negates their benefits.

In this paper we propose the first characterisation of SI solely in terms of dependency graphs (§4) and apply it to develop new static analyses (§5 and §6). Namely, we show that SI allows exactly the executions represented by dependency graphs that contain only cycles with at least two adjacent anti-dependency edges. The proof of this fact is highly non-trivial and represents a key technical contribution of this paper. It requires showing that, given a dependency graph satisfying the above acyclicity condition, we can construct certain relations describing how the transactions can be processed by the SI concurrency control, e.g., the order in which transactions commit. Constructing these relations from transactional dependencies is challenging, and the main insight of our proof is given by a procedure for this construction, based on solving certain kinds of inequalities over relations.

To illustrate the benefits of our dependency graph characterisation of SI, we exploit it to develop two kinds of static analyses. First, we propose a new static analysis for the classical problem of *transaction chopping* [Shasha et al., 1995; Xiang and Scott, 2015; Afek et al., 2011]—checking when transactions in an application can be chopped into smaller pieces without introducing new behaviours (§5). When applied to long-running transactions executing under SI, chopping can improve performance, because the longer an SI transaction runs, the higher the chances are that it will abort due to a write conflict. There are analyses for transaction chopping under serializability [Shasha et al., 1995] and parallel SI [Cerone et al., 2015b], a recently proposed weaker version of SI for large-scale databases [Sovran

et al., 2011; Saeida Ardekani et al., 2013a]. However, there has been no such analysis under SI, despite the widespread use of this consistency model.

Our dependency graph characterisation of SI is instrumental in deriving the static analysis for transaction chopping, and not only due to the feasibility of determining possible dependencies statically. In more detail, chopping transforms transactions in a program into *sessions* [Terry et al., 1994; Daudjee and Salem, 2006] (aka *chains* [Zhang et al., 2013]) of smaller transactions, which ensure that the transactions will be executed in the order given, but provide no isolation guarantees. A chopping is correct if each SI execution of the resulting program can be *spliced* into an SI execution that has the same operations as the original one, but where all operations from each session are executed inside a single transaction. Showing the existence of the spliced execution is challenging on SI because it is non-trivial to pick the order in which its transactions should commit. Our characterisation of SI in terms of transactional dependencies avoids this complication, because unlike low-level aspects of an execution, these dependencies do not change significantly during splicing, and this makes it easy to construct the spliced execution.

The other kind of static analyses that we consider checks whether an application is *robust* [Fekete et al., 2005; Shasha and Snir, 1988] against weakening consistency: it behaves the same regardless of whether it uses a database providing a weak consistency model or a database proving a stronger model (§6). When this is the case, the application programmer can reap the performance benefits of using the weaker model, yet can reason about the correctness of the application assuming the stronger one. We first show that our SI characterisation allows easily deriving a variant of an existing analysis that checks whether an application executing under SI behaves the same as when executing under serializability [Fekete et al., 2005] (robustness *against SI*, §6.1). We then propose a new static analysis that checks whether an application executing under the recently proposed parallel SI [Sovran et al., 2011; Saeida Ardekani et al., 2013a] behaves the same as when executing under the stronger classical SI (robustness *against parallel SI towards SI*, §6.2). To derive this static analysis, we formulate a dependency graph characterisation of parallel SI, which can be given more easily than for classical SI. Again, our characterisations of consistency models in terms of dependency graphs greatly facilitate deriving the above robustness analyses, since the characterisations allow us to easily map between executions on different models.

2. SNAPSHOT ISOLATION

We start by formally defining snapshot isolation (SI), as well as serializability. Rather than using the classical definition of SI by a concurrency-control algorithm (§1), it is technically convenient for us to build on a more declarative specification that we previously proposed and proved equivalent to the standard one [Cerone et al., 2015a]. Even though this specification is stated in terms of lower-level relations than transactional dependencies, it avoids referring explicitly to times at which a transaction takes a snapshot in the SI concurrency-control algorithm. We first introduce mathematical structures that represent transaction execution in the specification.

We consider a transactional system managing a set of integer-valued ***objects*** $\mathsf{Obj} = \{x, y, \ldots\}$. Transactions read and write the objects, and in our representation of executions, we denote each invocation of such an operation by an ***event*** from a set $\mathsf{Event} = \{e, f, \ldots\}$. A function $\mathsf{op} : \mathsf{Event} \to \mathsf{Op}$ for

$$\mathsf{Op} = \{\mathtt{read}(x, n), \mathtt{write}(x, n) \mid x \in \mathsf{Obj}, n \in \mathbb{Z}\}$$

determines the operation a given event denotes: reading a value n from an object x or writing n to x. We call a binary relation a ***strict partial order*** if it is transitive and irreflexive. We call it a ***total order*** if it additionally relates any pair of distinct elements one way or another. We represent an execution of a single transaction by the following structure, recording a set of operations and the order in which they were invoked.

Definition 2.1. A ***transaction*** $T, S, \ldots$ is a pair (E, po), where $E \subseteq \mathsf{Event}$ is a finite, non-empty set of events and the ***program order*** $\mathsf{po} \subseteq E \times E$ is a total order.

For simplicity, all transactions in this paper are assumed to be committed: our specifications do not constrain values read inside aborted or ongoing transactions; this limitation could be lifted following [Adya, 1999; Doherty et al., 2013; Guerraoui and Kapalka, 2008]. We denote components of transactions and similar structures as in E_T and po_T. We write $e \xrightarrow{\mathsf{po}} f$ and $(e, f) \in \mathsf{po}$ interchangeably, and similarly for other relations.

To allow transaction chopping (§5), we assume that the transactional system allows its clients to group several transactions into a session [Terry et al., 1994], which establishes an ordering on the transactions. Thus, instead of classical SI and serializability, we actually define their ***strong session*** variants [Daudjee and Salem, 2006; Daudjee and Salem, 2004]. We represent the client-visible results of executing a set of sessions by a *history*.

Definition 2.2. A ***history*** is a pair $\mathcal{H} = (\mathcal{T}, \mathsf{SO})$, where $\mathcal{T}$ is a finite set of transactions with disjoint sets of events and the ***session order*** $\mathsf{SO} \subseteq \mathcal{T} \times \mathcal{T}$ is a union of total orders defined on disjoint subsets of $\mathcal{T}$, which correspond to transactions in different sessions.

For simplicity, we elide the treatment of infinite computations, and thus histories are always finite. A consistency model, such as SI or serializability, is specified by a set of histories. To define this set, we extend histories with two relations, declaratively describing how the transactional system processes transactions.

Definition 2.3. An ***abstract execution*** (or just an execution) is a tuple

$$\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO}),$$

where $(\mathcal{T}, \mathsf{SO})$ is a history and the ***visibility*** and ***commit orders*** $\mathsf{VIS}, \mathsf{CO} \subseteq \mathcal{T} \times \mathcal{T}$ are such that $\mathsf{VIS} \subseteq \mathsf{CO}$ and CO is total.

For $\mathcal{H} = (\mathcal{T}, \mathsf{SO})$ we shorten $(\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO})$ to $(\mathcal{H}, \mathsf{VIS}, \mathsf{CO})$. In terms of the SI concurrency-control algorithm sketched in §1, $T \xrightarrow{\mathsf{VIS}} S$ means that the writes done by the transaction T are included into the snapshot taken by the transaction S; $T \xrightarrow{\mathsf{CO}} S$ means that T commits earlier than S. The constraint $\mathsf{VIS} \subseteq \mathsf{CO}$ ensures that the snapshot taken by a transaction may only include previously committed transactions. SI or serializability allow those histories that can be extended to an abstract execution satisfying certain ***consistency axioms*** from Figure 1, which specify the corresponding guarantees about transaction processing.

Definition 2.4. The sets of executions and histories ***allowed by (strong session) SI and serializability*** are:

$$\begin{aligned}
\mathsf{ExecSI} &= \{\mathcal{X} \mid \mathcal{X} \models \textsc{Int} \wedge \textsc{Ext} \wedge \textsc{Session} \wedge \textsc{Prefix} \wedge \textsc{NoConflict}\};\\
\mathsf{ExecSER} &= \{\mathcal{X} \mid \mathcal{X} \models \textsc{Int} \wedge \textsc{Ext} \wedge \textsc{Session} \wedge \textsc{TotalVis}\};\\
\mathsf{HistSI} &= \{\mathcal{H} \mid \exists \mathsf{VIS}, \mathsf{CO}.\, (\mathcal{H}, \mathsf{VIS}, \mathsf{CO}) \in \mathsf{ExecSI}\};\\
\mathsf{HistSER} &= \{\mathcal{H} \mid \exists \mathsf{VIS}, \mathsf{CO}.\, (\mathcal{H}, \mathsf{VIS}, \mathsf{CO}) \in \mathsf{ExecSER}\}.
\end{aligned}$$

We now explain the axioms in Figure 1, as well as anomalies that SI allows or disallows; the latter are summarised in Figure 2. We use the following notation. For a set A and a total order $\mathcal{R} \subseteq A \times A$, we let $\max_{\mathcal{R}}(A)$ be the element $a \in A$ such that $\forall b \in A.\, a = b \vee (b, a) \in \mathcal{R}$; if $A = \emptyset$, then $\max_{\mathcal{R}}(A)$ is undefined. In the following, the use of $\max_{\mathcal{R}}(A)$ in an expression implicitly assumes that it is defined. We define $\min_{\mathcal{R}}(A)$ similarly. For a relation $\mathcal{R} \subseteq A \times A$ and an element $a \in A$, we let $\mathcal{R}^{-1}(a) = \{b \mid (b, a) \in \mathcal{R}\}$. We define the sequential composition of relations $\mathcal{R}_1$ and $\mathcal{R}_2$ as

$$\mathcal{R}_1 ; \mathcal{R}_2 = \{(a, b) \mid \exists c.\, (a, c) \in \mathcal{R}_1 \wedge (c, b) \in \mathcal{R}_2\}.$$

$\forall (E, \mathsf{po}) \in \mathcal{T}. \forall e \in E. \forall x, n. (\mathsf{op}(e) = \mathtt{read}(x, n) \wedge \{f \mid \mathsf{op}(f) = \_(x, \_) \wedge f \xrightarrow{\mathsf{po}} e\} \neq \emptyset \implies \mathsf{op}(\max_{\mathsf{po}}\{f \mid \mathsf{op}(f) = \_(x, \_) \wedge f \xrightarrow{\mathsf{po}} e\}) = \_(x, n)$	(INT)
$\forall T \in \mathcal{T}. \forall x, n. T \vdash \mathtt{read}(x, n) \implies \max_{\mathsf{CO}}(\mathsf{VIS}^{-1}(T) \cap \mathsf{WriteTx}_x) \vdash \mathtt{write}(x, n)$	(EXT)
$\forall T, S \in \mathcal{T}. \forall x. (T, S \in \mathsf{WriteTx}_x \wedge T \neq S) \implies (T \xrightarrow{\mathsf{VIS}} S \vee S \xrightarrow{\mathsf{VIS}} T)$	(NOCONFLICT)
$\mathsf{SO} \subseteq \mathsf{VIS}$	(SESSION)
$\mathsf{CO} \,;\, \mathsf{VIS} \subseteq \mathsf{VIS}$	(PREFIX)
$\mathsf{CO} = \mathsf{VIS}$	(TOTALVIS)

Fig. 1. Axioms constraining an abstract execution $(\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO})$.

We write $\_$ for a value that is irrelevant and implicitly existentially quantified.

The INT and EXT axioms in Figure 1 ensure that a transaction reads from a snapshot of object states and its own writes. The ***internal consistency axiom*** INT ensures that a read event e on an object x returns the same value as the last write to or a read from x preceding e in the same transaction. In particular, the axiom ensures that, if a transaction writes to an object and then reads the object, then it will observe its last write. The axiom also disallows ***unrepeatable reads***: if a transaction reads an object twice without writing to it in-between, then it will read the same value in both cases.

If a read is not preceded in the same transaction by an operation on the same object, then its value is determined in terms of writes by other transactions using the ***external consistency axiom*** EXT. For $T = (E, \mathsf{po})$, we let $T \vdash \mathtt{write}(x, n)$ if T writes to x and the last value written is n:

$$\mathsf{op}(\max_{\mathsf{po}}\{e \mid \mathsf{op}(e) = \mathtt{write}(x, \_)\}) = \mathtt{write}(x, n).$$

We let $T \vdash \mathtt{read}(x, n)$ if T reads from x before writing to it and n is the value returned by the first such read:

$$\mathsf{op}(\min_{\mathsf{po}}\{e \mid \mathsf{op}(e) = \_(x, \_)\}) = \mathtt{read}(x, n).$$

We also let $\mathsf{WriteTx}_x = \{T \mid T \vdash \mathtt{write}(x, \_)\}$. Then EXT ensures that, if a transaction T reads an object x before writing to it, then the value read is determined by the transactions that are included into T's snapshot according to VIS and that wrote to x; T reads the value written by the transaction from this set that committed last according to CO. For simplicity, we consider only executions where the above set is always non-empty; this can be ensured by introducing a special transaction that writes initial values of all objects. The executions in Figures 2(a) and 2(b) satisfy EXT. Note that the read in T_3 in Figure 2(b) returns 25, because T_1 precedes T_2 in the commit order. The axiom EXT implies the absence of ***dirty reads***: a committed transaction cannot read a value written by an aborted or ongoing transaction (as such a transaction are not present in abstract executions), and a transaction also cannot read a value that was overwritten by the transaction that wrote it (ensured by the definition of $T \vdash \mathtt{write}(x, n)$). Finally, EXT guarantees that either all or none of writes by a transaction can be visible to another transaction. For example, EXT disallows the execution in Figure 2(c) and, in fact, any execution with the same history. This illustrates

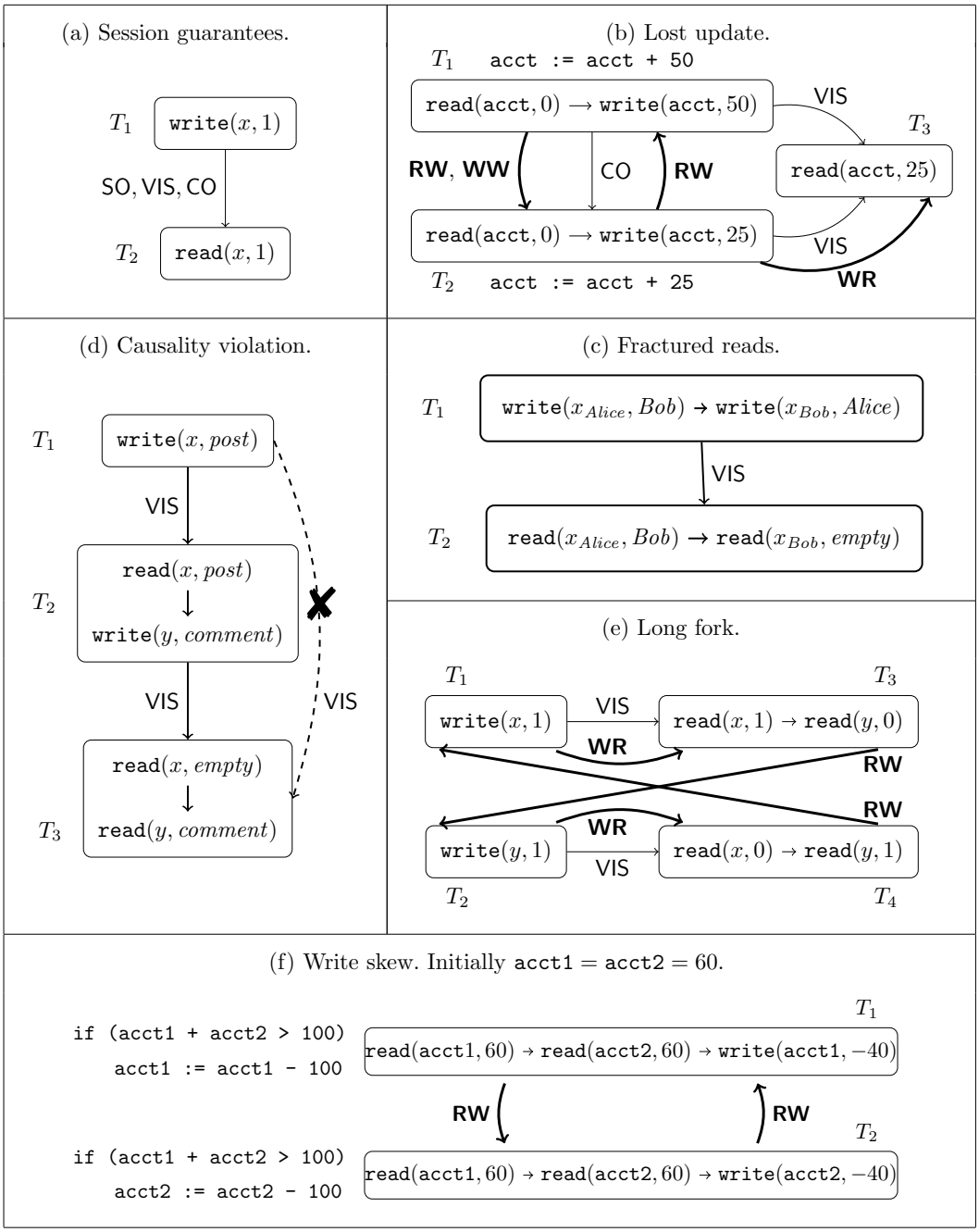


Fig. 2. Abstract executions illustrating SI and serializability. Boxes represent transactions, and arrows inside boxes represent the program order. We omit irrelevant CO edges. We also omit a special transaction that writes initial versions of all objects and precedes all the other transactions in VIS and CO. The bold edges are explained in §3.

a ***fractured reads*** anomaly: T_1 makes *Alice* and *Bob* friends, but T_2 observes only one direction of the friendship relationship.

Our specification determines the snapshot that a transaction reads from based on an arbitrary visibility relation and does not require the snapshot to be "latest"; this is similar to so-called ***generalised SI*** [Elnikety et al., 2005]. However, following strong session SI [Daudjee and Salem, 2006; Daudjee and Salem, 2004], the SESSION axiom requires the snapshot to include the effects of all preceding transactions in the same session. For example, in the execution in Figure 2(a), the session order between T_1 and T_2 induces a visibility edge according to SESSION.

The PREFIX axiom ensures that, if the snapshot taken by a transaction T includes a (committed) transaction S, then this snapshot also includes all transactions that committed before S. Note that PREFIX and the property $\mathsf{VIS} \subseteq \mathsf{CO}$ in Definition 2.4 imply that VIS is transitive. Hence, PREFIX disallows the ***causality violation*** anomaly in Figure 2(d): here T_3 sees the effects of T_2, but not the effects of T_1, which is seen by T_2. Since VIS is transitive, we must have a VIS edge between T_1 and T_2; but then by EXT, T_3 has to read *comment* from y. PREFIX also disallows the ***long fork*** anomaly shown in Figure 2(e), which is allowed by parallel SI [Sovran et al., 2011; Saeida Ardekani et al., 2013a], a weakening of SI. There transactions T_1 and T_2 concurrently write to objects x and y. Transaction T_3 sees the write by T_1, but not the write by T_2; conversely, transaction T_4 sees the write by T_2, but not the write by T_1. Thus, from the perspectives of T_3 and T_4, the writes of T_1 and T_2 happen in different orders. PREFIX disallows any execution with the history in Figure 2(e), because in such an execution T_1 and T_2 have to be related by CO one way or another; but then by PREFIX, either T_4 has to observe the write to x or T_3 has to observe the write to y.

The axioms explained so far do not prevent the ***lost update*** anomaly, illustrated by the execution in Figure 2(b). This execution could arise from the code in the figure that uses transactions T_1 and T_2 to make deposits into an account. The two transactions read the initial balance of the account and concurrently modify it, resulting in one deposit getting lost. This anomaly is disallowed by the NOCONFLICT axiom: if two distinct transactions write to the same object, then one of them has to be aware of the other. This axiom rules out any execution with the history in Figure 2(b): it forces T_1 and T_2 to be ordered by VIS, so that they cannot both read 0 from `acct`. In the SI concurrency control this is ensured by the write-conflict detection check (§1).

The set HistSI (Definition 2.4) defined using the consistency axioms explained so far is exactly the one produced by the SI concurrency-control algorithm [Cerone et al., 2015a]. The axioms allow the execution in Figure 2(f) with the characteristic SI anomaly of ***write skew*** (§1), disallowed by serializability. We formalise the latter by the axiom TOTALVIS, which requires visibility to totally order all transactions. Then the axioms INT and EXT ensure that the transactions are processed according to the usual sequential semantics. We thus have $\mathsf{HistSER} \subset \mathsf{HistSI}$.

3. DEPENDENCY GRAPHS

From an abstract execution we can extract several kinds of dependencies between its transactions, which are used in consistency model specifications in the style of Adya et al. [Adya, 1999; Adya et al., 2000].

Definition 3.1. Let $\mathcal{X} = (\mathcal{H}, \mathsf{VIS}, \mathsf{CO})$ be an execution. For $x \in \mathsf{Obj}$, we define the following relations on $\mathcal{T}_{\mathcal{H}}$:

— **read dependency**:

$$T \xrightarrow{\mathsf{WR}_{\mathcal{X}}(x)} S \iff S \vdash \mathtt{read}(x, \_) \wedge T = \max_{\mathsf{CO}}(\mathsf{VIS}^{-1}(S) \cap \mathsf{WriteTx}_x);$$

— **write dependency**:

$$T \xrightarrow{\mathsf{WW}_{\mathcal{X}}(x)} S \iff T \xrightarrow{\mathsf{CO}} S \wedge T, S \in \mathsf{WriteTx}_x;$$

— **anti-dependency**:

$$T \xrightarrow{\mathsf{RW}_{\mathcal{X}}(x)} S \iff T \neq S \wedge \exists T'.\, T' \xrightarrow{\mathsf{WR}_{\mathcal{X}}(x)} T \wedge T' \xrightarrow{\mathsf{WW}_{\mathcal{X}}(x)} S.$$

Informally, $T \xrightarrow{\mathsf{WR}_{\mathcal{X}}(x)} S$ means that S reads T's write to x (cf. the EXT axiom in Figure 1); $T \xrightarrow{\mathsf{WW}_{\mathcal{X}}(x)} S$ means that S overwrites T's write to x; $T \xrightarrow{\mathsf{RW}_{\mathcal{X}}(x)} S$ means that S overwrites the write to x read by T. For example, the dependencies of the executions in Figures 2(b), 2(e) and 2(f) are shown there with bold arrows (keep in mind that the pictures omit a special initialisation transaction). We often abuse notation and use the symbol $\mathsf{WR}_{\mathcal{X}}$ to also denote the relation $\bigcup_{x \in \mathsf{Obj}} \mathsf{WR}_{\mathcal{X}}(x) \subseteq \mathcal{T}_{\mathcal{H}} \times \mathcal{T}_{\mathcal{H}}$, and similarly for $\mathsf{WW}_{\mathcal{X}}$ and $\mathsf{RW}_{\mathcal{X}}$.

