Analysing Snapshot Isolation

Andrea Cerone
IMDEA Software Institute

Alexey Gotsman
IMDEA Software Institute

ABSTRACT

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 it running under SI.

Keywords

Snapshot isolation; transaction chopping; robustness

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 [7] and, more recently, by transactional memory systems [22]. Ideally, programmers would like to get strong guarantees about the isolation of transactional computations, formalised by the notion of *serializability* [7]: 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) [6] is one of the most popular such models, implemented by major centralised databases (e.g., MS SQL Sever, Oracle), distributed databases [14, 27, 29] and transactional memory systems [1, 8, 15, 25].

PODC'16, July 25 - 28, 2016, Chicago, IL, USA
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-3964-3/16/07...$15.00
DOI: http://dx.doi.org/10.1145/2933057.2933096

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(d). 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 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. [2, 3]. 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 [7]. 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 [12, 19, 23, 30, 38], run-time monitoring [9, 37] and proving concurrency-control algorithms correct [16, 24, 36]. 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 [2], as well as more recent proposals of consistency models [5, 36]. 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 [2]. 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* [4, 30, 35]—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 [30] and parallel SI [12], a recently-proposed weaker version of SI for large-scale databases [32]. 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* [14, 33] (aka *chains* [38]) 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* [19, 31] 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 [19] (robustness *against SI*, §6.1). We then propose a new static analysis that checks whether an application executing under the recently-proposed parallel SI [32] 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.

Due to space constraints, we defer some of the proofs to [11, §A].

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 [10]. 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 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 [2, 17, 21]. We denote components of transactions and similar structures as in E_T and po_T.

To allow transaction chopping (§5), we assume that the transactional system allows its clients to group several transactions into a session [33], which establishes an ordering on the transactions. Thus, instead of classical SI and serializability, we actually define their ***strong session*** variants [13, 14]. We represent the client-visible results of an execution of a set of sessions by a *history*.

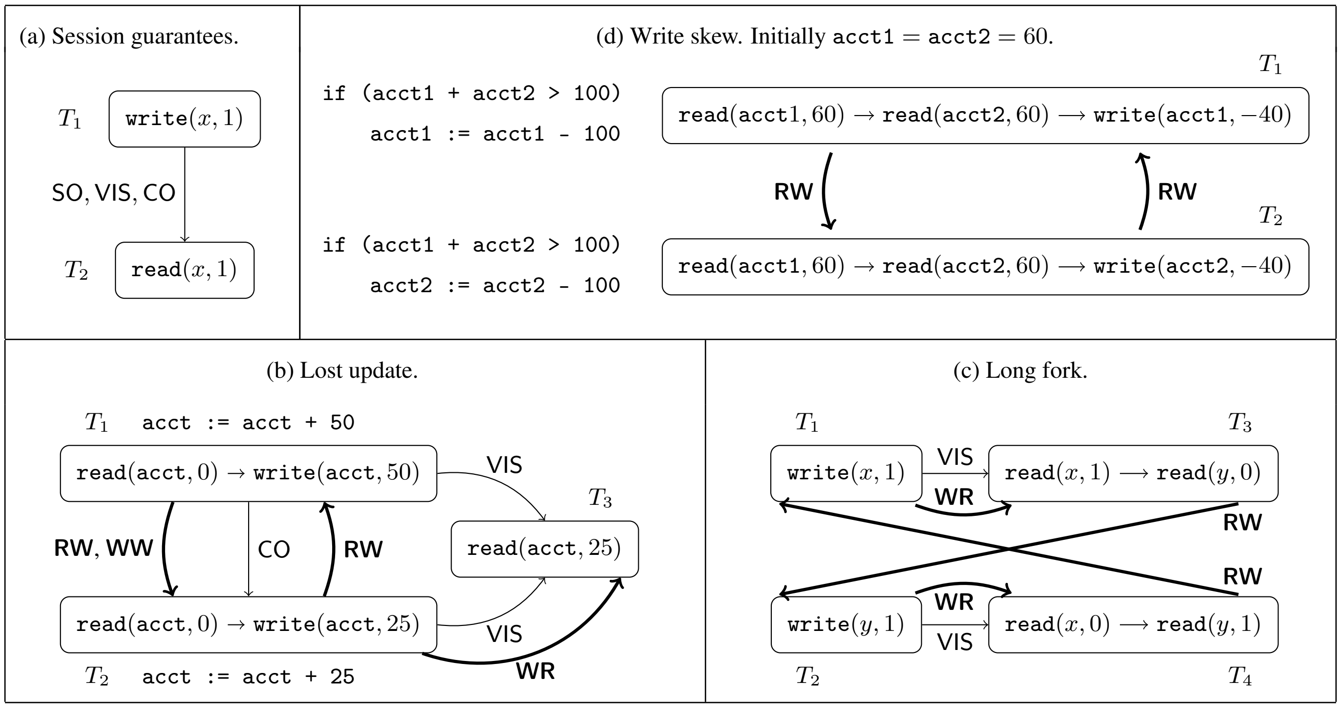
DEFINITION 2. *A **history** is a pair $\mathcal{H} = (\mathcal{T}, \mathsf{SO})$, where $\mathcal{T}$ is a finite set of transactions with disjoint sets of events and the **session order** $\mathsf{SO} \subseteq \mathcal{T} \times \mathcal{T}$ is a union of total orders defined on disjoint subsets of $\mathcal{T}$, 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 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.*