A key goal of this paper is to characterise SI solely in terms of dependencies: we want to determine whether SI allows a given history by looking for appropriate dependencies between its transactions rather than visibility and commit orders, as in Definition 2.4. To this end, we extend histories to *dependency graphs* (aka direct serialization graphs) [Adya, 1999], which include relations representing the dependencies.

Definition 3.2. A ***dependency graph*** is a tuple $\mathcal{G} = (\mathcal{T}, \mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, where $(\mathcal{T}, \mathsf{SO})$ is a history and

(1) $\mathsf{WR} : \mathsf{Obj} \to 2^{\mathcal{T} \times \mathcal{T}}$ is such that:
   (a) $\forall T, S \in \mathcal{T}.\, \forall x.\, T \xrightarrow{\mathsf{WR}(x)} S \implies \exists n.\, T \neq S \wedge T \vdash \mathtt{write}(x, n) \wedge S \vdash \mathtt{read}(x, n)$;
   (b) $\forall S \in \mathcal{T}.\, \forall x.\, S \vdash \mathtt{read}(x, \_) \implies \exists T.\, T \xrightarrow{\mathsf{WR}(x)} S$;
   (c) $\forall T, T', S \in \mathcal{T}.\, \forall x.\, (T \xrightarrow{\mathsf{WR}(x)} S \wedge T' \xrightarrow{\mathsf{WR}(x)} S) \implies T = T'$.
(2) $\mathsf{WW} : \mathsf{Obj} \to 2^{\mathcal{T} \times \mathcal{T}}$ is such that for every $x \in \mathsf{Obj}$, $\mathsf{WW}(x)$ is a total order on the set $\mathsf{WriteTx}_x$;
(3) $\mathsf{RW} : \mathsf{Obj} \to 2^{\mathcal{T} \times \mathcal{T}}$ is derived from WR and WW as in Definition 3.1:

$$\forall T, S \in \mathcal{T}.\, \forall x.\, T \xrightarrow{\mathsf{RW}(x)} S \iff T \neq S \wedge \exists T'.\, T' \xrightarrow{\mathsf{WR}(x)} T \wedge T' \xrightarrow{\mathsf{WW}(x)} S.$$

PROPOSITION 3.3. *For any* $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO}) \in \mathsf{ExecSI}$,

$$\mathsf{graph}(\mathcal{X}) = (\mathcal{T}, \mathsf{SO}, \mathsf{WR}_{\mathcal{X}}, \mathsf{WW}_{\mathcal{X}}, \mathsf{RW}_{\mathcal{X}})$$

is a dependency graph.

Note that the constraints on WR in Definition 3.2 ensure that it uniquely determines the values read by transactions. For $\mathcal{H} = (\mathcal{T}, \mathsf{SO})$ we write $(\mathcal{H}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$ for $(\mathcal{T}, \mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$.

We write $\mathcal{T} \models \textsc{Int}$ if a set of transactions $\mathcal{T}$ satisfies the internal consistency axiom INT in Figure 1. A relation $\mathcal{R} \subseteq \mathcal{T} \times \mathcal{T}$ is ***acyclic*** if $\mathcal{R}^+ \cap \{(T, T) \mid T \in \mathcal{T}\} = \emptyset$, where R^+ is the transitive closure of R. (Strong session) serializability can be characterised by the set of acyclic dependency graphs with internally consistent transactions [Adya, 1999].

THEOREM 3.4. *Let*

$$\mathsf{GraphSER} = \{\mathcal{G} \mid (\mathcal{T}_{\mathcal{G}} \models \textsc{Int}) \wedge ((\mathsf{SO}_{\mathcal{G}} \cup \mathsf{WR}_{\mathcal{G}} \cup \mathsf{WW}_{\mathcal{G}} \cup \mathsf{RW}_{\mathcal{G}}) \textit{ is acyclic})\}.$$

Then

$$\mathsf{HistSER} = \{\mathcal{H} \mid \exists \mathsf{WR}, \mathsf{WW}, \mathsf{RW}.\, (\mathcal{H}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{GraphSER}\}.$$

For example, the histories in Figures 2(b), 2(e) and 2(f) are not serializable, and they cannot be extended to acyclic dependency graphs; in particular, the graphs $\mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3$, shown with bold edges in figure 2(b), 2(e) and 2(f), respectively, satisfy the conditions of Definition 3.2, but contain cycles. We now set out to find a characterisation of the above form for SI.

4. SI CHARACTERISATION

For a set $\mathcal{T}$ and a relation $\mathcal{R} \subseteq \mathcal{T} \times \mathcal{T}$ let $\mathcal{R}? = \mathcal{R} \cup \{(T, T) \mid T \in \mathcal{T}\}$. We show that (strong session) SI is characterised by dependency graphs that contain only cycles with at least two adjacent anti-dependency edges.

THEOREM 4.1. *Let*

$$\mathsf{GraphSI} = \{\mathcal{G} \mid (\mathcal{T}_\mathcal{G} \models \textsc{Int}) \wedge (((\mathsf{SO}_\mathcal{G} \cup \mathsf{WR}_\mathcal{G} \cup \mathsf{WW}_\mathcal{G}) \,;\, \mathsf{RW}_\mathcal{G}?) \textit{ is acyclic})\}.$$

Then

$$\mathsf{HistSI} = \{\mathcal{H} \mid \exists \mathsf{WR}, \mathsf{WW}, \mathsf{RW}.\, (\mathcal{H}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{GraphSI}\}.$$

According to the theorem, to determine whether a particular history is allowed by SI, we can look for dependencies that extend it to a graph in GraphSI. As we demonstrate in §5 and §6, this way of defining SI is particularly suitable for developing static analyses for this consistency model. The history in Figure 2(f) is allowed by SI, and indeed the dependency graph $\mathcal{G}_3$ shown in the figure contains only cycles with two adjacent anti-dependencies (e.g., $T_1 \xrightarrow{\mathsf{RW}} T_2 \xrightarrow{\mathsf{RW}} T_1$). In contrast, the histories in Figures 2(b) and 2(e) are not allowed by SI, and they cannot be extended to graphs where every cycle has at least two adjacent anti-dependencies. In particular, the graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ shown in Figures 2(b) and 2(e), respectively, contain cycles without these: e.g., $T_1 \xrightarrow{\mathsf{WW}} T_2 \xrightarrow{\mathsf{RW}} T_1$ in Figure 2(b) and $T_1 \xrightarrow{\mathsf{WR}} T_3 \xrightarrow{\mathsf{RW}} T_2 \xrightarrow{\mathsf{WR}} T_4 \xrightarrow{\mathsf{RW}} T_1$ in Figure 2(e).

To prove Theorem 4.1, we prove a slightly stronger result, showing that we can establish a correspondence between executions in ExecSI and graphs in GraphSI that preserves histories and dependencies.

THEOREM 4.2.

(i) ***Soundness:*** $\forall \mathcal{G} \in \mathsf{GraphSI}.\, \exists \mathcal{X} \in \mathsf{ExecSI}.\, \mathsf{graph}(\mathcal{X}) = \mathcal{G}$.

(ii) ***Completeness:*** $\forall \mathcal{X} \in \mathsf{ExecSI}.\, \mathsf{graph}(\mathcal{X}) \in \mathsf{GraphSI}$.

As we explain in §7, the easier completeness direction of this theorem actually follows from existing results [Fekete et al., 2005]. Our main technical contribution is the more challenging proof of the soundness direction, which is required for the static analyses that we propose (§5 and §6).

4.1. Relationships between Abstract Executions and Dependency Graphs

We start by establishing several relationships between abstract executions in ExecSI and dependency graphs in GraphSI that are useful in proving Theorem 4.2. We introduce them by example.

First, consider the dependency graph $\mathcal{G}_4$ in Figure 3 and an abstract execution $\mathcal{X} \in \mathsf{ExecSI}$ such that $\mathsf{graph}(\mathcal{X}) = \mathcal{G}_4$. We show that we cannot have $T_4 \xrightarrow{\mathsf{VIS}_\mathcal{X}} T_2$. Indeed, if we had $T_4 \xrightarrow{\mathsf{VIS}_\mathcal{X}} T_2$, then by Definition 3.1 we would have $T_4 \xrightarrow{\mathsf{WR}_{\mathcal{G}_4}(x)} T_2$, contradicting the hypothesis that $T_3 \xrightarrow{\mathsf{WR}_{\mathcal{G}_4}(x)} T_2$. Hence, $\neg(T_4 \xrightarrow{\mathsf{VIS}_\mathcal{X}} T_2)$. This fact is not unique to the dependency graph $\mathcal{G}_4$, as we show in the following proposition.

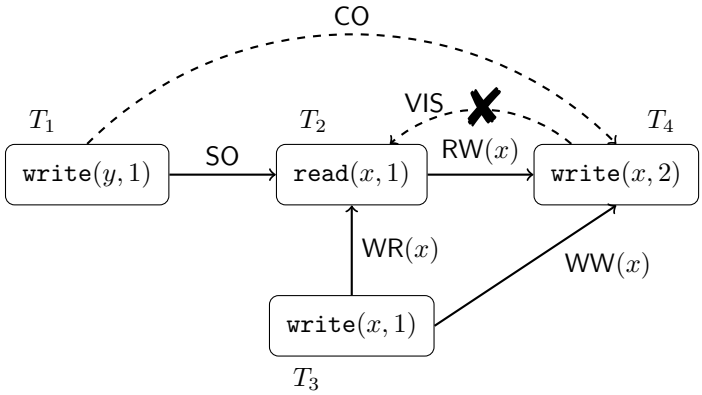


Fig. 3. An illustration of a correspondence between abstract executions and dependency graphs.

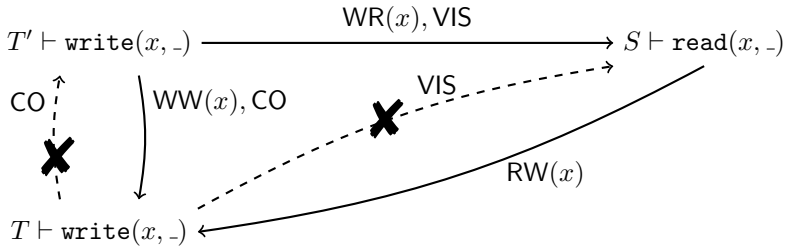


Fig. 4. An illustration of the proof of Proposition 4.3.

PROPOSITION 4.3.

$$\forall \mathcal{X} \in \mathsf{ExecSI}.\, \forall T, S \in \mathcal{T}_{\mathcal{X}}.\, S \xrightarrow{\mathsf{RW}_{\mathcal{X}}} T \iff$$
$$S \neq T \wedge \exists x.\, S \vdash \mathtt{read}(x, \_) \wedge T \vdash \mathtt{write}(x, \_) \wedge \neg(T \xrightarrow{\mathsf{VIS}_{\mathcal{X}}} S).$$

PROOF. Fix $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO}) \in \mathsf{ExecSI}$ and $T, S \in \mathcal{T}$. Let $\mathsf{graph}(\mathcal{X}) = (\mathcal{T}, \mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$. We illustrate the proof in Figure 4.

"$\Longrightarrow$". Assume $S \xrightarrow{\mathsf{RW}} T$. Then for some $x \in \mathsf{Obj}$ and $T' \in \mathcal{T}$ we have $T' \xrightarrow{\mathsf{WR}(x)} S$ and $T' \xrightarrow{\mathsf{WW}(x)} T$. This implies $S \neq T$, $S \vdash \mathtt{read}(x, \_)$, $T \vdash \mathtt{write}(x, \_)$ and $T' = \max_{\mathsf{CO}}(\mathsf{VIS}^{-1}(S) \cap \mathsf{WriteTx}_x)$. Since $T' \xrightarrow{\mathsf{WW}(x)} T$, we have $T' \xrightarrow{\mathsf{CO}} T$, and since $T \vdash \mathtt{write}(x, \_)$, we have $T \in \mathsf{WriteTx}_x$. Together with $T' = \max_{\mathsf{CO}}(\mathsf{VIS}^{-1}(S) \cap \mathsf{WriteTx}_x)$, these statements imply $T \notin \mathsf{VIS}^{-1}(S)$, i.e., $\neg(T \xrightarrow{\mathsf{VIS}} S)$.

"$\Longleftarrow$". Assume $S \vdash \mathtt{read}(x, \_)$, $T \vdash \mathtt{write}(x, \_)$ and $\neg(T \xrightarrow{\mathsf{VIS}} S)$ for some $x \in \mathsf{Obj}$. Let T' be the unique transaction such that $T' \xrightarrow{\mathsf{WR}(x)} S$; then $T' \vdash \mathtt{write}(x, \_)$. Since CO is total, we must have one of $T' = T$, $T \xrightarrow{\mathsf{CO}} T'$ or $T' \xrightarrow{\mathsf{CO}} T$. We cannot have $T' = T$, since then we would get $T \xrightarrow{\mathsf{VIS}} S$ from $T' \xrightarrow{\mathsf{WR}(x)} S$. We also cannot have $T \xrightarrow{\mathsf{CO}} T'$, since then we would get $T \xrightarrow{\mathsf{VIS}} S$ by $T' \xrightarrow{\mathsf{WR}(x)} S$ and PREFIX. Therefore, $T' \xrightarrow{\mathsf{CO}} T$ and, hence, $T' \xrightarrow{\mathsf{WW}(x)} T$. But then $S \xrightarrow{\mathsf{RW}(x)} T$. □

$$\begin{cases} \mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW} \subseteq \mathsf{VIS} & \text{(S1)} \\ \mathsf{CO} \,;\, \mathsf{VIS} \subseteq \mathsf{VIS} & \text{(S2)} \\ \mathsf{VIS} \subseteq \mathsf{CO} & \text{(S3)} \\ \mathsf{CO} \,;\, \mathsf{CO} \subseteq \mathsf{CO} & \text{(S4)} \\ \mathsf{VIS} \,;\, \mathsf{RW} \subseteq \mathsf{CO} & \text{(S5)} \end{cases}$$

Fig. 5. Requirements on a pre-execution $\mathcal{P} = (\mathcal{H}, \mathsf{VIS}, \mathsf{CO})$ constructed from a dependency graph $\mathcal{G} = (\mathcal{H}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$.

We next show that Proposition 4.3 implies an interesting relationship among anti-dependencies, visibility and the commit order. Consider again the dependency graph $\mathcal{G}_4$ of Figure 3 and $\mathcal{X} \in \mathsf{ExecSI}$ such that $\mathsf{graph}(\mathcal{X}) = \mathcal{G}_4$. Even though transactions T_1 and T_4 access different objects, we cannot choose the commit order between them arbitrarily: we must have $T_1 \xrightarrow{\mathsf{CO}_\mathcal{X}} T_4$. This is because (Session) guarantees $T_1 \xrightarrow{\mathsf{VIS}_\mathcal{X}} T_2$. Then, if we had $T_4 \xrightarrow{\mathsf{CO}_\mathcal{X}} T_1$, then by (Prefix) we would have $T_4 \xrightarrow{\mathsf{VIS}_\mathcal{X}} T_2$. But this contradicts the edge $T_2 \xrightarrow{\mathsf{RW}_\mathcal{X}} T_4$ and Proposition 4.3, according to which $\neg(T_4 \xrightarrow{\mathsf{VIS}_\mathcal{X}} T_2)$. Again, the pattern illustrated in this example is not unique to the dependency graph $\mathcal{G}_4$, as we now show.

Lemma 4.4. $\forall \mathcal{X} \in \mathsf{ExecSI}.\, \mathsf{VIS}_\mathcal{X} \,;\, \mathsf{RW}_\mathcal{X} \subseteq \mathsf{CO}_\mathcal{X}$.

Proof. Consider $\mathcal{X} \in \mathsf{ExecSI}$ and $T, S', S \in \mathcal{T}_\mathcal{X}$ such that $S' \xrightarrow{\mathsf{VIS}_\mathcal{X}} S \xrightarrow{\mathsf{RW}_\mathcal{X}} T$. If $T = S'$, then $S' \xrightarrow{\mathsf{VIS}_\mathcal{X}} S \xrightarrow{\mathsf{RW}_\mathcal{X}} S'$, contradicting Proposition 4.3. If $T \xrightarrow{\mathsf{CO}_\mathcal{X}} S'$ then by Prefix we get $T \xrightarrow{\mathsf{VIS}_\mathcal{X}} S$, contradicting Proposition 4.3. Then, since $\mathsf{CO}_\mathcal{X}$ is total, we must have $S' \xrightarrow{\mathsf{CO}_\mathcal{X}} T$. □

4.2. Pre-Executions

The main challenge in the proof of Theorem 4.2(i) is to construct a total commit order in the desired execution $\mathcal{X}$ from the dependencies given by $\mathcal{G}$ while satisfying the SI axioms (Definition 2.4). We do this incrementally; at intermediate stages of the construction we get structures similar to abstract executions, but where the commit order can be partial.

Definition 4.5. A tuple $\mathcal{P} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO})$ is a ***pre-execution*** if it satisfies all the conditions of Definition 2.3, except CO is a strict partial order that may not be total. We let $\mathsf{PreExecSI}$ be the set of pre-executions satisfying the SI axioms (Figure 1):

$$\mathsf{PreExecSI} = \{\mathcal{P} \mid \mathcal{P} \models \textsc{Int} \wedge \textsc{Ext} \wedge \textsc{Session} \wedge \textsc{Prefix} \wedge \textsc{NoConflict}\}.$$

Thus, an execution is a pre-execution whose commit order is total. In the following, we apply the graph function of §3 also to pre-executions.

Proposition 4.6. *For any* $\mathcal{P} \in \mathsf{PreExecSI}$, $\mathsf{graph}(\mathcal{P})$ *is a dependency graph.*

Towards proving Theorem 4.2(i), in this section, for a dependency graph

$$\mathcal{G} = (\mathcal{T}, \mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{GraphSI}$$

we construct a pre-execution

$$\mathcal{P} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO}) \in \mathsf{PreExecSI}$$

such that $\mathsf{graph}(\mathcal{P}) = \mathcal{G}$. In §4.3 we show how to extend $\mathcal{P}$ to a desired execution $\mathcal{X} \in \mathsf{ExecSI}$.

We start by restating the requirements on the pre-execution $\mathcal{P}$ in a way more suitable for guiding its construction; these are given by the system of inequalities in Figure 5. First,

to ensure $\mathsf{graph}(\mathcal{P}) = \mathcal{G}$, by Definition 3.1 at the very least we must have $\mathsf{WR} \cup \mathsf{WW} \subseteq \mathsf{VIS}$. For $\mathcal{P}$ to satisfy the SESSION axiom we must also have $\mathsf{SO} \subseteq \mathsf{VIS}$. These two observations motivate (S1). This inequality also implies that $\mathcal{P}$ satisfies NOCONFLICT, since according to Definition 3.1, WW is total over transactions that write to a given object. Inequality (S2) is equivalent to PREFIX, and inequality (S3) states a relationship between VIS and CO inherited by Definition 4.5 from Definition 2.3. Inequality (S4) requires CO to be transitive; (S2) and (S3) ensure that so is VIS.

As we now show, (S5), motivated by Lemma 4.4, ensures the axiom EXT. More generally, the system of inequalities in Figure 5 can be used to ensure that a pre-execution $\mathcal{P} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO})$ satisfies all SI axioms save INT.

LEMMA 4.7. *Let $\mathcal{G} = (\mathcal{T}, \mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$ be a dependency graph such that $\mathcal{T} \models$ INT and $\mathsf{VIS}, \mathsf{CO} \subseteq \mathcal{T} \times \mathcal{T}$ be acyclic relations satisfying the system of inequalities in Figure 5. Then $\mathcal{P} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO})$ is a pre-execution such that $\mathcal{P} \in \mathsf{PreExecSI}$ and $\mathsf{graph}(\mathcal{P}) = \mathcal{G}$.*

PROOF. We only prove that $\mathcal{P} \models$ EXT and $\mathsf{WR}_{\mathcal{P}} = \mathsf{WR}$; discharging the other obligations is straightforward. Consider $S \in \mathcal{T}$ such that $S \vdash \mathtt{read}(x, n)$. Then there exists a unique T' such that $T' \xrightarrow{\mathsf{WR}(x)} S$. Let $T = \max_{\mathsf{CO}}(\mathsf{VIS}^{-1}(S) \cap \mathsf{WriteTx}_x)$. This is defined because: CO is acyclic; by (S1) and (S3) we have $\mathsf{WW} \subseteq \mathsf{CO}$, so that CO is total over $\mathsf{WriteTx}_x$; and by (S1) we have $\mathsf{WR} \subseteq \mathsf{VIS}$, so that $T' \in \mathsf{VIS}^{-1}(S) \cap \mathsf{WriteTx}_x$. Hence, either $T = T'$ or $T' \xrightarrow{\mathsf{CO}} T$. We now show that the the latter case is impossible. Since $T \xrightarrow{\mathsf{VIS}} S$, we cannot have $S \xrightarrow{\mathsf{RW}(x)} T$, for in this case (S5) would imply $T \xrightarrow{\mathsf{CO}} T$, contradicting the acyclicity of CO. Hence, $\neg(S \xrightarrow{\mathsf{RW}(x)} T)$. Since $S \vdash \mathtt{read}(x, \_)$, $T \vdash \mathtt{write}(x, \_)$ and $T' \xrightarrow{\mathsf{WR}(x)} S$, this can be the case only if $\neg(T' \xrightarrow{\mathsf{WW}(x)} T)$, which implies $\neg(T' \xrightarrow{\mathsf{CO}} T)$. Thus, we must have $T = T'$, which entails the required. □

According to Lemma 4.7, to construct a desired pre-execution $\mathcal{P} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO}) \in \mathsf{PreExecSI}$ from a dependency graph $\mathcal{G} = (\mathcal{T}, \mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$, it is sufficient to find a solution to the system of inequalities in Figure 5 in terms of *acyclic* relations VIS and CO. This is not completely trivial because of the recursive nature of the inequalities: according to them, adding more edges into VIS forces adding more edges into CO and vice versa, increasing the risk of tying a cycle. Our insight is to look for the solution that is *smallest* and, hence, least likely to contain cycles. The following lemma gives a closed form for this solution. In anticipation of using the lemma when extending a pre-execution to an execution, we state it in a generalised form that gives the smallest solution where CO contains at least a given set of edges R. We use * to denote the transitive and reflexive closure of a given relation.

LEMMA 4.8. *Let $\mathcal{G} = (\mathcal{T}, \mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$ be a dependency graph. For any $R \subseteq \mathcal{T} \times \mathcal{T}$, the relations*

$$\begin{aligned} \mathsf{VIS} &= (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^* \,;\, (\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}); \\ \mathsf{CO} &= (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^+ \end{aligned} \tag{1}$$

are a solution to the system of inequalities in Figure 5. They also are the smallest solution to the system for which $\mathsf{CO} \supseteq R$: for any other solution $(\mathsf{VIS}', \mathsf{CO}')$ with $\mathsf{CO}' \supseteq R$ we have $\mathsf{VIS} \subseteq \mathsf{VIS}'$ and $\mathsf{CO} \subseteq \mathsf{CO}'$.

PROOF. We first prove that the relations in the statement of the lemma are indeed a solution to the system of inequalities in Figure 5.

(S1).

$$\begin{aligned}\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW} &= (\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \\ &\subseteq (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^* \,;\, (\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \\ &= \mathsf{VIS}.\end{aligned}$$

(S2).

$$\begin{aligned}\mathsf{CO} \,;\, \mathsf{VIS} &= (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^+ \,; \\ &\quad (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^* \,;\, (\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \\ &= (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^+ \,;\, (\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \\ &\subseteq (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^* \,;\, (\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \\ &= \mathsf{VIS}.\end{aligned}$$

(S3).

$$\begin{aligned}\mathsf{VIS} &= (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^* \,;\, (\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \\ &\subseteq (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^* \,;\, ((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \\ &\subseteq (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^* \,;\, (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R) \\ &= (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^+ \\ &= \mathsf{CO}.\end{aligned}$$

(S4).

$$\begin{aligned}\mathsf{CO} \,;\, \mathsf{CO} &= (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^+ \,;\, (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^+ \\ &= (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^+ \\ &= \mathsf{CO}.\end{aligned}$$

(S5).

$$\begin{aligned}\mathsf{VIS} \,;\, \mathsf{RW} &= (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^* \,;\, (\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW} \\ &\subseteq (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^* \,;\, ((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \\ &\subseteq (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^* \,;\, (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R) \\ &= (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^+ \\ &= \mathsf{CO}.\end{aligned}$$

Next, consider VIS' and CO' such that they satisfy (S1)-(S5) and $R \subseteq \mathsf{CO}'$. We first show that $\mathsf{CO} \subseteq \mathsf{CO}'$. By (S1) we have

$$\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW} \subseteq \mathsf{VIS}'. \tag{2}$$

Then by (S3) and (S5) we get

$$(\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}? \subseteq \mathsf{CO}'.$$

Since $R \subseteq \mathsf{CO}'$, this implies

$$((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R \subseteq \mathsf{CO}',$$

and by (S4) we get

$$(((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^+ \subseteq (\mathsf{CO}')^+ \subseteq \mathsf{CO}'.$$

But this is exactly $\mathsf{CO} \subseteq \mathsf{CO}'$.

To prove that $\mathsf{VIS} \subseteq \mathsf{VIS}'$, we rewrite VIS as

$$\mathsf{VIS} = (\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \cup ((((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \,;\, \mathsf{RW}?) \cup R)^+ \,;\, (\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW})),$$

and we prove that both parameters of the union are included in VIS'. We have already proved (2). We also have

$$(((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \mathbin{;} \mathsf{RW}?) \cup R)^+ \mathbin{;} (\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) = \mathsf{CO} \mathbin{;} (\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}).$$

Since $\mathsf{CO} \subseteq \mathsf{CO}'$ and $\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW} \subseteq \mathsf{VIS}'$, by (S2) we get

$$\mathsf{CO} \mathbin{;} (\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \subseteq \mathsf{CO}' \mathbin{;} \mathsf{VIS}' \subseteq \mathsf{VIS}',$$

as required. □

For $R = \emptyset$, Lemma 4.8 gives the smallest solution to the system of inequalities in Figure 5, which we denote by $(\mathsf{VIS}_0, \mathsf{CO}_0)$. We now show that this solution gives us the pre-execution we originally set out to construct.

COROLLARY 4.9. *If*

$$\mathcal{G} = (\mathcal{T}, \mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{GraphSI},$$

then

$$\mathcal{P}_0 = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}_0, \mathsf{CO}_0) \in \mathsf{PreExecSI}$$

and $\mathsf{graph}(\mathcal{P}_0) = \mathcal{G}$.

PROOF. Since $\mathcal{G} \in \mathsf{GraphSI}$, CO_0 is acyclic. By (S3), so is VIS_0. The required then follows from Lemma 4.7. □

4.3. From Pre-Executions to Executions: Proof of Theorem 4.2(i)

Fix $\mathcal{G} = (\mathcal{T}, \mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{GraphSI}$. Let $\mathcal{P}_0 = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}_0, \mathsf{CO}_0)$ be the pre-execution constructed in Corollary 4.9, so that $\mathcal{P}_0 \in \mathsf{PreExecSI}$ and $\mathsf{graph}(\mathcal{P}_0) = \mathcal{G}$. To prove Theorem 4.2(i), we now show how to extend $\mathcal{P}$ to a desired execution $\mathcal{X} \in \mathsf{ExecSI}$ such that $\mathsf{graph}(\mathcal{P}) = \mathcal{G}$. To this end, we take an incremental approach: we construct a sequence of pre-executions

$$\{\mathcal{P}_i = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}_i, \mathsf{CO}_i)\}_{i=0}^{n} \subseteq \mathsf{PreExecSI}$$

for some $n \geq 0$. The sequence is such that $\mathsf{graph}(\mathcal{P}_i) = \mathcal{G}$ for $i = 0..n$, and $\mathsf{CO}_i \subset \mathsf{CO}_{i+1}$ for $i = 0..(n-1)$. Furthermore, CO_n is total, so that $\mathcal{P}_n$ is an execution.

To define the above sequence, assume that $\mathcal{P}_i$ has been defined. To construct $\mathcal{P}_{i+1}$, we choose two transactions (if any) T_i, S_i that are not related by CO_i and define $(\mathsf{VIS}_{i+1}, \mathsf{CO}_{i+1})$ to be the smallest solution to the system of inequalities of Figure 5 that contains $\mathsf{CO}_i \cup \{(T_i, S_i)\}$. If the above transactions T_i, S_i do not exist, we terminate the construction and let $n = i$. To prove that the constructed sequence satisfies the properties stated above, we need to show that CO_{i+1} is acyclic. This is established using the following lemma.

LEMMA 4.10. *Let $\mathcal{P}_i = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}_i, \mathsf{CO}_i) \in \mathsf{PreExecSI}$, where $(\mathsf{VIS}_i, \mathsf{CO}_i)$ is a solution to the system of inequalities in Figure 5. Assume that $T_i, S_i \in \mathcal{T}$ are two different transactions not related by CO_i, and $(\mathsf{VIS}_{i+1}, \mathsf{CO}_{i+1})$ is the smallest solution to the system of inequalities in Figure 5 such that $\mathsf{CO}_i \cup \{(T_i, S_i)\} \subseteq \mathsf{CO}_{i+1}$. Then CO_{i+1} is acyclic.*

PROOF. To prove the required, we exploit the fact that Lemma 4.8 can be used to express the relation CO_{i+1} as a function of CO_i and (T_i, S_i). Namely, by Lemma 4.8 we have:

$$\begin{aligned}
\mathsf{CO}_{i+1} &= (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \mathbin{;} \mathsf{RW}?) \cup (\mathsf{CO}_i \cup \{(T_i, S_i)\}))^+ \\
&= (((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW}) \mathbin{;} \mathsf{RW}?)^+ \cup (\mathsf{CO}_i \cup \{(T_i, S_i)\}))^+ \\
&= (\mathsf{CO}_0 \cup \mathsf{CO}_i \cup \{(T_i, S_i)\})^+.
\end{aligned}$$

We furthermore have $\mathsf{CO}_0 \subseteq \mathsf{CO}_i$, because $(\mathsf{VIS}_0, \mathsf{CO}_0)$ is the smallest solution of the system in Figure 5. Then

$$\mathsf{CO}_{i+1} = (\mathsf{CO}_i \cup \{(T_i, S_i)\})^+.$$

We now use this equality to show that CO_{i+1} must be acyclic. Reasoning by contradiction, suppose it has a cycle. Then there exists a sequence of transactions $T^0, \ldots, T^m$, where $m \geq 0$, such that $T^0 = T^m$, and

$$T^0 \xrightarrow{\mathsf{CO}_i \cup \{(T_i, S_i)\}} \cdots \xrightarrow{\mathsf{CO}_i \cup \{(T_i, S_i)\}} T^m.$$

Because $\mathcal{P}_i$ is a pre-execution, CO_i is acyclic, so in the sequence above there exists at least one index $k \in \{0, \ldots, m-1\}$ such that $T^k = T_i$ and $T^{k+1} = S_i$. Let j, h be the smallest and the largest such indices, respectively (note that we can have $j = h$). Then we can convert the above cycle into the following one:

$$T^0 \xrightarrow{\mathsf{CO}_i} \cdots \xrightarrow{\mathsf{CO}_i} T^j = T_i \xrightarrow{\{(T_i, S_i)\}} S_i = T^{h+1} \xrightarrow{\mathsf{CO}_i} \cdots \xrightarrow{\mathsf{CO}_i} T^m,$$

where there is exactly one arrow labelled with the relation $\{(T_i, S_i)\}$, and all other arrows are labelled with CO_i. Now it remains to note that we have

$$S_i = T^{h+1} \xrightarrow{\mathsf{CO}_i} \cdots \xrightarrow{\mathsf{CO}_i} T^m = T^0 \xrightarrow{\mathsf{CO}_i} \cdots \xrightarrow{\mathsf{CO}_i} T^j = T_i,$$

and because CO_i is transitive by (S3), we have $S_i \xrightarrow{\mathsf{CO}_i} T_i$. But this contradicts the assumption that S_i and T_i are not related by CO_i. Therefore, CO_{i+1} must be acyclic. □

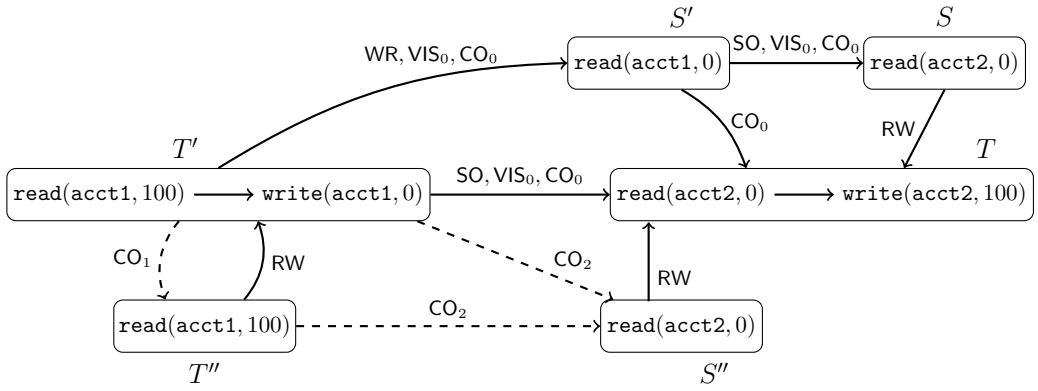
Proof of Theorem 4.2(i). Consider the sequence $\{\mathcal{P}_i = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}_i, \mathsf{CO}_i)\}_{i=0}^n$ constructed as described above. We prove by induction on i that $\mathcal{P}_i \in \mathsf{PreExecSI}$ and $\mathsf{graph}(\mathcal{P}_i) = \mathcal{G}$ for $i = 0..n$. If $i = 0$, then $\mathcal{P}_0 \in \mathsf{PreExecSI}$ and $\mathsf{graph}(\mathcal{P}_0) = \mathcal{G}$ by Corollary 4.9. Assume $\mathcal{P}_i \in \mathsf{PreExecSI}$ for $0 \leq i < n$. Then by Lemma 4.10, CO_{i+1} is acyclic. Hence, by Lemma 4.7, $\mathcal{P}_{i+1} \in \mathsf{PreExecSI}$ and $\mathsf{graph}(\mathcal{P}_{i+1}) = \mathcal{G}$, which completes the induction. Finally, by construction CO_n is total, so that $\mathcal{X} = \mathcal{P}_n$ is an abstract execution such that $\mathsf{graph}(\mathcal{P}_n) = \mathcal{G}$, as required. □

We now illustrate the construction in the proof of Theorem 4.2(i) by an example. Consider the dependency graph $\mathcal{G}_5 \in \mathsf{GraphSI}$ in Figure 6, which we also use as an example in §5. Its transactions could arise from the programs, also shown in the figure, that make a transfer between two accounts and query their balances or the sum thereof. The transactions arising from `lookup1` and `lookup2` see the initial state of the database, while the transactions arising from `lookupAll` see its state in the middle of a transfer.

By Lemma 4.8, the smallest solution $(\mathsf{VIS}_0, \mathsf{CO}_0)$ of the system of inequalities in Figure 5 consists of the solid edges in the Figure 6. In particular, we have $S' \xrightarrow{\mathsf{CO}_0} T$ because of $S' \xrightarrow{\mathsf{SO}} S$ (so that $S' \xrightarrow{\mathsf{VIS}} S$), $S \xrightarrow{\mathsf{RW}} T$ and the inequality (S5). Since CO_0 is not total, we have to pick an arbitrary pair of transactions (T_1, S_1) unrelated by CO_0 and construct VIS_1 and CO_1 as the smallest solution to the system of inequalities in Figure 5 such that $\mathsf{CO}_1 \supseteq \mathsf{CO}_0 \cup \{(T_1, S_1)\}$. By Lemma 4.8 this solution is given by (1) for $R = \mathsf{CO}_0 \cup \{(T_1, S_1)\}$. For example, in Figure 6 the transactions T' and T'' are unrelated by the commit order. If we pick as (T_1, S_1) the pair (T', T'') in Figure 6, then we get $\mathsf{VIS}_1 = \mathsf{VIS}_0$, and $\mathsf{CO}_1 = \mathsf{CO}_0 \cup \{(T', T'')\}$. The relation CO_1 is not yet total: for example, it does not relate the transactions T'' and S''. By picking as (T_2, S_2) the pair (T'', S''), we construct VIS_2 and CO_2 by letting $R = \mathsf{CO}_1 \cup \{(T'', S'')\}$ in (1); this corresponds to all the solid and dashed edges in Figure 6. Note that CO_2 also includes the edge (T', S''). Since CO_2 is total in this example, the construction terminates with $n = 2$.

4.4. Proof of Theorem 4.2(ii)

Consider $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO}) \in \mathsf{ExecSI}$. As follows from Lemma 4.4, VIS and CO give a solution to the system of inequalities of Figure 5 for $\mathsf{WR} = \mathsf{WR}_{\mathcal{X}}$, $\mathsf{WW} = \mathsf{WW}_{\mathcal{X}}$, $\mathsf{RW} = \mathsf{RW}_{\mathcal{X}}$.




```
T′, T: session transfer  {
         tx { acct1 = acct1 - 100 }; tx { acct2 = acct2 + 100 } }
  T″: session lookup1   { tx { return acct1 } }
  S″: session lookup2   { tx { return acct2 } }
S′, S: session lookupAll {
         tx { var1 = acct1 }; tx { var2 = acct2 }; return var1 + var2 }
```

Fig. 6. An illustration of constructing an execution from a dependency graph (§4) and splicing an execution (§5). We omit an initialisation transaction that sets `acct1` $= 100$ and `acct2` $= 0$.

We now apply Lemma 4.8 for $R = \emptyset$; the minimality of the solution given by Lemma 4.8 implies that $((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW})\ ;\ \mathsf{RW}?)^+ \subseteq \mathsf{CO}$. Then $((\mathsf{SO} \cup \mathsf{WR} \cup \mathsf{WW})\ ;\ \mathsf{RW}?)^+$ is acyclic because so is CO. This establishes $\mathsf{graph}(\mathcal{X}) \in \mathsf{GraphSI}$. $\square$

4.5. More Precise Characterisations

In this section we present corollaries of Theorem 4.1 that give more precise SI characterisations. We use these in our static analyses (§5 and §6).

For the following, it is helpful to introduce some notation. We use $\gamma, \gamma', \ldots$ to range over cycles in a dependency graph $\mathcal{G}$, i.e., paths of the form

$$T_1 \xrightarrow{\mathcal{R}_1} T_2 \xrightarrow{\mathcal{R}_2} \cdots \xrightarrow{\mathcal{R}_{n-1}} T_n \tag{3}$$

such that $T_1 = T_n$ and for any $i = 1..n$ we have $T_i \in \mathcal{T}_\mathcal{G}$ and $\mathcal{R}_i \in \{\mathsf{SO}_\mathcal{G}, \mathsf{WR}_\mathcal{G}, \mathsf{WW}_\mathcal{G}, \mathsf{RW}_\mathcal{G}\}$. For a given cycle γ of the form (3), we let

$$\mathsf{rep}(\gamma) = |\{T_i \mid \exists j.\, 1 \leq i, j < n \wedge i \neq j \wedge T_i = T_j\}|$$

be the number of repeated vertices in it. Note that γ has no repeated vertices if and only if $\mathsf{rep}(\gamma) = 0$. We call such a cycle ***simple***. Our first strengthening of the SI characterisation limits the scope of the acyclicity check in Theorem 4.1 to simple cycles.

THEOREM 4.11. *Let $\mathcal{G}$ be a dependency graph. Then $\mathcal{G} \in \mathsf{GraphSI}$ if and only if $\mathcal{G} \models$ INT and all simple cycles in $\mathcal{G}$ contain at least two adjacent anti-dependency edges.*

PROOF. The "only if" direction trivially follows from Theorem 4.1. For the "if" direction, let $\mathcal{G}$ be a dependency graph. We show that if $\mathcal{G}$ contains a cycle without adjacent $\mathsf{RW}_\mathcal{G}$ edges, then it also contains a *simple* cycle without adjacent $\mathsf{RW}_\mathcal{G}$ edges. To this end, let γ be a cycle in $\mathcal{G}$ without adjacent $\mathsf{RW}_\mathcal{G}$ edges. If $\mathsf{rep}(\gamma) = 0$, then there is nothing to prove, so let us assume that $\mathsf{rep}(\gamma) > 0$. Below we show how to extract a sub-cycle γ' of γ such that

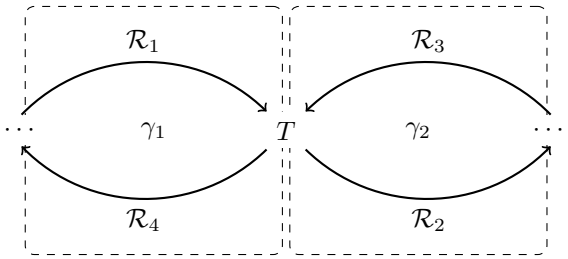


Fig. 7. A cycle with a repeated transaction T

$\mathsf{rep}(\gamma') < \mathsf{rep}(\gamma)$ and γ' has no adjacent $\mathsf{RW}_\mathcal{G}$ edges. By applying this procedure repeatedly, we obtain a cycle γ'' with no adjacent $\mathsf{RW}_\mathcal{G}$ edges and such that $\mathsf{rep}(\gamma'') = 0$, as required.

To construct γ' from γ, we proceed as follows: since $\mathsf{rep}(\gamma) > 0$, we have

$$\gamma = S \xrightarrow{} \ldots \xrightarrow{\mathcal{R}_1} T \xrightarrow{\mathcal{R}_2} \ldots \xrightarrow{\mathcal{R}_3} T \xrightarrow{\mathcal{R}_4} \ldots \xrightarrow{} S$$

for some $T, S \in \mathcal{T}_\mathcal{G}$ and $\{\mathcal{R}_i\}_{i=1}^4 \subseteq \{\mathsf{SO}_\mathcal{G}, \mathsf{WR}_\mathcal{G}, \mathsf{WW}_\mathcal{G}, \mathsf{RW}_\mathcal{G}\}$. A graphical representation of γ is given in Figure 7. From γ we can derive the cycles

$$\gamma_1 = S \xrightarrow{} \cdots \xrightarrow{\mathcal{R}_1} T \xrightarrow{\mathcal{R}_4} \cdots \xrightarrow{} S$$

and

$$\gamma_2 = T \xrightarrow{\mathcal{R}_2} \cdots \xrightarrow{\mathcal{R}_3} T,$$

which are contained inside the dashed boxes in Figure 7.

Since γ contains no adjacent $\mathsf{RW}_\mathcal{G}$ edges, we can have two adjacent $\mathsf{RW}_\mathcal{G}$ edges in γ_1 only if $\mathcal{R}_1 = \mathsf{RW}_\mathcal{G}$ and $\mathcal{R}_4 = \mathsf{RW}_\mathcal{G}$; similarly, we can have two adjacent $\mathsf{RW}_\mathcal{G}$ edges in γ_2 only if $\mathcal{R}_2 = \mathsf{RW}_\mathcal{G}$ and $\mathcal{R}_3 = \mathsf{RW}_\mathcal{G}$. Therefore, if either $\mathcal{R}_1 \neq \mathsf{RW}_\mathcal{G}$ or $\mathcal{R}_2 \neq \mathsf{RW}_\mathcal{G}$, then we know that γ_1 has no adjacent $\mathsf{RW}_\mathcal{G}$ edges, and we choose $\gamma' = \gamma_1$. Otherwise $\mathcal{R}_1 = \mathsf{RW}_\mathcal{G}$. Since γ contains no adjacent $\mathsf{RW}_\mathcal{G}$ edges, it follows that $\mathcal{R}_2 \neq \mathsf{RW}_\mathcal{G}$; therefore γ_2 contains no adjacent $\mathsf{RW}_\mathcal{G}$ edges, and we choose $\gamma' = \gamma_2$. □

Next, we show another characterisation of GraphSI that involves the notion of *vulnerable* anti-dependencies [Fekete et al., 2005]. In our setting, we define these as the anti-dependency edges that are not covered by the other types of edges. Formally, in a dependency graph $\mathcal{G}$ an anti-dependency edge $T \xrightarrow{\mathsf{RW}_\mathcal{G}} S$ is ***vulnerable*** if we do not have $T \xrightarrow{\mathsf{SO}_\mathcal{G} \cup \mathsf{WR}_\mathcal{G} \cup \mathsf{WW}_\mathcal{G}} S$.

THEOREM 4.12. *Let $\mathcal{G}$ be a dependency graph. Then $\mathcal{G} \in \mathsf{GraphSI}$ if and only if $\mathcal{G} \models \textsc{Int}$ and all simple cycles in $\mathcal{G}$ contain at least two adjacent vulnerable anti-dependency edges.*

PROOF. First note that

$$((\mathsf{SO}_\mathcal{G} \cup \mathsf{WR}_\mathcal{G} \cup \mathsf{WW}_\mathcal{G}) \,;\, \mathsf{RW}_\mathcal{G}?)^+ = \\ ((\mathsf{SO}_\mathcal{G} \cup \mathsf{WR}_\mathcal{G} \cup \mathsf{WW}_\mathcal{G}) \,;\, (\mathsf{RW}_\mathcal{G} \setminus (\mathsf{SO}_\mathcal{G} \cup \mathsf{WR}_\mathcal{G} \cup \mathsf{WW}_\mathcal{G}))?)^+.$$

Then by Theorem 4.1 and the above equality, $\mathcal{G} \in \mathsf{GraphSI}$ if and only if $\mathcal{G} \models \textsc{Int}$ and all cycles in $\mathcal{G}$ contain at least two adjacent vulnerable anti-dependency edges. We can now proceed as in the proof of Theorem 4.13 to obtain the required. □

So far, none of our SI characterisations has considered the objects involved in dependencies between transactions. The following theorem takes these into account.

THEOREM 4.13. *Let $\mathcal{G}$ be a dependency graph. Then $\mathcal{G} \in \mathsf{GraphSI}$ if and only if $\mathcal{G} \models \textsc{Int}$ and all simple cycles in $\mathcal{G}$ contain at least two adjacent vulnerable anti-dependency edges over different objects.*

PROOF. The "if" part of the theorem is trivial. For the "only if" part, suppose that $\mathcal{G} \in \mathsf{GraphSI}$. We prove that

$$\forall x.\, \mathsf{RW}_{\mathcal{G}}(x) \,;\, \mathsf{RW}_{\mathcal{G}}(x) \subseteq \mathsf{RW}_{\mathcal{G}}(x) \,;\, (\mathsf{RW}_{\mathcal{G}}(x) \cap \mathsf{WW}_{\mathcal{G}}(x)), \tag{4}$$

This implies that, whenever we have two adjacent anti-dependencies over the same object in $\mathcal{G}$, then at least one of them is not vulnerable. Since $\mathcal{G} \in \mathsf{GraphSI}$, by Theorem 4.12 the graph has only simple cycles with two adjacent vulnerable anti-dependencies, and by (4) such anti-dependencies need to be over different objects.

To prove (4), consider $x \in \mathsf{Obj}$ and $T, T', T'' \in \mathcal{T}_{\mathcal{G}}$ such that $T \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} T' \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} T''$. By definition, $T' \vdash \mathtt{write}(x, \_)$, $T'' \vdash \mathtt{write}(x, \_)$ and $T' \neq T''$. Therefore, either $T' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T''$ or $T'' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T'$. However, we cannot have $T'' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T'$, because $T' \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} T''$ and the hypothesis $\mathcal{G} \in \mathsf{GraphSI}$ ensures that $\mathsf{WW}_{\mathcal{G}}(x) \,;\, \mathsf{RW}_{\mathcal{G}}(x)$ is acyclic. Hence, we must have $T' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T''$ and, therefore, $T \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} T' \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x) \cap \mathsf{WW}_{\mathcal{G}}(x)} T''$. □

As an illustration of Theorem 4.13, the dependency graph of the write skew anomaly in Figure 2(f), allowed by SI, contains only cycles with at least two adjacent vulnerable anti-dependencies over different objects: e.g., $T_1 \xrightarrow{\mathsf{RW}(\mathtt{acct2})} T_2 \xrightarrow{\mathsf{RW}(\mathtt{acct1})} T_1$.

5. TRANSACTION CHOPPING UNDER SI

In this section, we exploit our characterisation of SI in terms of dependency graphs to derive a static analysis that checks when transactions in an application executing under SI can be ***chopped*** [Shasha et al., 1995] into sessions of smaller transactions without introducing new behaviours (the sessions are also called ***chains*** in this context [Zhang et al., 2013]). To this end, the analysis must check that any SI execution of the application with chopped transactions can be *spliced* into an SI execution that has the same operations as the original one, but where all operations from each session are executed inside a single transaction. We first establish a ***dynamic chopping criterion*** that checks whether a single SI execution, represented by a dependency graph, is spliceable. From this we then derive a static analysis that checks whether this is the case for all executions produced by a given chopped application.

For a history $\mathcal{H}$, let

$$\approx_{\mathcal{H}} = \mathsf{SO}_{\mathcal{H}} \cup \mathsf{SO}_{\mathcal{H}}^{-1} \cup \{(T, T) \mid T \in \mathcal{T}_{\mathcal{H}}\}$$

be the equivalence relation grouping transactions from the same session. We let $\boxed{T}_{\mathcal{H}}$ be the result of splicing all transactions in the session to which T belongs in $\mathcal{H}$ into a single transaction: $\boxed{T}_{\mathcal{H}} = (E, \mathsf{po})$, where $E = (\bigcup\{E_S \mid S \approx_{\mathcal{H}} T\})$ and

$$\mathsf{po} = \{(e, f) \mid (\exists S.\, e, f \in E_S \wedge e \xrightarrow{\mathsf{po}_S} f \wedge S \approx_{\mathcal{H}} T) \vee (\exists S, S'.\, e \in E_S \wedge f \in E_{S'} \wedge S \xrightarrow{\mathsf{SO}_{\mathcal{H}}} S' \wedge S' \approx_{\mathcal{H}} T)\}.$$

We let $\mathsf{splice}(\mathcal{H})$ be the history resulting from splicing all sessions in a history $\mathcal{H}$:

$$\mathsf{splice}(\mathcal{H}) = \left(\left\{\boxed{T}_{\mathcal{H}} \mid T \in \mathcal{T}_{\mathcal{H}}\right\}, \emptyset\right).$$

A dependency graph $\mathcal{G} \in \mathsf{GraphSI}$ is ***spliceable*** if there exists a dependency graph $\mathcal{G}' \in \mathsf{GraphSI}$ such that $\mathcal{H}_{\mathcal{G}'} = \mathsf{splice}(\mathcal{H}_{\mathcal{G}})$. For a dependency graph $\mathcal{G}$, we let $\approx_{\mathcal{G}} = \approx_{\mathcal{H}_{\mathcal{G}}}$ and $\boxed{T}_{\mathcal{G}} = \boxed{T}_{\mathcal{H}_{\mathcal{G}}}$.

For example, the graph $\mathcal{G}_5$ in Figure 6 is not spliceable, because $\mathsf{splice}(\mathcal{H}_{\mathcal{G}_5}) \notin \mathsf{HistSI}$: informally, $\boxed{S}_{\mathcal{G}_5}$ observes the write by $\boxed{T}_{\mathcal{G}_5}$ to `acct1`, but not its write to `acct2`. On the other hand, let $\mathcal{G}_6$ be the graph obtained by removing the transactions S and S' from $\mathcal{G}_5$. Then $\mathcal{G}_6$ is spliceable, as witnessed by the graph $\mathcal{G}'_6 \in \mathsf{GraphSI}$ with $\mathcal{H}_{\mathcal{G}'_6} = \mathsf{splice}(\mathcal{H}_{\mathcal{G}_6})$ and only the edges $\boxed{T''}_{\mathcal{G}_6} \xrightarrow{\mathsf{RW}_{\mathcal{G}'_6}} \boxed{T}_{\mathcal{G}_6}$ and $\boxed{S''}_{\mathcal{G}_6} \xrightarrow{\mathsf{RW}_{\mathcal{G}'_6}} \boxed{T}_{\mathcal{G}_6}$.

Our criterion for checking that a dependency graph is spliceable requires the absence of certain cycles in a variant of the graph.

Definition 5.1. Given a dependency graph $\mathcal{G}$, the ***dynamic chopping graph*** corresponding to $\mathcal{G}$ is the graph $\mathsf{DCG}(\mathcal{G})$ with the node set $\mathcal{T}_{\mathcal{G}}$ and the edge set obtained from the union of the following sets:

— $\mathsf{SO}_{\mathcal{G}}$ (a ***successor*** edge);
— $\mathsf{SO}_{\mathcal{G}}^{-1}$ (a ***predecessor*** edge):
— $\mathsf{WR}_{\mathcal{G}} \setminus \approx_{\mathcal{G}}$ (a ***read dependency*** edge);
— $\mathsf{WW}_{\mathcal{G}} \setminus \approx_{\mathcal{G}}$ (a ***write dependency*** edge); and
— $\mathsf{RW}_{\mathcal{G}} \setminus \approx_{\mathcal{G}}$ (an ***anti-dependency*** edge).

We refer to the last three kinds of edges as ***conflict*** edges.

A cycle in a chopping graph $\mathsf{DCG}(\mathcal{G})$ is ***critical*** if:

(i) it is simple;
(ii) it contains a fragment of three consecutive edges of the form "conflict, predecessor, conflict"; and
(iii) any two anti-dependency edges on the cycle are separated by at least one read or write dependency edge.

THEOREM 5.2. *For $\mathcal{G} \in \mathsf{GraphSI}$, if $\mathsf{DCG}(\mathcal{G})$ contains no critical cycles, then $\mathcal{G}$ is spliceable.*

For example, the above graph $\mathcal{G}_6$ contains no critical cycles, and the graph $\mathcal{G}_5$ contains a critical cycle

$$T' \xrightarrow{\mathsf{WR}_{\mathcal{G}_5}} S' \xrightarrow{\mathsf{SO}_{\mathcal{G}_5}} S \xrightarrow{\mathsf{RW}_{\mathcal{G}_5}} T \xrightarrow{\mathsf{SO}^{-1}_{\mathcal{G}_5}} T'.$$

5.1. Proof of the Dynamic Chopping Criterion under SI

To prove Theorem 5.2, we exhibit a particular dependency graph $\mathsf{splice}(\mathcal{G})$ such that $\mathsf{splice}(\mathcal{G}) \in \mathsf{GraphSI}$ and $\mathcal{H}_{\mathsf{splice}(\mathcal{G})} = \mathsf{splice}(\mathcal{H}_{\mathcal{G}})$. We define read dependencies $\mathsf{WR}_{\mathsf{splice}(\mathcal{G})}$ by lifting those in $\mathsf{WR}_{\mathcal{G}}$ to spliced transactions:

$$\forall T, S \in \mathcal{T}_{\mathcal{G}}.\, \forall x \in \mathsf{Obj}.\ \boxed{T}_{\mathcal{G}} \xrightarrow{\mathsf{WR}_{\mathsf{splice}(\mathcal{G})}(x)} \boxed{S}_{\mathcal{G}} \iff \boxed{T}_{\mathcal{G}} \neq \boxed{S}_{\mathcal{G}} \wedge T \xrightarrow{\approx_{\mathcal{G}}\,;\,\mathsf{WR}_{\mathcal{G}}(x)\,;\,\approx_{\mathcal{G}}} S. \quad (5)$$

We define $\mathsf{WW}_{\mathsf{splice}(\mathcal{G})}$ similarly and derive $\mathsf{RW}_{\mathsf{splice}(\mathcal{G})}$ from $\mathsf{WR}_{\mathsf{splice}(\mathcal{G})}$ and $\mathsf{WW}_{\mathsf{splice}(\mathcal{G})}$ as in Definition 3.1. Then Theorem 5.2 is a corollary of the following theorem.

THEOREM 5.3. *For $\mathcal{G} \in \mathsf{GraphSI}$, if $\mathsf{DCG}(\mathcal{G})$ contains no critical cycles, then $\mathsf{splice}(\mathcal{G})$ is a dependency graph and $\mathsf{splice}(\mathcal{G}) \in \mathsf{GraphSI}$.*

To prove the above theorem, we first show that $\mathsf{RW}_{\mathsf{splice}(\mathcal{G})}$ can be decomposed into a form similar to (5).

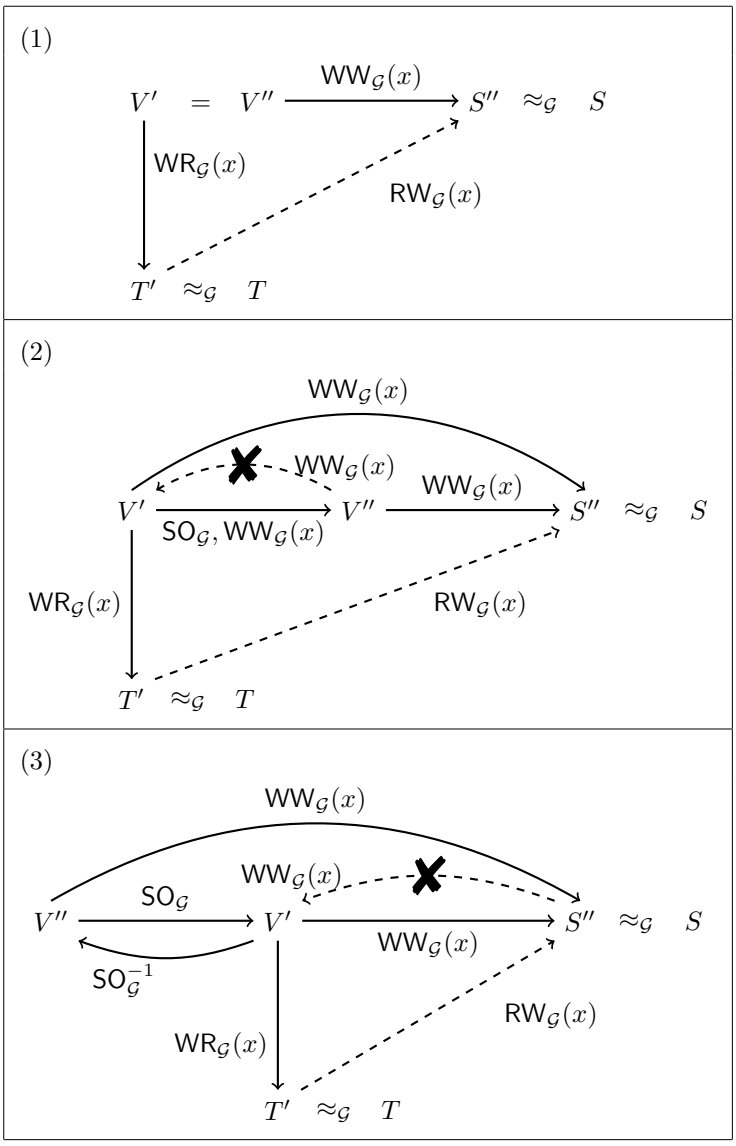


Fig. 8. Graphical representation of the different cases in the proof of Lemma 5.4.

LEMMA 5.4. *Let* $\mathcal{G} \in \mathsf{GraphSI}$ *be such that* $\mathsf{DCG}(\mathcal{G})$ *contains no critical cycles. Then*

$$\forall T, S \in \mathcal{T}_{\mathcal{G}}.\, \forall x \in \mathsf{Obj}.\, \boxed{T}_{\mathcal{G}} \xrightarrow{\mathsf{RW}_{\mathsf{splice}(\mathcal{G})}(x)} \boxed{S}_{\mathcal{G}} \implies \boxed{T}_{\mathcal{G}} \neq \boxed{S}_{\mathcal{G}} \wedge T \xrightarrow{\approx_{\mathcal{G}}\,;\,\mathsf{RW}_{\mathcal{G}}(x)\,;\,\approx_{\mathcal{G}}} S.$$

PROOF. Let $\mathcal{G} \in \mathsf{GraphSI}$ and suppose that $\boxed{T}_{\mathcal{G}} \xrightarrow{\mathsf{RW}_{\mathsf{splice}(\mathcal{G})}(x)} \boxed{S}_{\mathcal{G}}$ for some $T, S \in \mathcal{T}_{\mathcal{G}}$ and $x \in \mathsf{Obj}$. By definition, $\boxed{T}_{\mathcal{G}} \neq \boxed{S}_{\mathcal{G}}$ and there exists $V \in \mathcal{T}_{\mathcal{G}}$ such that $\boxed{V}_{\mathcal{G}} \xrightarrow{\mathsf{WR}_{\mathsf{splice}(\mathcal{G})}(x)} \boxed{T}_{\mathcal{G}}$ and $\boxed{V}_{\mathcal{G}} \xrightarrow{\mathsf{WW}_{\mathsf{splice}(\mathcal{G})}(x)} \boxed{S}_{\mathcal{G}}$. That is, there exist $T' \approx_{\mathcal{G}} T, V' \approx_{\mathcal{G}} V$, $V'' \approx_{\mathcal{G}} V$ and

$S'' \approx_{\mathcal{G}} S$ such that $V' \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} T'$ and $V'' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S''$. We show that $T' \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} S''$, so that $T \xrightarrow{\approx_{\mathcal{G}} ; \mathsf{RW}_{\mathcal{G}}(x) ; \approx_{\mathcal{G}}} S$.

First note that from $V' \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} T'$ and $V'' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S''$ we can infer $V' \vdash \mathtt{write}(x, \_)$, $V'' \vdash \mathtt{write}(x, \_)$ and $S'' \vdash \mathtt{write}(x, \_)$. Since $V' \approx_{\mathcal{G}} V \approx_{\mathcal{G}} V''$, one of the following must be true: $V' = V'', V' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} V''$ or $V'' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} V'$. We handle these three cases separately, as illustrated in Figure 8.

(1) $V' = V''$. Then $V' \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} T'$ and $V' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S''$, from which $T' \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} S''$ follows.

(2) $V' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} V''$. We have $V' \vdash \mathtt{write}(x, \_)$ and $V'' \vdash \mathtt{write}(x, \_)$. The assumption $\mathcal{G} \in \mathsf{GraphSI}$ mandates that $\neg(V'' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} V')$: otherwise we would have the cycle $V' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} V'' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} V'$, contradicting $\mathcal{G} \in \mathsf{GraphSI}$. Hence, $V' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} V''$. We thus have $V' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} V'' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S''$ and by transitivity of $\mathsf{WW}_{\mathcal{G}}(x)$ we obtain $V' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S''$. We have established $V' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S''$ and $V' \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} T'$, so that $T' \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} S''$.

(3) $V'' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} V'$. Since $V' \approx_{\mathcal{G}} V$, we have $\boxed{V'}_{\mathcal{G}} = \boxed{V}_{\mathcal{G}}$; similarly, $\boxed{S''}_{\mathcal{G}} = \boxed{S}_{\mathcal{G}}$. Therefore we have $\boxed{V'}_{\mathcal{G}} \xrightarrow{\mathsf{WW}_{\mathsf{splice}(\mathcal{G})}(x)} \boxed{S''}_{\mathcal{G}}$, which implies $\boxed{V'}_{\mathcal{G}} \neq \boxed{S''}_{\mathcal{G}}$, or equivalently $V' \not\approx_{\mathcal{G}} S''$. In particular $V' \neq S''$. We have already observed that $V' \vdash \mathtt{write}(x, \_)$ and $S'' \vdash \mathtt{write}(x, \_)$; since $V' \neq S''$, we must have either $V' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S''$ or $S'' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} V'$. However, in the latter case we would have the critical cycle $S'' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} V' \xrightarrow{\mathsf{SO}_{\mathcal{G}}^{-1}} V'' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S''$ in $\mathsf{DCG}(\mathcal{G})$, which contradicts the hypothesis of the lemma. Therefore, we must have $V' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S''$. Since we also have $V' \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} T'$, it follows that $T' \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} S''$.

□

We next show that $\mathsf{splice}(\mathcal{G})$ is a dependency graph (Lemma 5.6 below) and that $\mathcal{T}_{\mathsf{splice}(\mathcal{G})} \models \textsc{Int}$ (Lemma 5.7 below). The proof of the former fact relies on the following easy proposition.

PROPOSITION 5.5. *Let $\mathcal{G}$ be a dependency graph. For any $T \in \mathcal{T}_{\mathcal{G}}$:*

(1) $\boxed{T}_{\mathcal{G}} \vdash \mathtt{read}(x, n)$ *if and only if* $\min_{\mathsf{SO}_{\mathcal{G}}}\{S \mid S \approx_{\mathcal{G}} T \wedge S \vdash \_(x, \_)\} \vdash \mathtt{read}(x, n)$*;*

(2) $\boxed{T}_{\mathcal{G}} \vdash \mathtt{write}(x, n)$ *if and only if* $\max_{\mathsf{SO}_{\mathcal{G}}}\{S \mid S \approx_{\mathcal{G}} T \wedge S \vdash \mathtt{write}(x, \_)\} \vdash \mathtt{write}(x, n)$.

LEMMA 5.6. *Let $\mathcal{G} \in \mathsf{GraphSI}$ be such that $\mathsf{DCG}(\mathcal{G})$ contains no critical cycles. Then $\mathsf{splice}(\mathcal{G})$ is a dependency graph.*

PROOF. We prove that, if the chopping graph of $\mathcal{G}$ contains no critical cycles, then $\mathsf{splice}(\mathcal{G})$ satisfies all the constraints of Definition 3.2.

(1a). Consider $x \in \mathsf{Obj}$ and $\boxed{T}_{\mathcal{G}}, \boxed{S}_{\mathcal{G}} \in \mathcal{T}_{\mathsf{splice}(\mathcal{G})}$ such that $\boxed{T}_{\mathcal{G}} \xrightarrow{\mathsf{WR}_{\mathsf{splice}(\mathcal{G})}(x)} \boxed{S}_{\mathcal{G}}$. By definition $\boxed{T}_{\mathcal{G}} \neq \boxed{S}_{\mathcal{G}}$ and there exist two transactions $T', S' \in \mathcal{T}_{\mathcal{G}}$ such that $T \approx_{\mathcal{G}} T' \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} S' \approx_{\mathcal{G}} S$. Hence, for some n we have $T' \vdash \mathtt{write}(x, n)$ and $S' \vdash \mathtt{read}(x, n)$. We now prove that *(i)* $\boxed{T}_{\mathcal{G}} \vdash \mathtt{write}(x, n)$ and *(ii)* $\boxed{S}_{\mathcal{G}} \vdash \mathtt{read}(x, n)$.

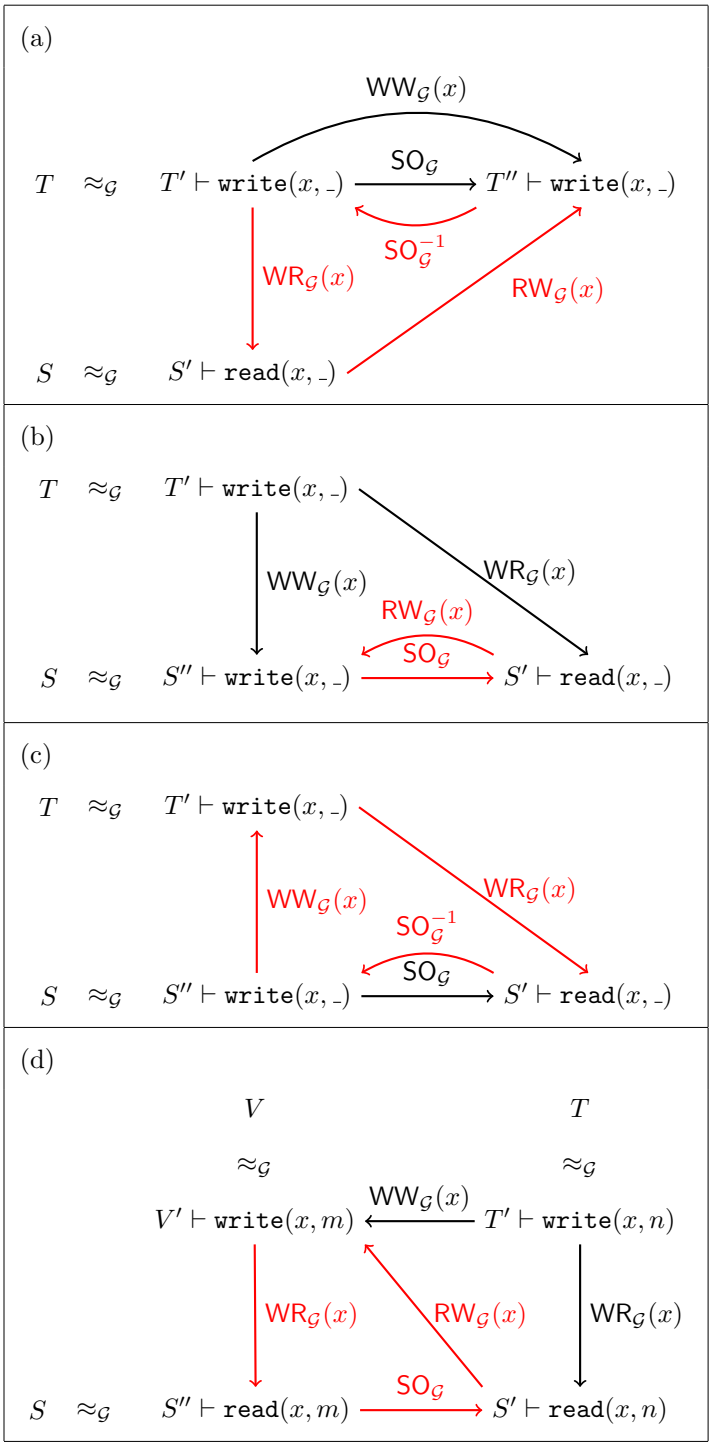


Fig. 9. Graphical representation of the different cases in the proof of Lemma 5.6.

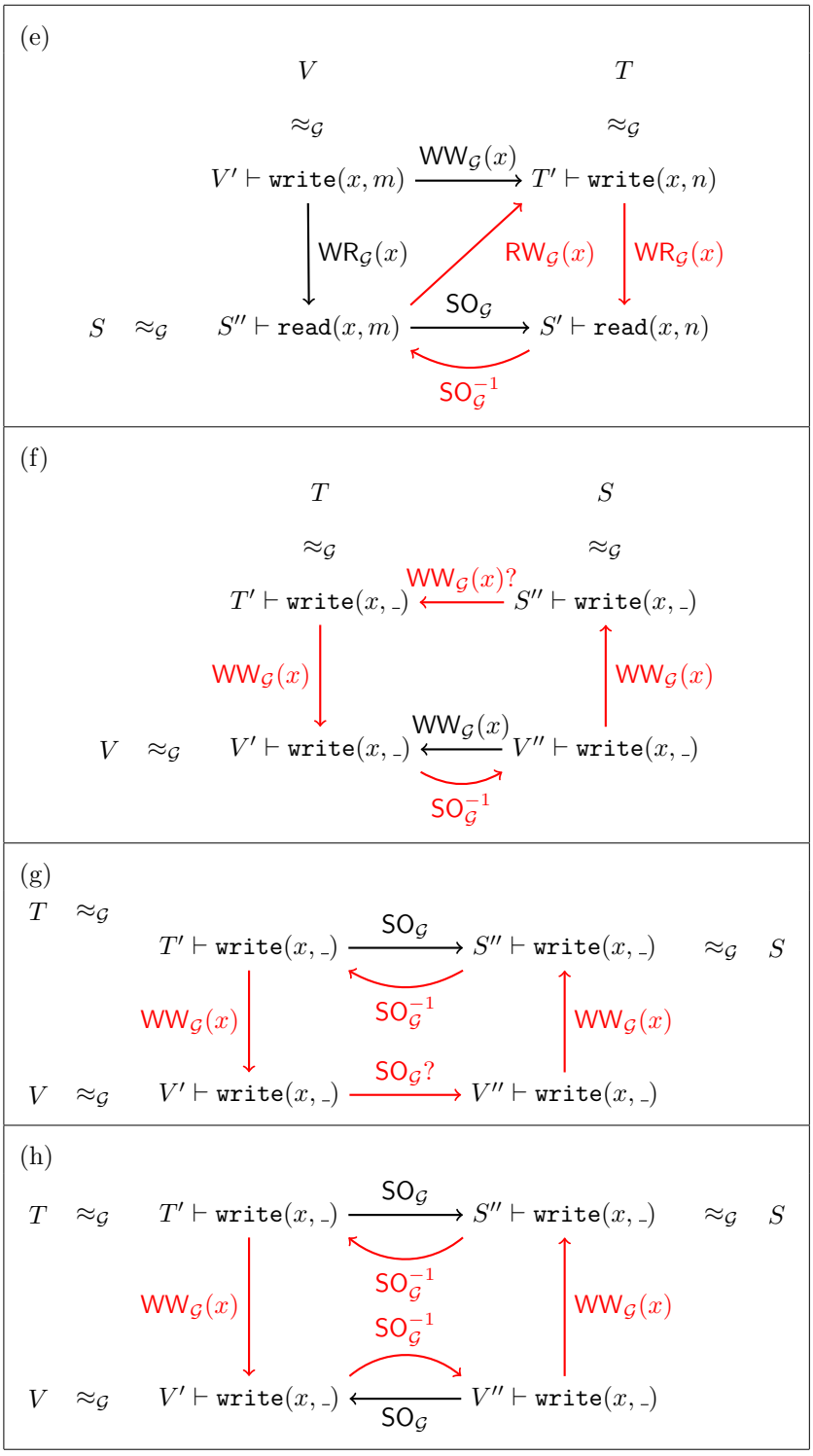


Fig. 10. Graphical representation of the different cases in the proof of Lemma 5.6.

(i). Consider an arbitrary transaction $T'' \in \mathcal{T}$ such that $T'' \vdash \texttt{write}(x, \_)$ and $T'' \approx_{\mathcal{G}} T'$ (Figure 9(a)). We show that it cannot be the case that $T' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T''$. Then $T' = \max_{\mathsf{SO}_{\mathcal{G}}}\{S \mid S \approx_{\mathcal{G}} T \wedge S \vdash \texttt{write}(x, \_)\}$, and by Proposition 5.5(ii) it follows that $\boxed{T}_{\mathcal{G}} \vdash \texttt{write}(x, n)$, as required.

Assume $T' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T''$; then $T' \neq T''$. Since $T', T'' \vdash \texttt{write}(x, \_)$, either $T' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T''$ or $T'' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T'$. However, the latter case would lead to the cycle $T'' \xrightarrow{\mathsf{WW}_{\mathcal{G}}} T' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T''$, which cannot exist because $\mathcal{G} \in \mathsf{GraphSI}$. Therefore $T' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T''$. Together with $T' \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} S'$, this yields the anti-dependency $S' \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} T''$. Since $S' \approx_{\mathcal{G}} S \not\approx_{\mathcal{G}} T \approx_{\mathcal{G}} T''$, this implies $S' \xrightarrow{(\mathsf{RW}_{\mathcal{G}}(x) \setminus \approx_{\mathcal{G}})} T''$. We have thus obtained the cycle $T' \xrightarrow{(\mathsf{WR}_{\mathcal{G}} \setminus \approx_{\mathcal{G}})} S' \xrightarrow{(\mathsf{RW}_{\mathcal{G}} \setminus \approx_{\mathcal{G}})} T'' \xrightarrow{\mathsf{SO}_{\mathcal{G}}^{-1}} T'$ in $\mathsf{DCG}(\mathcal{G})$, which is critical. This contradicts the assumption of the lemma.

(ii). We show that, for any transaction $S'' \approx_{\mathcal{G}} S'$ such that $S'' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} S'$ we have $\neg(S'' \vdash \texttt{write}(x, \_))$, and if $S'' \vdash \texttt{read}(x, m)$, then $m = n$. As a consequence, $\min_{\mathsf{SO}_{\mathcal{G}}}(\{V \mid V \approx_{\mathcal{G}} S \wedge V \vdash \_(x, \_)\} \vdash \texttt{read}(x, n))$, and hence, by Proposition 5.5(i) we have $\boxed{S}_{\mathcal{G}} \vdash \texttt{read}(x, n)$.

Let S'' be a transaction such that $S'' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} S'$; we prove that $\neg(S'' \vdash \texttt{write}(x, \_))$ by contradiction. Assume $S'' \vdash \texttt{write}(x, \_)$. Then by the definition of $\mathsf{WW}_{\mathcal{G}}(x)$, either $T' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S''$ or $S'' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T'$; the case $S'' = T'$ is ruled out because $S'' \approx_{\mathcal{G}} S \not\approx_{\mathcal{G}} T \approx_{\mathcal{G}} T'$.

We cannot have $T' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S''$ (Figure 9(b)), since together with $T' \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} S'$, this would imply the anti-dependency edge $S' \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} S''$. But then we have a cycle $S' \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} S'' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} S'$, contradicting $\mathcal{G} \in \mathsf{GraphSI}$. We cannot have $S'' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T'$ either (Figure 9(c)): in this case the chopping graph of $\mathcal{G}$ contains the critical cycle $S'' \xrightarrow{(\mathsf{WW}_{\mathcal{G}} \setminus \approx_{\mathcal{G}})} T' \xrightarrow{(\mathsf{WR}_{\mathcal{G}} \setminus \approx_{\mathcal{G}})} S' \xrightarrow{\mathsf{SO}_{\mathcal{G}}^{-1}} S''$. We have thus established $\neg(S'' \vdash \texttt{write}(x, \_))$.

Suppose now that $S'' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} S'$ and $S'' \vdash \texttt{read}(x, m)$ for some m. Then there exists a transaction $V' \in \mathcal{T}_{\mathcal{G}}$ such that $V' \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} S''$ and $V' \vdash \texttt{write}(x, m)$. Since $T' \vdash \texttt{write}(x, n)$, we have $V' = T'$, $V \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T'$ or $T' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} V$. We show that the latter two cases are impossible, so that $T' = V'$ and, hence, $m = n$, as required. If $T' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} V'$, then we have the anti-dependency edge $S' \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} V'$ (Figure 9(d)). Then the graph $\mathcal{G}$ contains the cycle $S' \xrightarrow{\mathsf{RW}_{\mathcal{G}}} V' \xrightarrow{\mathsf{WR}_{\mathcal{G}}} S'' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} S'$, which contradicts $\mathcal{G} \in \mathsf{GraphSI}$. If $V' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T'$, then we have the anti-dependency edge $S'' \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} T'$ (Figure 10(e)). In this case $\mathsf{DCG}(\mathcal{G})$ contains the critical cycle $S'' \xrightarrow{(\mathsf{RW}_{\mathcal{G}} \setminus \approx_{\mathcal{G}})} T' \xrightarrow{(\mathsf{WR}_{\mathcal{G}} \setminus \approx_{\mathcal{G}})} S' \xrightarrow{\mathsf{SO}_{\mathcal{G}}^{-1}} S''$, and again we obtain a contradiction.

(1b). Let $x \in \mathsf{Obj}$ and $\boxed{S}_{\mathcal{G}} \in \mathcal{T}_{\mathsf{splice}(\mathcal{G})}$ be such that $\boxed{S}_{\mathcal{G}} \vdash \texttt{read}(x, \_)$. By Proposition 5.5(i), there exists a transaction $S' \approx_{\mathcal{G}} S$ such that $S' \vdash \texttt{read}(x, \_)$. Since $\mathcal{G}$ is a dependency graph, there exists a transaction $T \in \mathcal{T}_{\mathcal{G}}$ such that $T \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} S'$. Then $T \neq S'$. We cannot have $T \xrightarrow{\mathsf{SO}_{\mathcal{G}}} S'$, because $T \vdash \texttt{write}(x, \_)$ and $S' = \min_{\mathsf{SO}}\{S \mid S \approx_{\mathcal{G}}$

$S' \wedge S \vdash \_(x,\_)\}$ by Proposition 5.5(i). Finally, we cannot have $S' \xrightarrow{\mathsf{SO}_\mathcal{G}} T$, because this would contradict the hypothesis $\mathcal{G} \in \mathsf{GraphSI}$ due to the cycle $S' \xrightarrow{\mathsf{SO}_\mathcal{G}} T \xrightarrow{\mathsf{WR}_\mathcal{G}(x)} S'$. As a consequence, it cannot be $T \approx_\mathcal{G} S'$. Then $\boxed{T}_\mathcal{G} \xrightarrow{\mathsf{WR}_{\mathsf{splice}(\mathcal{G})}(x)} \boxed{S}_\mathcal{G}$.

(1c). Let $x \in \mathsf{Obj}$ and $\boxed{S}_\mathcal{G}, \boxed{T}_\mathcal{G}, \boxed{V}_\mathcal{G} \in \mathcal{T}_{\mathsf{splice}(\mathcal{G})}$ be such that $\boxed{T}_\mathcal{G} \xrightarrow{\mathsf{WR}_{\mathsf{splice}(\mathcal{G})}(x)} \boxed{S}_\mathcal{G}$ and $\boxed{V}_\mathcal{G} \xrightarrow{\mathsf{WR}_{\mathsf{splice}(\mathcal{G})}(x)} \boxed{S}_\mathcal{G}$. Then $S \not\approx_\mathcal{G} T$, $S \not\approx_\mathcal{G} V$ and there exist transactions S', S'', T', V' such that $S' \approx_\mathcal{G} S \approx_\mathcal{G} S''$, $T' \xrightarrow{\mathsf{WR}_\mathcal{G}(x)} S'$, $V' \xrightarrow{\mathsf{WR}_\mathcal{G}(x)} S''$. Note that if $S' = S''$, then $T' = V'$, because $\mathcal{G}$ is a dependency graph; hence $\boxed{T}_\mathcal{G} = \boxed{T'}_\mathcal{G} = \boxed{V'}_\mathcal{G} = \boxed{V}_\mathcal{G}$, and there is nothing left to prove.

It remains to analyse the case when $S' \neq S''$, so that either $S' \xrightarrow{\mathsf{SO}_\mathcal{G}} S''$ or $S'' \xrightarrow{\mathsf{SO}_\mathcal{G}} S'$. Without loss of generality, assume that $S'' \xrightarrow{\mathsf{SO}_\mathcal{G}} S'$ (Figure 9(d)). Since $T' \xrightarrow{\mathsf{WR}_\mathcal{G}(x)} S'$ and $V' \xrightarrow{\mathsf{WR}_\mathcal{G}(x)} S''$, we get $T' \vdash \mathtt{write}(x,\_)$ and $V' \vdash \mathtt{write}(x,\_)$. Therefore, we must have one of the following: $T' = V'$, $T' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} V'$ or $V' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} T'$. However, the latter two cases are impossible. If we had $T' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} V'$, then $\mathcal{G}$ would contain the configuration shown in Figure 9(d), which contradicts $\mathcal{G} \in \mathsf{GraphSI}$. If we had $V' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} T'$, then $\mathcal{G}$ would contain the configuration shown in Figure 10(e) and, hence, $\mathsf{DCG}(\mathcal{G})$ would contain a critical cycle. Hence, we must have $T' = V'$ and $\boxed{T}_\mathcal{G} = \boxed{V}_\mathcal{G}$.

(2). We prove that $\mathsf{WW}_{\mathsf{splice}(\mathcal{G})}$ is transitive, irreflexive and total over $\mathsf{WriteTx}_x$.

$\mathsf{WW}_{\mathsf{splice}(\mathcal{G})}$ *is transitive.* Let $x \in \mathsf{Obj}$ and $\boxed{T}_\mathcal{G}, \boxed{V}_\mathcal{G}, \boxed{S}_\mathcal{G} \in \mathcal{T}_{\mathsf{splice}(\mathcal{G})}$ be such that $\boxed{T}_\mathcal{G} \xrightarrow{\mathsf{WW}_{\mathsf{splice}(\mathcal{G})}(x)} \boxed{V}_\mathcal{G} \xrightarrow{\mathsf{WW}_{\mathsf{splice}(\mathcal{G})}(x)} \boxed{S}_\mathcal{G}$. By definition there exist transactions $T', V', V'', S'' \in \mathcal{T}_\mathcal{G}$ such that

$$T \approx_\mathcal{G} T' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} V' \approx_\mathcal{G} V \approx_\mathcal{G} V'' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} S'' \approx_\mathcal{G} S.$$

To prove that $\boxed{T}_\mathcal{G} \xrightarrow{\mathsf{WW}_{\mathsf{splice}(\mathcal{G})}(x)} \boxed{S}_\mathcal{G}$, it suffices to prove that $T' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} S''$ and $T \not\approx_\mathcal{G} S$. We prove these two statements separately.

— Since $T' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} V'$ and $V'' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} S''$, we have $V' \vdash \mathtt{write}(x,\_)$ and $V'' \vdash \mathtt{write}(x,\_)$. Then one of the following holds: $V' = V''$, $V' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} V''$ or $V'' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} V'$. In the first two cases, the transitivity of $\mathsf{WW}_\mathcal{G}(x)$ guarantees $T' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} S''$, as required. Suppose now that $V'' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} V'$. Then $V'' \xrightarrow{\mathsf{SO}_\mathcal{G}} V'$, because $V'' \approx_\mathcal{G} V'$ and $\mathcal{G} \in \mathsf{GraphSI}$. Since $T' \vdash \mathtt{write}(x,\_)$ and $S'' \vdash \mathtt{write}(x,\_)$, we have one of the following: $T' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} S''$, $S'' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} T'$ or $T' = S''$. However, in the latter two cases we would end up with the chopping graph of Figure 10(f), which contains the critical cycle $T' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)\setminus\approx_\mathcal{G}} V' \xrightarrow{\mathsf{SO}_\mathcal{G}^{-1}} V'' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)\setminus\approx_\mathcal{G}} S'' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} T'$. Therefore, it has to be $T' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} S''$.

— We prove $T \not\approx_\mathcal{G} S$ by contradiction. Suppose that $T \approx_\mathcal{G} S$ and, hence, $T' \approx_\mathcal{G} S''$. We have $V' \vdash \mathtt{write}(x,\_)$ and $V'' \vdash \mathtt{write}(x,\_)$, so that one of the following holds: $V' = V''$, $V' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} V''$ or $V'' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} V'$.

In the first two cases, $\mathcal{G} \in \mathsf{GraphSI}$ guarantees $V' \xrightarrow{\mathsf{SO}_\mathcal{G}(x)?} V''$ and the transitivity of $\mathsf{WW}_\mathcal{G}(x)$ guarantees $T' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} S''$. Since $\mathsf{WW}_\mathcal{G}(x)$ is irreflexive, we cannot have $T' = S''$, and since $\mathcal{G} \in \mathsf{GraphSI}$, we cannot have $S'' \xrightarrow{\mathsf{SO}_\mathcal{G}} T'$. Therefore, $T' \xrightarrow{\mathsf{SO}_\mathcal{G}} S''$. Then, as illustrated in Figure 10(g), $\mathsf{DCG}(\mathcal{G})$ contains the critical cycle $T' \xrightarrow{(\mathsf{WW}_\mathcal{G} \setminus \approx_\mathcal{G})} V' \xrightarrow{\mathsf{SO}_\mathcal{G}?} V'' \xrightarrow{(\mathsf{WW}_\mathcal{G} \setminus \approx_\mathcal{G})} S'' \xrightarrow{\mathsf{SO}_\mathcal{G}^{-1}} T'$, yielding a contradiction.

It remains to consider the case when $V'' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} V'$. Then $V'' \xrightarrow{\mathsf{SO}_\mathcal{G}} V'$, because $V'' \approx_\mathcal{G} V'$ and $\mathcal{G} \in \mathsf{GraphSI}$. Since we are assuming $T' \approx_\mathcal{G} S''$, one of the following holds: $T' = S''$, $T' \xrightarrow{\mathsf{SO}_\mathcal{G}} S''$ or $S'' \xrightarrow{\mathsf{SO}_\mathcal{G}} T'$. In all cases there is a critical cycle in $\mathsf{DCG}(\mathcal{G})$. For example, if $T' \xrightarrow{\mathsf{SO}_\mathcal{G}} S''$, then $\mathcal{G}$ has a configuration shown in Figure 10(h), and $\mathsf{DCG}(\mathcal{G})$ contains the critical cycle $T' \xrightarrow{(\mathsf{WW}_\mathcal{G} \setminus \approx_\mathcal{G})} V' \xrightarrow{\mathsf{SO}_\mathcal{G}^{-1}} V'' \xrightarrow{(\mathsf{WW}_\mathcal{G} \setminus \approx_\mathcal{G})} S'' \xrightarrow{\mathsf{SO}_\mathcal{G}^{-1}} T'$.

$\mathsf{WW}_{\mathsf{splice}(\mathcal{G})}(x)$ *is irreflexive.* This follows immediately from the definition of $\mathsf{WW}_{\mathsf{splice}(\mathcal{G})}(x)$.

$\mathsf{WW}_{\mathsf{splice}(\mathcal{G})}(x)$ *is total over* $\mathsf{WriteTx}_x$. Let $x \in \mathsf{Obj}$ and $\boxed{T}_\mathcal{G}, \boxed{S}_\mathcal{G} \in \mathcal{T}_{\mathsf{splice}(\mathcal{G})}$ be such that $\boxed{T}_\mathcal{G} \vdash \mathtt{write}(x, \_)$, $\boxed{S}_\mathcal{G} \vdash \mathtt{write}(x, \_)$ and $\boxed{T}_\mathcal{G} \neq \boxed{S}_\mathcal{G}$. By Proposition 5.5(ii), there exist $T', S' \in \mathcal{T}_\mathcal{G}$ such that $T' \approx_\mathcal{G} T$, $T' \vdash \mathtt{write}(x, \_)$, $S' \approx_\mathcal{G} S$ and $S' \vdash \mathtt{write}(x, \_)$. Also, $T' \approx_\mathcal{G} T \not\approx_\mathcal{G} S \approx_\mathcal{G} S'$, so that it cannot be $T' = S'$. Since $\mathsf{WW}_\mathcal{G}(x)$ is total over $\mathsf{WriteTx}_x$, we must have either $T' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} S'$ or $S' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} T'$; without loss of generality, we assume $T' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} S'$. We thus have $\boxed{T}_\mathcal{G} \neq \boxed{S}_\mathcal{G}$ and $T \approx_\mathcal{G} T' \xrightarrow{\mathsf{WW}_\mathcal{G}(x)} S' \approx_\mathcal{G} S$. Hence, $\boxed{T}_\mathcal{G} \xrightarrow{\mathsf{WW}_{\mathsf{splice}(\mathcal{G})}(x)} \boxed{S}_\mathcal{G}$.

(3). This follows immediately from the definition of $\mathsf{RW}_{\mathsf{splice}(\mathcal{G})}$.

□

LEMMA 5.7. *Let* $\mathcal{G} \in \mathsf{GraphSI}$ *be such that* $\mathsf{DCG}(\mathcal{G})$ *contains no critical cycles. Then* $\mathcal{T}_{\mathsf{splice}(\mathcal{G})} \models \textsc{Int}$.

PROOF. Proceeding by contradiction, let us assume that INT is violated. Then there exist: $T \in \mathcal{T}_\mathcal{G}$; $e \in E_T$ such that $\mathsf{op}(e) = \mathtt{read}(x, n)$ for some $x \in \mathsf{Obj}, n \in \mathbb{N}$; and, letting $\boxed{\mathsf{po}_T} = \mathsf{po}_{\boxed{T}_\mathcal{G}}$,

$$f = \max_{\boxed{\mathsf{po}_T}} \{e' \mid \mathsf{op}(e') = \_(x, \_) \wedge e' \xrightarrow{\boxed{\mathsf{po}_T}} e\} \tag{6}$$

such that $\mathsf{op}(f) = \_(x, m)$ for some $m \neq n$. Since $\mathcal{T}_\mathcal{G} \models \textsc{Int}$, we cannot have $f \in E_T$. Therefore, there exists a transaction $T' \xrightarrow{\mathsf{SO}_\mathcal{G}} T$ such that $f \in E_{T'}$; see Figure 11(a). We now make a case split on whether $T' \vdash \mathtt{write}(x, \_)$.

(1) $T' \vdash \mathtt{write}(x, \_)$. Then there exists an event $g \in E_{T'}$ such that $\mathsf{op}(g) = \mathtt{write}(x, \_)$. Without loss of generality, let g be the last write to object x in $E_{T'}$. If $f \xrightarrow{\mathsf{po}_{T'}} g$, then $f \xrightarrow{\boxed{\mathsf{po}_T}} g \xrightarrow{\boxed{\mathsf{po}_T}} e$, contradicting (6). Thus, either $g = f$ or $g \xrightarrow{\mathsf{po}_{T'}} f$. In both cases, we show that $\mathsf{op}(g) = \mathtt{write}(x, m)$. If $f = g$, we have $\_(x, m) = \mathsf{op}(f) = \mathsf{op}(g) =$

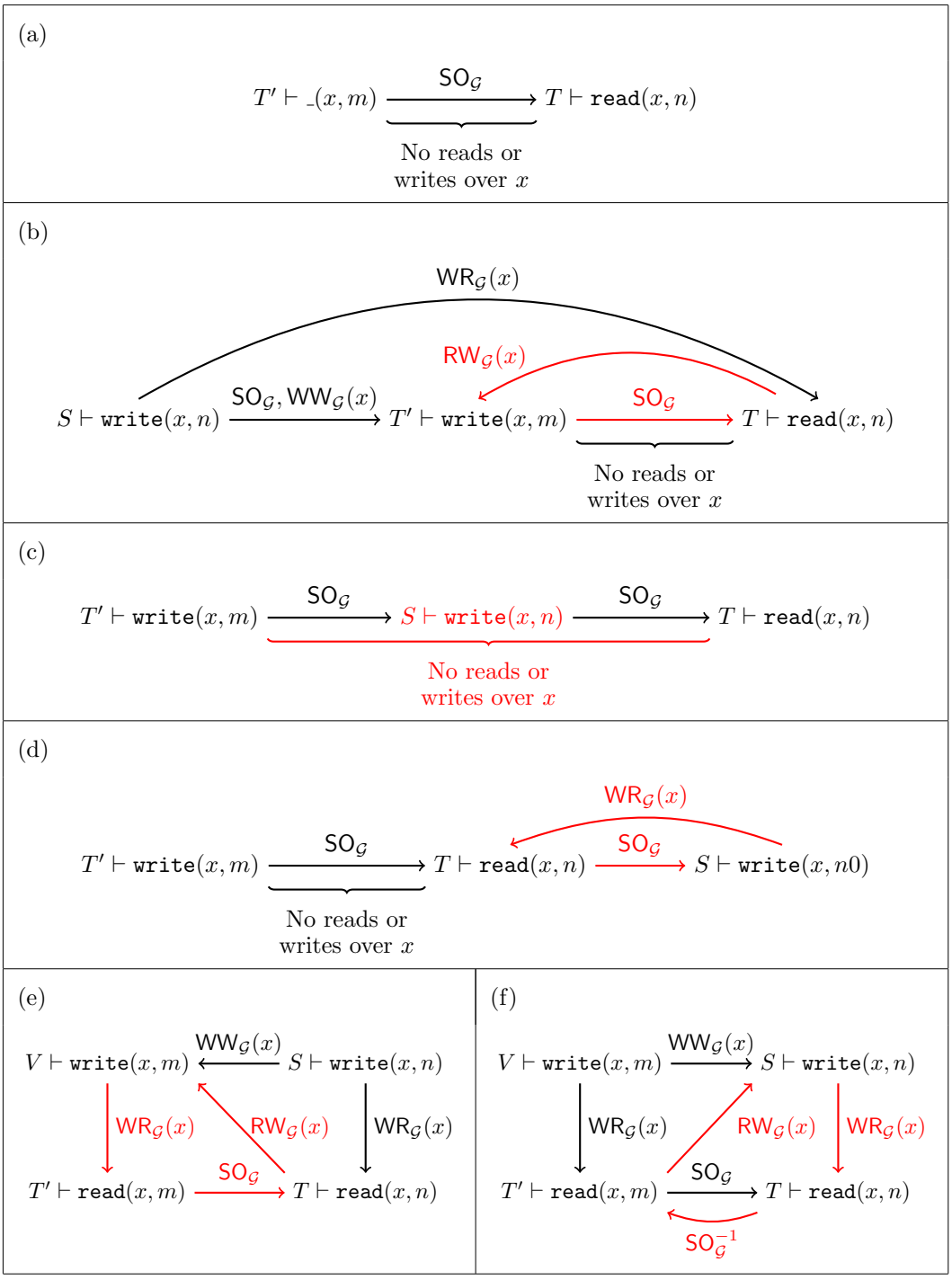


Fig. 11. Graphical representation of the different cases in the proof of Lemma 5.7.

$\texttt{write}(x, \_)$, so that $\mathsf{op}(g) = \texttt{write}(x, m)$. If $g \xrightarrow{\mathsf{po}_{T'}} f$, then $\mathsf{op}(f) = \texttt{read}(x, m)$, since g is the last write to x in T'. Also, for any other event event h such that $\mathsf{op}(h) = \_(x, \_)$ and $g \xrightarrow{\mathsf{po}_{T'}} h \xrightarrow{\mathsf{po}_{T'}} f$, we have $\mathsf{op}(h) = \texttt{read}(x, \_)$. Then because $\mathcal{T}_{\mathcal{G}} \models \textsc{Int}$ and $\mathsf{op}(f) = \texttt{read}(x, m)$, we must have $\mathsf{op}(h) = \texttt{read}(x, m)$. But then $\mathcal{T}_{\mathcal{G}} \models \textsc{Int}$ again ensures $\mathsf{op}(g) = \texttt{write}(x, m)$.

We have proved that $T' \vdash \texttt{write}(x, m)$. By hypothesis, $T \vdash \texttt{read}(x, n)$ for some $n \neq m$, so that there exists a transaction $S \neq T'$ such that $S \vdash \texttt{write}(x, n)$ and $S \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} T$.

Next we prove that $T' \not\approx_{\mathcal{G}} S$. Assume the contrary. Since $S \neq T'$ and $T' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T$, we must have one of the following:

$$S \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T; \quad T' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} S \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T; \quad T' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T \xrightarrow{\mathsf{SO}_{\mathcal{G}}} S.$$

We show that each of these cases leads to a contradiction.

— $S \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T$, Figure 11(b). Since $S, T' \vdash \texttt{write}(x, \_)$ and $S \neq T'$, we must have either $S \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T'$ or $T' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S$. However, the last case is impossible because it would lead to a cycle $S \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S$, contradicting $\mathcal{G} \in \mathsf{GraphSI}$; therefore, $S \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T'$. Together with $S \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} T$, this yields $T \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} T'$. But then we obtain a cycle $T' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} T'$, which again contradicts $\mathcal{G} \in \mathsf{GraphSI}$.

— $T' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} S \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T$, Figure 11(c). Since $S \vdash \texttt{write}(x, \_)$, there exists an event $h \in E_S$ such that $\mathsf{op}(h) = \texttt{write}(x, \_)$ and $f \xrightarrow{\boxed{\mathsf{po}_T}} h \xrightarrow{\boxed{\mathsf{po}_T}} e$. But this contradicts (6).

— $T' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T \xrightarrow{\mathsf{SO}_{\mathcal{G}}} S$, Figure 11(d). In this case we have the cycle $T \xrightarrow{\mathsf{SO}_{\mathcal{G}}} S \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} T$, which contradicts $\mathcal{G} \in \mathsf{GraphSI}$.

We have thus proved that $T' \not\approx_{\mathcal{G}} S$. Next, we observe that since $T' \vdash \texttt{write}(x, m)$, $S \vdash \texttt{write}(x, n)$ and $T' \neq S$, we must have either $T' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S$ or $S \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T'$. We show that both of these cases lead to a contradiction. If $S \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T'$, then since $S \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} T$, we get $T \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} T'$, causing the cycle $T \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} T' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T$; this contradicts $\mathcal{G} \in \mathsf{GraphSI}$. On the other hand, if $T' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S$, then we have a critical cycle $T' \xrightarrow{(\mathsf{WW}_{\mathcal{G}} \setminus \approx_{\mathcal{G}})} S \xrightarrow{(\mathsf{WR}_{\mathcal{G}} \setminus \approx_{\mathcal{G}})} T \xrightarrow{\mathsf{SO}_{\mathcal{G}}^{-1}} T'$ in $\mathsf{DCG}(\mathcal{G})$, contradicting the hypothesis of the lemma.

(2) $\neg(T' \vdash \texttt{write}(x, \_))$. In this case there exists no event $g \in E_{T'}$ such that $\mathsf{op}(g) = \texttt{write}(x, \_)$. Using the fact that $\mathcal{T}_{\mathcal{G}} \models \textsc{Int}$, we can easily show that for any $g \in E_{T'}$ such that $\mathsf{op}(g) = \texttt{read}(x, \_)$, we have $\mathsf{op}(g) = \texttt{read}(x, m)$. Then $T' \vdash \texttt{read}(x, m)$.

Since $\mathcal{G}$ is a dependency graph, there exist two transactions S, V such that $S \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} T$ and $V \xrightarrow{\mathsf{WR}_{\mathcal{G}}(x)} T'$. Since $S \vdash \texttt{write}(x, n)$ and $V \vdash \texttt{write}(x, m)$, we have $S \neq V$. Since $\neg(T' \vdash \texttt{write}(x, \_))$ and $S \vdash \texttt{write}(x, \_)$, we have $T' \neq S$. We also have $V \neq T$, for otherwise we would have a cycle $T' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T \xrightarrow{\mathsf{WR}_{\mathcal{G}}} T'$, contradicting $\mathcal{G} \in \mathsf{GraphSI}$. Hence, the transactions T', T, V, S are pairwise distinct.

We must have either $S \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} V$ or $V \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S$. We show that neither of these cases is possible. If $S \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} V$ (Figure 11(e)), then $T \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} V$. This causes a cycle $T \xrightarrow{\mathsf{RW}_{\mathcal{G}}} V \xrightarrow{\mathsf{WR}_{\mathcal{G}}} T' \xrightarrow{\mathsf{SO}_{\mathcal{G}}} T$, contradicting $\mathcal{G} \in \mathsf{GraphSI}$. If $V \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S$ (Figure 11(f)), then $T' \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} S$. As in the case above, we can show that $S \not\approx_{\mathcal{G}} T$. This yields a critical

cycle $T' \xrightarrow{(\mathsf{RW}_\mathcal{G} \setminus \approx_\mathcal{G})} S \xrightarrow{(\mathsf{WR}_\mathcal{G} \setminus \approx_\mathcal{G})} T \xrightarrow{\mathsf{SO}_\mathcal{G}^{-1}} T'$ in $\mathsf{DCG}(\mathcal{G})$, contradicting the assumptions of the lemma.

□

PROOF OF THEOREM 5.3. Let $\mathcal{G} \in \mathsf{GraphSI}$ be a dependency graph such that $\mathsf{DCG}(\mathcal{G})$ contains no critical cycles. We prove that $\mathsf{splice}(\mathcal{G}) \in \mathsf{GraphSI}$. First, Lemmas 5.6 and 5.7 ensure that $\mathsf{splice}(\mathcal{G})$ is indeed a dependency graph and $\mathcal{T}_{\mathsf{splice}(\mathcal{G})} \models \textsc{Int}$. Since $\mathsf{SO}_{\mathsf{splice}(\mathcal{G})} = \emptyset$, by Theorem 4.2 it remains to prove that the relation $((\mathsf{WR}_{\mathsf{splice}(\mathcal{G})} \cup \mathsf{WW}_{\mathsf{splice}(\mathcal{G})}) \; ; \; \mathsf{RW}_{\mathsf{splice}(\mathcal{G})}?)$ is acyclic. The proof goes by contradiction: we assume that this relation contains a cycle and exhibit a critical cycle in $\mathsf{DCG}(\mathcal{G})$. Let

$$\gamma = \boxed{T_0}_\mathcal{G} \xrightarrow{\mathcal{C}_0} \ldots \xrightarrow{\mathcal{C}_{n-1}} \boxed{T_n}_\mathcal{G} \quad (n \geq 1)$$

be a cycle in $\mathsf{splice}(\mathcal{G})$, where

$$T_0, \ldots, T_n \in \mathcal{T}_\mathcal{G}, \quad \mathcal{C}_0, \ldots \mathcal{C}_{n-1} \in \{\mathsf{WR}_{\mathsf{splice}(\mathcal{G})}, \mathsf{WW}_{\mathsf{splice}(\mathcal{G})}, \mathsf{RW}_{\mathsf{splice}(\mathcal{G})}\}$$

(the letter $\mathcal{C}$ stands for *conflict*), $\boxed{T_n}_\mathcal{G} = \boxed{T_0}_\mathcal{G}$ (in particular, $T_n \approx_\mathcal{G} T_0$), and there is no index $i = 0..(n-1)$ such that $\mathcal{C}_i = \mathsf{RW}_{\mathsf{splice}(\mathcal{G})}$ and $\mathcal{C}_{(i+1) \mod n} = \mathsf{RW}_{\mathsf{splice}(\mathcal{G})}$. By Theorem 4.11, we can assume that γ is simple. Thus, for any $i, j = 0..(n-1)$ we have $\boxed{T_i}_\mathcal{G} = \boxed{T_j}_\mathcal{G}$ (equivalently, $T_i \approx_\mathcal{G} T_j$) only if $i = j$.

By applying Lemma 5.4, we can convert γ into the following path:

$$T_0' \approx_\mathcal{G} T_0'' \xrightarrow{\mathcal{C}_0^\mathcal{G}} T_1' \approx_\mathcal{G} T_1'' \xrightarrow{\mathcal{C}_2^\mathcal{G}} \ldots \xrightarrow{\mathcal{C}_{n-1}^\mathcal{G}} T_n' \approx_\mathcal{G} T_n'', \tag{7}$$

where for any $i = 0..n$, $T_i' \approx_\mathcal{G} T_i \approx_\mathcal{G} T_i''$ (note that because $T_n \approx_\mathcal{G} T_0$, this implies $T_n'' \approx_\mathcal{G} T_0'$), and for any $i = 0..(n-1)$, $\mathcal{C}_i^\mathcal{G}$ is the relation in $\mathcal{G}$ corresponding to the relation $\mathcal{C}_i$ in $\mathsf{splice}(\mathcal{G})$ (e.g., if $\mathcal{C}_i = \mathsf{WR}_{\mathsf{splice}(\mathcal{G})}$, then $\mathcal{C}_i^\mathcal{G} = (\mathsf{WR}_\mathcal{G} \setminus \approx_\mathcal{G})$). We also know that

$$\neg \exists i = 0..(n-1). \, (\mathcal{C}_i^\mathcal{G} = (\mathsf{RW}_\mathcal{G} \setminus \approx_\mathcal{G})) \wedge (\mathcal{C}_{(i+1) \mod n}^\mathcal{G} = (\mathsf{RW}_\mathcal{G} \setminus \approx_\mathcal{G})). \tag{8}$$

Since the cycle γ is simple, the only possibility for vertices to be repeated on the path (7) is when they are adjacent: $T_i' = T_i''$ for some $i = 0..(n-1)$. Recall that whenever $T \approx_\mathcal{G} S$, for some transaction $T, S \in \mathcal{T}_\mathcal{G}$, then one of the following holds: $T = S$, $T \xrightarrow{\mathsf{SO}_\mathcal{G}} S$ or $T \xrightarrow{\mathsf{SO}_\mathcal{G}^{-1}} S$. Also, we know that $T_n' \approx T_n \approx T_0 \approx T_0''$. Therefore, we can rewrite the path (7) as follows:

$$T_n' \xrightarrow{\mathcal{S}_0} T_0'' \xrightarrow{\mathcal{C}_1^\mathcal{G}} T_1' \xrightarrow{\mathcal{S}_1} T_1'' \xrightarrow{\mathcal{C}_2^\mathcal{G}} \ldots \xrightarrow{\mathcal{S}_{n-1}} T_{n-1}'' \xrightarrow{\mathcal{C}_{n-1}^\mathcal{G}} T_n', \tag{9}$$

where $\mathcal{S}_0, \ldots \mathcal{S}_{n-1} \in \{\mathsf{SO}_\mathcal{G}?, \mathsf{SO}_\mathcal{G}^{-1}\}$. In this cycle repeated vertices are always adjacent and connected by an $\mathsf{SO}_\mathcal{G}?$-edge. By removing such edges from the cycle we obtain a simple cycle, where all the occurrences of $\mathsf{SO}_\mathcal{G}?$-edges are actually $\mathsf{SO}_\mathcal{G}$-edges; this is a cycle in $\mathsf{DCG}(\mathcal{G})$. Due to (8), in this cycle any two anti-dependency edges are separated by a read- or write-dependency edge. To prove this cycle yields a critical cycle in $\mathsf{DCG}(\mathcal{G})$, it remains to show that there exists an index $i = 0..(n-1)$ such that $\mathcal{S}_i = \mathsf{SO}_\mathcal{G}^{-1}$. This holds because, if we had $\mathcal{S}_i = \mathsf{SO}_\mathcal{G}$ for all $i = 0..(n-1)$, then we would obtain a cycle in $((\mathsf{SO}_\mathcal{G} \cup \mathsf{WR}_\mathcal{G} \cup \mathsf{WW}_\mathcal{G}) \; ; \; \mathsf{RW}_\mathcal{G}?)^+$, contradicting the assumption that $\mathcal{G} \in \mathsf{GraphSI}$. □

Discussion. Our dependency graph characterisation of SI is instrumental in checking chopping correctness due to the ease of splicing a dependency graph (cf. (5)). Splicing abstract executions directly would be problematic. To illustrate this, consider the abstract

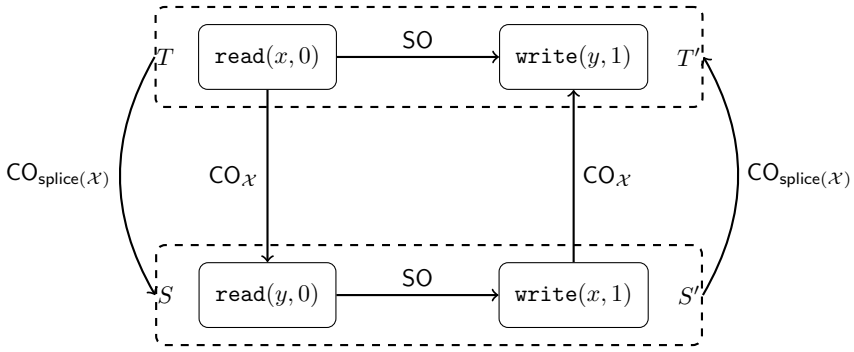


Fig. 12. An attempt to splice an execution directly.

execution $\mathcal{X}$ in Figure 12, which is in ExecSI. A straightforward way to define $\mathsf{splice}(\mathcal{X})$ is by letting

$$\boxed{T}_{\mathcal{X}} \xrightarrow{\mathsf{CO}_{\mathsf{splice}(\mathcal{X})}} \boxed{S}_{\mathcal{X}} \iff \exists T', S'.\, T \approx_{\mathcal{H}_{\mathcal{X}}} T' \xrightarrow{\mathsf{CO}_{\mathcal{X}}} S' \approx_{\mathcal{H}_{\mathcal{X}}} S,$$

and similarly for $\mathsf{VIS}_{\mathsf{splice}(\mathcal{X})}$. In this case we would have

$$\boxed{T}_{\mathcal{X}} \xrightarrow{\mathsf{CO}_{\mathsf{splice}(\mathcal{X})}} \boxed{S}_{\mathcal{X}} \xrightarrow{\mathsf{CO}_{\mathsf{splice}(\mathcal{X})}} \boxed{T}_{\mathcal{X}},$$

so that $\mathsf{CO}_{\mathsf{splice}(\mathcal{X})}$ is not irreflexive. Hence $\mathsf{splice}(\mathcal{X})$ is not a valid execution. On the other hand, by extracting a dependency graph $\mathcal{G}$ from $\mathcal{X}$ and computing $\mathsf{splice}(\mathcal{G})$, we easily obtain a dependency graph in GraphSI. This allows us to construct an execution $\mathcal{X}'$ with the dependency graph $\mathsf{splice}(\mathcal{G})$ such that $\mathcal{X}' \in \mathsf{ExecSI}$.

5.2. Static Analysis

We now derive a static analysis from Theorem 5.2. Assume a set of ***programs*** $\mathcal{P} = \{P_1, P_2, \ldots\}$, each defining the code of sessions resulting from chopping the code of a single transaction.

We leave the precise syntax of the programs unspecified, but assume that each P_i consists of k_i ***program pieces*** P_i^j $(j = 1..k_i)$, defining the code of the transactions in the sessions. We further assume that for each P_i we are given a sequence

$$(R_1^i, W_1^i)\,(R_2^i, W_2^i)\,\ldots\,(R_{k_i}^i, W_{k_i}^i), \tag{10}$$

of ***read and write sets*** $R_j^i, W_j^i \subseteq \mathsf{Obj}$, i.e., the sets of all objects that may be, respectively, read and written by the j-th piece of P_i. For example, the program `transfer` in Figure 6 consists of two pieces; the first one has the read and write sets equal to $\{$`acct1`$\}$ and the second, to $\{$`acct2`$\}$. The program `lookup1` consists of a single piece with the read set $\{$`acct1`$\}$ and the write set $\emptyset$.

Following Shasha et al. [Shasha et al., 1995], we make certain assumptions about the way clients execute programs. We assume that, if a transaction initiated by a program piece aborts, it will be resubmitted repeatedly until it commits, and, if a piece is aborted due to system failure, it will be restarted. We also assume that the client does not abort transactions explicitly.

Definition 5.8. A history $\mathcal{H} = (\mathcal{T}, \mathsf{SO})$ ***can be produced*** by programs

$$\mathcal{P} = \{(P_1^1, \ldots, P_1^{k_1}), \ldots, (P_i^1, \ldots, P_i^{k_i})\} \tag{11}$$

with read and write sets (10) if there exists a bijective function f from transactions in $\mathcal{T}$ to program pieces P_i^j of $\mathcal{P}$, such that

$$(\forall T \in \mathcal{T}.\, \forall x \in \mathsf{Obj}.\, \forall i, j.\, f(T) = P_i^j \wedge T \vdash \mathtt{read}(x, \_) \implies x \in R_i^j) \wedge$$
$$(\forall T \in \mathcal{T}.\, \forall x \in \mathsf{Obj}.\, \forall i, j.\, f(T) = P_i^j \wedge T \vdash \mathtt{write}(x, \_) \implies x \in W_i^j) \wedge$$
$$(\forall T, S \in \mathcal{T}.\, \forall i, j, h.\, f(T) = P_i^j \wedge f(S) = P_i^h \wedge T \xrightarrow{\mathsf{SO}} S \implies j < h).$$

For example, the history in Figure 6 can be produced by the programs in the figure.

Definition 5.9. The chopping defined by the programs $\mathcal{P}$ is ***correct*** if every dependency graph $\mathcal{G} \in \mathsf{GraphSI}$, where $\mathcal{H}_\mathcal{G}$ can be produced by $\mathcal{P}$, is spliceable.

We check the correctness of $\mathcal{P}$ by defining an analogue of the dynamic chopping graph from Definition 5.1, whose nodes are pieces of $\mathcal{P}$, rather than transactions in a given execution. Each piece is identified by a pair (i, j) of the number i of a program in $\mathcal{P}$ and the piece's position j in the program.

Definition 5.10. Given a set of programs (11) with read and write sets (10), its ***static chopping graph*** $\mathsf{SCG}(\mathcal{P})$ is the directed graph whose nodes are pairs of indices identifying the pieces in $\mathcal{P}$: $\{(i, j) \mid i = 1..|\mathcal{P}|,\ j = 1..k_i\}$. We have an edge $((i_1, j_1), (i_2, j_2))$ if and only if one of the following holds:

— $i_1 = i_2$ and $j_1 < j_2$ (a ***successor*** edge, S);
— $i_1 = i_2$ and $j_1 > j_2$ (a ***predecessor*** edge, P);
— $i_1 \neq i_2$ and $W_{i_1}^{j_1} \cap R_{i_2}^{j_2} \neq \emptyset$ (a ***read dependency*** edge, WR);
— $i_1 \neq i_2$ and $W_{i_1}^{j_1} \cap W_{i_2}^{j_2} \neq \emptyset$ (a ***write dependency*** edge, WW); or
— $i_1 \neq i_2$ and $R_{i_1}^{j_1} \cap W_{i_2}^{j_2} \neq \emptyset$ (an ***anti-dependency*** edge, RW).

The notion of a critical cycle introduced above for dynamic graphs is also applicable to static ones. The edge set of a static graph $\mathsf{SCG}(\mathcal{P})$ over-approximates the edge sets of dynamic graphs $\mathsf{DCG}(\mathcal{G})$ corresponding to dependency graphs $\mathcal{G}$ produced by the programs $\mathcal{P}$. From this observation and Theorem 5.2 we easily get our static analysis.

Corollary 5.11. *The chopping defined by $\mathcal{P}$ is correct if $\mathsf{SCG}(\mathcal{P})$ contains no critical cycles.*

In Figure 13 we show the static chopping graph of the programs $\mathcal{P}^1 = \{\mathtt{transfer}, \mathtt{lookupAll}\}$, which contains a critical cycle:

$$(\mathtt{var1} = \mathtt{acct1}) \xrightarrow{\mathsf{RW}} (\mathtt{acct1} = \mathtt{acct1} - 100) \xrightarrow{\mathsf{S}} (\mathtt{acct2} = \mathtt{acct2} + 100) \xrightarrow{\mathsf{WR}}$$
$$(\mathtt{var2} = \mathtt{acct2}) \xrightarrow{\mathsf{P}} (\mathtt{var1} = \mathtt{acct1}). \quad (12)$$

In fact, since the dependency graph in Figure 6 is not spliceable, the chopping defined by the above programs is incorrect. In Figure 14 we show the static chopping graph of the programs $\mathcal{P}^2 = \{\mathtt{transfer}, \mathtt{lookup1}, \mathtt{lookup2}\}$. This graph contains no critical cycles, and hence, the chopping defined by these programs is correct: they behave the same as when `transfer` is implemented by a single transaction.

5.3. Comparison with Transaction Chopping Under Other Consistency Models

We now compare our chopping criterion for SI to criteria that have been proposed for other consistency models: serializability [Shasha et al., 1995] and parallel SI [Cerone et al., 2015b]. For clarity, in the following we refer to critical cycles of Definition 5.1 as ***SI-critical***. The notion of chopping correctness in Definition 5.9 straightforwardly generalises to other consistency models.

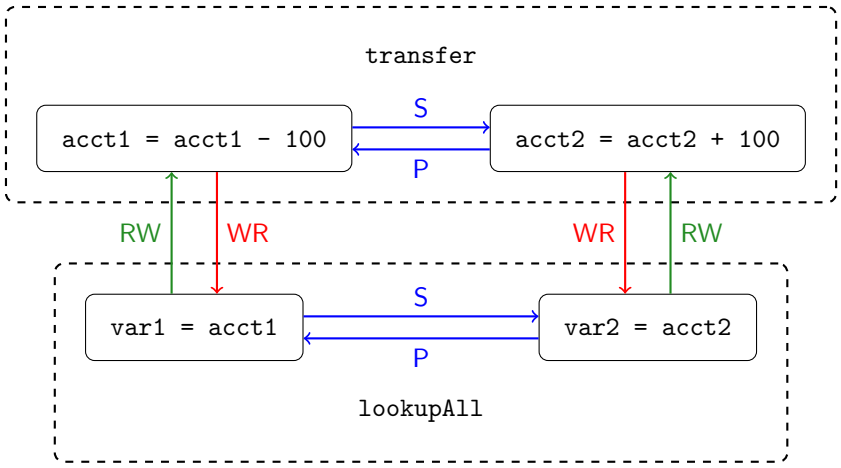


Fig. 13. The static chopping graph of the programs $\mathcal{P}^1 = \{\texttt{transfer}, \texttt{lookupAll}\}$ from Figure 6. Dashed boxes group program pieces into sessions.

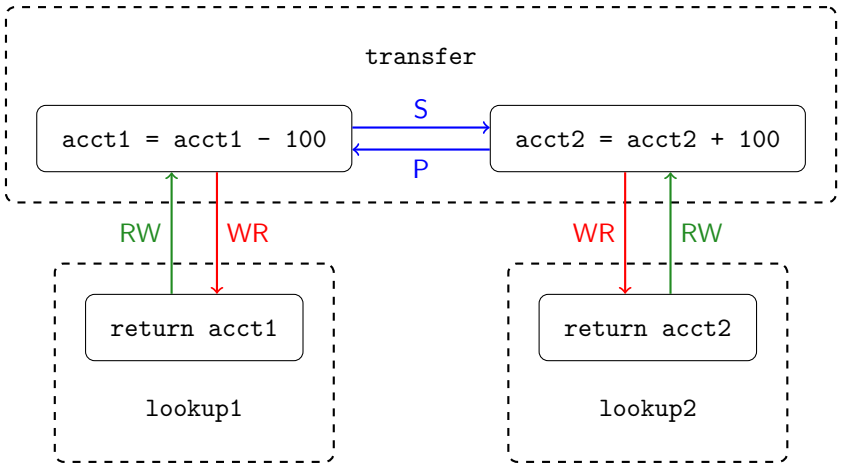


Fig. 14. The static chopping graph of the programs $\mathcal{P}^2 = \{\texttt{transfer}, \texttt{lookup1}, \texttt{lookup2}\}$ from Figure 6.

Transaction chopping under serializability. Following the approach in the proof of Theorem 5.2, we can easily establish the following improved version of the chopping criterion for serializability by Shasha et al. [Shasha et al., 1995].

Definition 5.12. A cycle in $\mathsf{SCG}(\mathcal{P})$ is ***SER-critical*** if:

(i) it is simple; and
(ii) it contains a fragment of three consecutive edges of the form "conflict, predecessor, conflict".

THEOREM 5.13. *The chopping defined by programs $\mathcal{P}$ is correct under serializability if $\mathsf{SCG}(\mathcal{P})$ contains no SER-critical cycles.*

For example, the chopping defined by the programs $\mathcal{P}^2$ considered in Figure 14 is correct under serializability. On the other hand, the chopping defined by $\mathcal{P}^1$ considered in Figure 13 is incorrect, and in fact $\mathsf{SCG}(\mathcal{P}^1)$ contains a SER-critical cycle (12).

Any SI-critical cycle is also SER-critical and, thus, a chopping that is correct under serializability is also correct under SI. It follows that the classical transaction chopping analysis for serializability of Shasha et al. is also sound for SI. Note that this result is non-trivial: the correctness of a chopping requires that the set of histories produced by the chopped program be included into the set of histories produced by the original program. Enlarging both sets when switching from serializability to SI may not preserve the inclusion.

The programs $\mathcal{P}^3 = \{\texttt{write1}, \texttt{write2}\}$ in Figure 15 define a correct chopping under SI, but not under serializability. Their static chopping graph $\mathsf{SCG}(\mathcal{P}^3)$, shown in the figure, has only one simple cycle with three consecutive edges of the form "conflict, predecessor, conflict":

$$(\texttt{var1} = \texttt{x}) \xrightarrow{\mathsf{S}} (\texttt{x} = \texttt{var2}) \xrightarrow{\mathsf{P}} (\texttt{var2} = \texttt{y}) \xrightarrow{\mathsf{RW}} (\texttt{y} = \texttt{var1}) \xrightarrow{\mathsf{P}} (\texttt{var1} = \texttt{x}).$$

This cycle is not SI-critical, and by Corollary 5.11, the chopping defined by $\mathcal{P}^3$ is correct under SI. However, it is incorrect under serializability. Indeed, consider the dependency graph $\mathcal{G}_7 \in \mathsf{GraphSER}$ in Figure 15, whose history can be produced by $\mathcal{P}^3$. Splicing the history $\mathcal{H}_{\mathcal{G}_7}$ results in a variant of the write skew anomaly, and thus $\mathsf{splice}(\mathcal{H}_{\mathcal{G}_7}) \notin \mathsf{HistSI}$.

Transaction chopping under parallel SI. We now compare our chopping criterion for SI to the one that we recently proposed [Cerone et al., 2015b] for parallel (aka non-monotonic) SI, a weakening of SI for large-scale databases [Sovran et al., 2011; Saeida Ardekani et al., 2013a]. To specify parallel SI in the framework of §2, we drop the axiom Prefix, while still requiring visibility to be transitive, a property that we refer to as TransVis [Cerone et al., 2015a].

Definition 5.14. The sets of executions and histories ***allowed by parallel SI*** are:

$$\mathsf{ExecPSI} = \{\mathcal{X} \mid \mathcal{X} \models \textsc{Int} \wedge \textsc{Ext} \wedge \textsc{Session} \wedge \textsc{TransVis} \wedge \textsc{NoConflict}\};$$
$$\mathsf{HistPSI} = \{\mathcal{H} \mid \exists \mathsf{VIS}, \mathsf{CO}.\, (\mathcal{H}, \mathsf{VIS}, \mathsf{CO}) \in \mathsf{ExecSI}\}.$$

Note that this specification essentially does not use the commit order CO: according to NoConflict, its edges used in Ext are uniquely determined by VIS.

The axiom TransVis ensures that transactions ordered by VIS are observed by others in this order; in particular, it disallows the anomaly in Figure 2(d). However, it allows two transactions unrelated by VIS to be observed in different orders; in particular, parallel SI allows the long fork anomaly of Figure 2(e), disallowed by the axiom Prefix in SI.

Definition 5.15. A cycle in $\mathsf{SCG}(\mathcal{P})$ is ***PSI-critical*** if:

(i) it is simple;
(ii) it contains a fragment of three consecutive edges of the form "conflict, predecessor, conflict"; and
(iii) it contains at most one anti-dependency edge.

Theorem 5.16 ([Cerone et al., 2015b]). *The chopping defined by programs $\mathcal{P}$ is correct under parallel SI if $\mathsf{SCG}(\mathcal{P})$ contains no PSI-critical cycles.*

Note that any cycle that is PSI-critical, is also SI-critical; as a consequence, the programs $\mathcal{P}^2$ from Figure 14 and $\mathcal{P}^3$ from Figure 15 considered above define a correct chopping under parallel SI. On the other hand, the programs $\mathcal{P}^1$ from Figure 13 are not chopped correctly under parallel SI, and the cycle (12) in $\mathsf{SCG}(\mathcal{P}^1)$ is PSI-critical.

The programs $\mathcal{P}^4 = \{\texttt{write1}, \texttt{write2}, \texttt{read1}, \texttt{read2}\}$ in Figure 16 define a correct chopping under parallel SI, but not SI. The static chopping graph $\mathsf{SCG}(\mathcal{P}^4)$, shown in the figure,

```
session write1 { tx { var1 = x } ; tx { y = var1 } }
session write2 { tx { var2 = y } ; tx { x = var2 } }
```

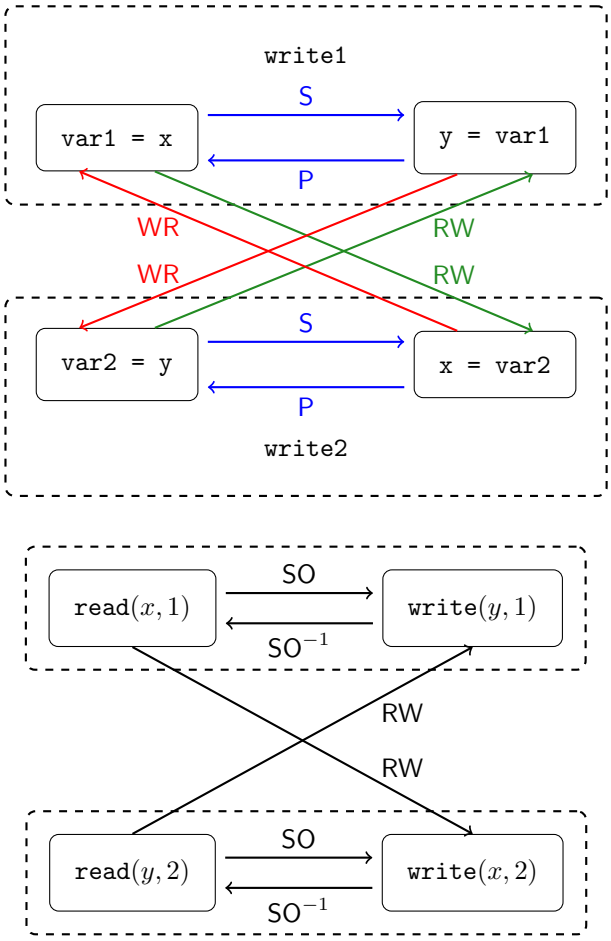


Fig. 15. Programs $\mathcal{P}^3$ defining a chopping correct under SI, but not under serializability.

contains exactly one simple cycle with three consecutive edges of the form "conflict, predecessor, conflict":

$$(\mathtt{x} = \mathtt{post1}) \xrightarrow{\mathsf{WR}} (\mathtt{b} = \mathtt{x}) \xrightarrow{\mathsf{P}} (\mathtt{a} = \mathtt{y}) \xrightarrow{\mathsf{RW}} (\mathtt{y} = \mathtt{post2}) \xrightarrow{\mathsf{WR}} (\mathtt{b} = \mathtt{y}) \xrightarrow{\mathsf{P}} (\mathtt{a} = \mathtt{x}) \xrightarrow{\mathsf{RW}} (\mathtt{x} = \mathtt{post1}).$$

This cycle is not PSI-critical, so that $\mathcal{P}^4$ indeed define a correct chopping under parallel SI. On the other hand, this cycle is SI-critical and $\mathcal{P}^4$ do not define a correct chopping under SI. Indeed, consider the dependency graph $\mathcal{G}_8 \in \mathsf{GraphSI}$ in Figure 16, whose history can be produced by $\mathcal{P}^4$. Splicing the history $\mathcal{H}_{\mathcal{G}_8}$ results in a long fork anomaly, and thus $\mathsf{splice}(\mathcal{H}_{\mathcal{G}_8}) \notin \mathsf{HistSI}$.

```
session write1 { tx { x = post1 } }
session write2 { tx { y = post2 } }
session read1 { tx { a = y }; tx { b = x }; return (a, b); }
session read2 { tx { a = x }; tx { b = y }; return (a, b); }
```

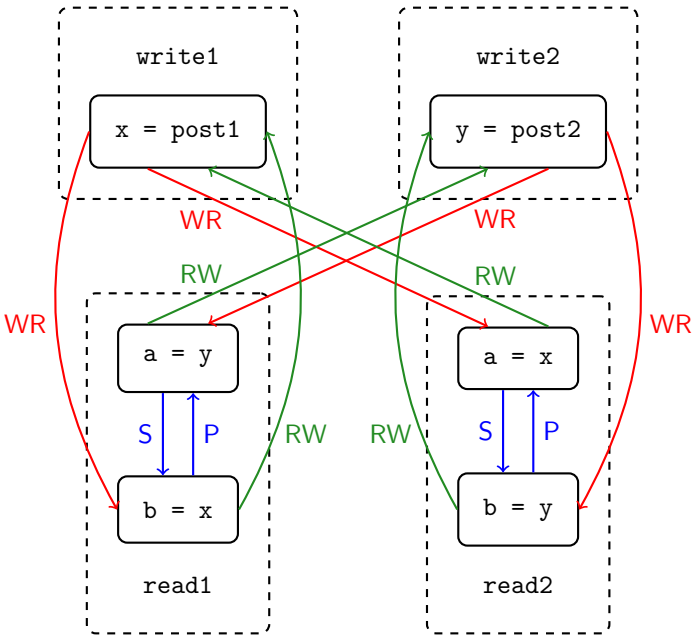


Fig. 16. Programs $\mathcal{P}^4$ defining a chopping correct under parallel SI, but not under SI.

6. ROBUSTNESS CRITERIA FOR SI

We now consider another type of a static analysis that checks whether an application is ***robust*** against weakening consistency: executing it under a weak consistency model produces the same client-observable behaviour as executing it under a stronger one.

6.1. Robustness against SI

We first show that our SI characterisation allows deriving a variant of an existing analysis that checks whether an application executing under SI behaves the same as when executing under serializability [Fekete et al., 2005] (robustness ***against SI***). For this the analysis checks that the application code may produce no histories in $\mathsf{HistSI} \setminus \mathsf{HistSER}$. Like for transaction chopping (§5), we first establish a ***dynamic robustness criterion*** that checks whether a single execution, represented by a dependency graph, is in $\mathsf{GraphSI} \setminus \mathsf{GraphSER}$. This easily follows from Theorems 3.4 and 4.2.

THEOREM 6.1. *For any $\mathcal{G}$, we have $\mathcal{G} \in \mathsf{GraphSI} \setminus \mathsf{GraphSER}$ if and only if $\mathcal{T}_\mathcal{G} \models \textsc{Int}$, $\mathcal{G}$ contains a cycle, and all its cycles have at least two adjacent anti-dependency edges.*

The dependency graph $\mathcal{G}_3$ of the write skew anomaly in Figure 2(f) contains a cycle: $T_1 \xrightarrow{\mathsf{RW}_{\mathcal{G}_3}} T_2 \xrightarrow{\mathsf{RW}_{\mathcal{G}_3}} T_1$. Furthermore, it is easy to see that all its cycles have two adjacent anti-dependencies, so that $\mathcal{G}_3 \in \mathsf{GraphSI} \setminus \mathsf{GraphSER}$. The dependency graph $\mathcal{G}_5$ from Figure 6 does not contain any cycles, so that $\mathcal{G}_5 \notin \mathsf{GraphSI} \setminus \mathsf{GraphSER}$; in fact, $\mathcal{G}_5 \in \mathsf{GraphSER}$. Finally, the dependency graph $\mathcal{G}_2$ of the long fork anomaly in Figure 2(e) contains a cycle, but without two adjacent anti-dependencies; hence, $\mathcal{G}_2 \notin \mathsf{GraphSI} \setminus \mathsf{GraphSER}$, and in fact, $\mathcal{G}_2 \notin \mathsf{GraphSI}$.

We can obtain the following refinement of Theorem 6.1 by using the characterisation of SI in Theorem 4.13 instead of that in Theorem 4.2.

THEOREM 6.2. *For any $\mathcal{G}$, we have $\mathcal{G} \in \mathsf{GraphSI} \setminus \mathsf{GraphSER}$ if and only if $\mathcal{T}_\mathcal{G} \models \textsc{Int}$, $\mathcal{G}$ contains a cycle, and all its cycles have at least two adjacent vulnerable anti-dependency edges over different objects.*

Fekete et al. previously established a result roughly corresponding to the "only if" direction of the above theorem [Fekete et al., 2005]. The "if" direction strengthens their result by showing that the criterion in the theorem is complete for checking whether a given dependency graph is admitted by SI, but not serializability.

We can derive a static analysis from Theorem 6.1 similarly to how it was done in §5. Namely, the analysis assumes that the code of transactions in an application is defined by a set of programs $\mathcal{P}$. It then constructs a ***static dependency graph***, over-approximating possible read and write dependencies and vulnerable anti-dependencies that may exist in executions of $\mathcal{P}$; in contrast with the chopping analysis, the robustness analysis also records the objects to which different edges refer to. The analysis then checks that the graph has no cycles with at least two adjacent vulnerable anti-dependency edges over different objects. By Theorem 6.2 this implies that the programs $\mathcal{P}$ produce no histories in $\mathsf{HistSI} \setminus \mathsf{HistSER}$, and hence, the corresponding application is robust against SI.

The construction of the static dependency graph above can be done like in the chopping analysis, using sets of objects that may be read or written by programs in $\mathcal{P}$. During this construction, we can exclude vulnerable anti-dependencies using information about objects that *must* necessarily be written by certain programs [Fekete et al., 2005] using the following proposition.

PROPOSITION 6.3. *Consider an anti-dependency edge $T \xrightarrow{\mathsf{RW}_\mathcal{G}(x)} S$ in a dependency graph $\mathcal{G} \in \mathsf{GraphSI}$. If $T \vdash \mathtt{write}(x, \_)$, then the anti-dependency is not vulnerable.*

PROOF. From $T \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} S$ we get $T \vdash \texttt{write}(x, \_)$ and $T \neq S$. Since we also know $T \vdash \texttt{write}(x, \_)$, we must have either $T \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S$ or $S \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T$. However, the latter yields a cycle $T \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} S \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T$, contradicting $\mathcal{G} \in \mathsf{GraphSI}$ by Theorem 4.1. Hence, we must have $T \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S$, implying the required. □

Note that the dependency graphs characterisation of consistency models greatly facilitates deriving the above robustness analysis, since the characterisations allow us to easily establish correspondences between executions on different models with the same histories.

6.2. Robustness against Parallel SI towards SI

We now use our SI characterisation to derive a static analysis that checks whether an application executing under parallel SI (§5.3) behaves the same as when executing under the classical SI (robustness ***against parallel SI towards SI***). First, we give a characterisation of parallel SI in terms of dependency graphs.

THEOREM 6.4 ([CERONE ET AL., 2015B, EXTENDED VERSION, LEMMA 14]). *Let*

$$\mathsf{GraphPSI} = \{\mathcal{G} \mid (\mathcal{T}_{\mathcal{G}} \models \textsc{Int}) \wedge (((\mathsf{SO}_{\mathcal{G}} \cup \mathsf{WR}_{\mathcal{G}} \cup \mathsf{WW}_{\mathcal{G}})^{+} \,;\, \mathsf{RW}_{\mathcal{G}}?) \textit{ is irreflexive})\}.$$

Then

$$\mathsf{HistPSI} = \{\mathcal{H} \mid \exists \mathsf{WR}, \mathsf{WW}, \mathsf{RW}.\, (\mathcal{H}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{GraphPSI}\}.$$

Thus, parallel SI is characterised by dependency graphs that contain only cycles with at least two anti-dependency edges. For example, consider the dependency graph $\mathcal{G}_2$ in Figure 2(e). It is easy to see that all its cycles contain at least two anti-dependencies, and therefore $\mathcal{G}_2 \in \mathsf{GraphPSI}$. On the other hand, let $\mathcal{G}_1$ be the dependency graph in Figure 2(b). The graph $\mathcal{G}_1$ contains a cycle with exactly one anti-dependency ($T_1 \xrightarrow{\mathsf{WW}_{\mathcal{G}_1}} T_2 \xrightarrow{\mathsf{RW}_{\mathcal{G}_1}} T_1$), and therefore $\mathcal{G}_1 \notin \mathsf{GraphPSI}$. As a corollary of Theorems 4.1 and 6.4, we obtain a dynamic robustness criterion that checks whether a given dependency graph is in $\mathsf{GraphPSI} \setminus \mathsf{GraphSI}$.

THEOREM 6.5. *For any $\mathcal{G}$, we have $\mathcal{G} \in \mathsf{GraphPSI} \setminus \mathsf{GraphSI}$ if and only if $\mathcal{T}_{\mathcal{G}} \models \textsc{Int}$, $\mathcal{G}$ contains at least one cycle with no adjacent anti-dependency edges, and all its cycles have at least two anti-dependency edges.*

For example, we have already noted that in the dependency graph $\mathcal{G}_2$ of the long fork anomaly (Figure 2(e)) all cycles have at least two anti-dependencies. Furthermore, $\mathcal{G}_2$ also has a cycle with no adjacent anti-dependencies:

$$T_1 \xrightarrow{\mathsf{WR}_{\mathcal{G}_2}} T_3 \xrightarrow{\mathsf{RW}_{\mathcal{G}_2}} T_2 \xrightarrow{\mathsf{WR}_{\mathcal{G}_2}} T_4 \xrightarrow{\mathsf{RW}_{\mathcal{G}_2}} T_1,$$

so that $\mathcal{G}_2 \in \mathsf{GraphPSI} \setminus \mathsf{GraphSI}$. The dependency graph $\mathcal{G}_3$ of the write skew anomaly in Figure 2(f) contains only cycles with at least two adjacent anti-dependencies, so that $\mathcal{G}_3 \notin \mathsf{GraphPSI} \setminus \mathsf{GraphSI}$; in fact, $\mathcal{G}_3 \in \mathsf{GraphSI}$. The dependency graph $\mathcal{G}_1$ of the lost update anomaly (Figure 2(b)) contains a cycle with exactly one anti-dependency, so that $\mathcal{G}_1 \notin \mathsf{GraphPSI} \setminus \mathsf{GraphSI}$; in fact, $\mathcal{G}_1 \notin \mathsf{GraphPSI}$.

We can refine the characterisation in Theorem 6.4 by taking into account the objects involved in dependencies between transactions, similarly to how we did this for SI (Theorem 4.13, §4.5); for brevity, we omit a refinement that takes into account vulnerability.

THEOREM 6.6. *Let $\mathcal{G}$ be a dependency graph. Then $\mathcal{G} \in \mathsf{GraphPSI}$ if and only if $\mathcal{T}_{\mathcal{G}} \models \textsc{Int}$ and all cycles in $\mathcal{G}$ contain at least two anti-dependency edges over different objects.*

PROOF. The "if" part of the theorem is trivial. For the "only if" part, suppose that $\mathcal{G} \in \mathsf{GraphPSI}$. We can show that all cycles in $\mathcal{G}$ contain at least two anti-dependency edges

over different objects using the following algebraic law:

$$\forall x.\, \mathsf{RW}_{\mathcal{G}}(x)\ ;\ (\mathsf{SO}_{\mathcal{G}} \cup \mathsf{WR}_{\mathcal{G}} \cup \mathsf{WW}_{\mathcal{G}})^*\ ;\ \mathsf{RW}_{\mathcal{G}}(x) \subseteq \mathsf{RW}_{\mathcal{G}}(x)\ ;\ \mathsf{WW}_{\mathcal{G}}(x). \tag{13}$$

Indeed, if $\mathcal{G}$ contains a cycle where all anti-dependency edges are over the same object x, then (13) allows us to convert this cycle into one with at most one anti-dependency edge, which contradicts $\mathcal{G} \in \mathsf{GraphPSI}$ by Theorem 6.4.

It remains to prove (13). To this end, consider $x \in \mathsf{Obj}$ and $T, T', S', S \in \mathcal{T}_{\mathcal{G}}$ such that

$$T \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} T' \xrightarrow{(\mathsf{SO}_{\mathcal{G}} \cup \mathsf{WR}_{\mathcal{G}} \cup \mathsf{WW}_{\mathcal{G}})^*} S \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} S'.$$

Then $T' \vdash \mathtt{write}(x, \_)$ and $S' \vdash \mathtt{write}(x, \_)$. Therefore, we have one of the following: $T' = S'$, $S' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T'$ or $T' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S'$. We cannot have that $T' = S'$: if this were the case, we would have the cycle

$$T' \xrightarrow{(\mathsf{SO}_{\mathcal{G}} \cup \mathsf{WR}_{\mathcal{G}} \cup \mathsf{WW}_{\mathcal{G}})^*} S \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} T'$$

in $\mathcal{G}$ with a single anti-dependency edge, contradicting $G \in \mathsf{GraphPSI}$. We cannot have $S' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} T'$: in this case we would have the cycle

$$S' \xrightarrow{(\mathsf{SO}_{\mathcal{G}} \cup \mathsf{WR}_{\mathcal{G}} \cup \mathsf{WW}_{\mathcal{G}})^+} S \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} S',$$

again disallowed because $\mathcal{G} \in \mathsf{GraphPSI}$. We are thus left with the case $T' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S$, so that $T \xrightarrow{\mathsf{RW}_{\mathcal{G}}(x)} T' \xrightarrow{\mathsf{WW}_{\mathcal{G}}(x)} S$. This establishes (13), as required. □

As an illustration of Theorem 6.6, the dependency graph $\mathcal{G}_2$ of the long fork anomaly in Figure 2(e), allowed by parallel SI, contains only cycles with at least two anti-dependencies over different objects: e.g.,

$$T_1 \xrightarrow{\mathsf{WR}_{\mathcal{G}_2}(x)} T_3 \xrightarrow{\mathsf{RW}_{\mathcal{G}_2}(y)} T_2 \xrightarrow{\mathsf{WR}_{\mathcal{G}_2}(y)} T_4 \xrightarrow{\mathsf{RW}_{\mathcal{G}_2}(x)} T_1.$$

We can obtain the following refinement of Theorem 6.5 by using the characterisations of SI and PSI in Theorems 4.13 and 6.6.

THEOREM 6.7. *For any $\mathcal{G}$, we have $\mathcal{G} \in \mathsf{GraphPSI} \setminus \mathsf{GraphSI}$ if and only if $\mathcal{T}_{\mathcal{G}} \models \textsc{Int}$, $\mathcal{G}$ contains at least one cycle with no adjacent anti-dependency edges over different objects, and all its cycles have at least two anti-dependency edges over different objects.*

From Theorem 6.7 it follows that a static analysis for robustness against parallel SI towards SI can check that the static dependency graph of an application contains no cycles where there are at least two anti-dependency edges over different objects and no two anti-dependency edges over different objects are adjacent.

7. RELATED WORK

Snapshot isolation was originally defined by an idealised algorithm formulated in terms of implementation-level concepts [Berenson et al., 1995]. Since then there have been proposals of more declarative SI specifications [Adya, 1999; Saeida Ardekani et al., 2013b; Cerone et al., 2015a], one of which [Cerone et al., 2015a] was our starting point (§2). However, these specifications are stated in terms of relations which make it challenging to obtain results such as transaction chopping and robustness analyses.

Fekete et al. [Fekete et al., 2005] proposed the analysis for robustness against SI that we considered in §6.1. To this end, they have proved a fact roughly equivalent to our completeness result (Theorem 4.2(ii)), but they did not establish an analogue of our soundness result (Theorem 4.2(i)). The latter more challenging result is the one that is needed to obtain analyses for transaction chopping under SI and for robustness against parallel SI

towards SI: both require proving that an execution with a particular dependency graph is in SI, rather than the other way round. We also hope that our specification of SI will be beneficial in other domains where dependency graphs have been useful, such as run-time monitoring [Cahill et al., 2009; Zellag and Kemme, 2014] and proving the correctness of concurrency-control algorithms [Xie et al., 2015; Diegues and Romano, 2014]. Finally, we expect that the approach to constructing a total commit order from transactional dependencies in the proof of our soundness theorem can be used to give dependency graph characterisations to other consistency models whose formulation includes similar total orders, such as *prefix consistency* [Terry et al., 2013].

The constraint on dependency graphs that we use to characterise SI also arose in the work of Lin et al. [Lin et al., 2009], who used it to formulate conditions under which a replicated database guarantees SI provided every one of its replicas does so. In comparison to them, we solve a more general problem of characterising SI regardless of how it is implemented and handle a variant of SI that does not require transactions to see the latest snapshot.

Transaction chopping has recently received a lot of attention. In particular, researchers have demonstrated that transactions arising in web applications can be chopped in a way that drastically improves their performance when executed under serializability [Zhang et al., 2013; Mu et al., 2014; Xie et al., 2015]. There have also been proposals of consistency models for transactional memory that weaken consistency guarantees in a way similar to chopping [Felber et al., 2009; Xiang and Scott, 2015; Afek et al., 2011]. Our chopping analysis enables bringing these benefits to transactional systems providing SI. We have previously proposed a chopping analysis for parallel SI [Cerone et al., 2015b], which also relies on a dependency graph characterisation of this consistency model (Theorem 6.4, §6.2). But since parallel SI can be formulated without using an analogue of SI's commit order, its dependency graph characterisation did not present the challenges that we had to deal with when establishing our soundness theorem.

Acknowledgements

We thank Hongseok Yang, who participated in the early stages of this work. We also thank Hagit Attiya, Ricardo Jiménez-Peris and Pierre Sutra for comments that helped improve the paper. This work was supported by an EU project ADVENT. The first author was also supported by the EPSRC Programme Grant REMS: Rigorous Engineering for Mainstream Systems (EP/K008528/1).

REFERENCES

Adya, A. (1999). Weak consistency: A generalized theory and optimistic implementations for distributed transactions. PhD thesis, MIT.

Adya, A., Liskov, B., and O'Neil, P. E. (2000). Generalized isolation level definitions. In *ICDE*.

Afek, Y., Avni, H., and Shavit, N. (2011). Towards consistency oblivious programming. In *OPODIS*.

Bailis, P., Fekete, A., Ghodsi, A., Hellerstein, J. M., and Stoica, I. (2014). Scalable atomic visibility with RAMP transactions. In *SIGMOD*.

Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., and O'Neil, P. (1995). A critique of ANSI SQL isolation levels. In *SIGMOD*.

Bernstein, P. A., Hadzilacos, V., and Goodman, N. (1987). *Concurrency Control and Recovery in Database Systems*. Addison-Wesley.

Bieniusa, A. and Fuhrmann, T. (2010). Consistency in hindsight: A fully decentralized STM algorithm. In *IPDPS*.

Cahill, M. J., Röhm, U., and Fekete, A. D. (2009). Serializable isolation for snapshot databases. *ACM Trans. Database Syst.*, 34(4).

Cerone, A., Bernardi, G., and Gotsman, A. (2015a). A framework for transactional consistency models with atomic visibility. In *CONCUR*. Dagstuhl.

Cerone, A., Gotsman, A., and Yang, H. (2015b). Transaction chopping for parallel snapshot isolation. In *DISC*. Extended version available from www.software.imdea.org/~gotsman.

Closure (2016). Refs and transactions. http://clojure.org/refs.

Daudjee, K. and Salem, K. (2004). Lazy database replication with ordering guarantees. In *ICDE*.

Daudjee, K. and Salem, K. (2006). Lazy database replication with snapshot isolation. In *VLDB*.

Dias, R. J., Lourenço, J. M., and Preguiça, N. (2011). Efficient and correct transactional memory programs combining snapshot isolation and static analysis. In *HotPar*.

Diegues, N. and Romano, P. (2014). Time-warp: Lightweight abort minimization in transactional memory. In *PPoPP*.

Doherty, S., Groves, L., Luchangco, V., and Moir, M. (2013). Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 25(5).

Elnikety, S., Zwaenepoel, W., and Pedone, F. (2005). Database replication using generalized snapshot isolation. In *SRDS*.

Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., and Shasha, D. (2005). Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2).

Felber, P., Gramoli, V., and Guerraoui, R. (2009). Elastic transactions. In *DISC*.

Guerraoui, R. and Kapalka, M. (2008). On the correctness of transactional memory. In *PPoPP*.

Herlihy, M. and Moss, J. E. B. (1993). Transactional memory: Architectural support for lock-free data structures. In *ISCA*.

Jorwekar, S., Fekete, A., Ramamritham, K., and Sudarshan, S. (2007). Automating the detection of snapshot isolation anomalies. In *VLDB*.

Lin, Y., Kemme, B., Jiménez-Peris, R., Patiño-Martínez, M., and Armendáriz-Iñigo, J. E. (2009). Snapshot isolation and integrity constraints in replicated databases. *ACM Trans. Database Syst.*, 34(2).

Litz, H., Cheriton, D., Firoozshahian, A., Azizi, O., and Stevenson, J. P. (2014). SI-TM: Reducing transactional memory abort rates through snapshot isolation. In *ASPLOS*.

Mu, S., Cui, Y., Zhang, Y., Lloyd, W., and Li, J. (2014). Extracting more concurrency from distributed transactions. In *OSDI*.

Peng, D. and Dabek, F. (2010). Large-scale incremental processing using distributed transactions and notifications. In *OSDI*.

Riegel, T., Fetzer, C., and Felber, P. (2006). Snapshot isolation for software transactional memory. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*.

Saeida Ardekani, M., Sutra, P., and Shapiro, M. (2013a). Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *SRDS*.

Saeida Ardekani, M., Sutra, P., Shapiro, M., and Preguiça, N. (2013b). On the scalability of snapshot isolation. In *Euro-Par*.

Serrano, D., Patiño-Martínez, M., Jiménez-Peris, R., and Kemme, B. (2007). Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *PRDC*.

Shasha, D., Llirbat, F., Simon, E., and Valduriez, P. (1995). Transaction chopping: Algorithms and performance studies. *ACM Trans. Database Syst.*, 20(3).

Shasha, D. and Snir, M. (1988). Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2).

Sovran, Y., Power, R., Aguilera, M. K., and Li, J. (2011). Transactional storage for geo-replicated systems. In *SOSP*.

Terry, D. B., Demers, A. J., Petersen, K., Spreitzer, M., Theimer, M., and Welch, B. W. (1994). Session guarantees for weakly consistent replicated data. In *PDIS*.

Terry, D. B., Prabhakaran, V., Kotla, R., Balakrishnan, M., Aguilera, M. K., and Abu-Libdeh, H. (2013). Consistency-based service level agreements for cloud storage. In *SOSP*.

Xiang, L. and Scott, M. L. (2015). Software partitioning of hardware transactions. In *PPoPP*.

Xie, C., Su, C., Littley, C., Alvisi, L., Kapritsos, M., and Wang, Y. (2015). High-performance ACID via modular concurrency control. In *SOSP*.

Zellag, K. and Kemme, B. (2014). Consistency anomalies in multi-tier architectures: Automatic detection and prevention. *The VLDB Journal*, 23(1).

Zhang, Y., Power, R., Zhou, S., Sovran, Y., Aguilera, M., and Li, J. (2013). Transaction chains: Achieving serializability with low latency in geo-distributed storage systems. In *SOSP*.