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

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



(a) Session guarantees.
T_1
$\mathtt{write}(x,1)$
SO, VIS, CO
T_2
$\mathtt{read}(x,1)$
(d) Write skew. Initially $\mathtt{acct1} = \mathtt{acct2} = 60$.
if (acct1 + acct2 > 100)
acct1 := acct1 - 100
if (acct1 + acct2 > 100)
acct2 := acct2 - 100
T_1
$\mathtt{read}(\mathtt{acct1},60) \to \mathtt{read}(\mathtt{acct2},60) \longrightarrow \mathtt{write}(\mathtt{acct1},-40)$
RW
RW
T_2
$\mathtt{read}(\mathtt{acct1},60) \to \mathtt{read}(\mathtt{acct2},60) \longrightarrow \mathtt{write}(\mathtt{acct2},-40)$
(b) Lost update.
T_1 acct := acct + 50
$\mathtt{read}(\mathtt{acct},0) \to \mathtt{write}(\mathtt{acct},50)$
VIS
T_3
$\mathtt{read}(\mathtt{acct},25)$
RW, WW
CO
RW
$\mathtt{read}(\mathtt{acct},0) \to \mathtt{write}(\mathtt{acct},25)$
VIS
WR
T_2 acct := acct + 25
(c) Long fork.
T_1
T_3
$\mathtt{write}(x,1)$
VIS
WR
$\mathtt{read}(x,1) \longrightarrow \mathtt{read}(y,0)$
RW
RW
WR
$\mathtt{write}(y,1)$
VIS
$\mathtt{read}(x,0) \longrightarrow \mathtt{read}(y,1)$
T_2
T_4

Figure 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.

We write $T \xrightarrow{\mathsf{VIS}} S$ and $(T, S) \in \mathsf{VIS}$ interchangeably, and similarly for other relations. 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 4. *The sets of executions and histories* ***allowed by (strong session) SI and serializability*** *are:*

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

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 $R \subseteq A \times A$, we let $\max_R(A)$ be the element $a \in A$ such that $\forall b \in A.\, a = b \vee (b, a) \in R$; if $A = \emptyset$, then $\max_R(A)$ is undefined. In the following, the use of $\max_R(A)$ in an expression implicitly assumes that it is defined. We define $\min_R(A)$ similarly. For a relation $R \subseteq A \times A$ and an element $a \in A$, we let $R^{-1}(a) = \{b \mid (b, a) \in R\}$. We define the sequential composition of relations R_1 and R_2 as

$$R_1 \,;\, R_2 = \{(a, b) \mid \exists c.\, (a, c) \in R_1 \wedge (c, b) \in R_2\}.$$

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. 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.

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*** [18]. However, following strong session SI [13, 14], 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 4 imply that VIS is transitive. PREFIX disallows the ***long fork*** anomaly shown in Figure 2(c), which is allowed by some weakening of SI (such as parallel SI [32]). 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(c), 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 4) defined using the consistency axioms explained so far is exactly the one produced by the SI concurrency-control algorithm [10]. The axioms allow the execution in Figure 2(d) 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. [2, 3].

DEFINITION 5. *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(d) and 2(c) 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 4. To this end, we extend histories to *dependency graphs* (aka direct serialization graphs) [2], which include relations representing the dependencies.

DEFINITION 6. *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*

- *$\mathsf{WR} : \mathsf{Obj} \to 2^{\mathcal{T} \times \mathcal{T}}$ is such that:*
  - *$\forall T, S \in \mathcal{T}.\, \forall x.\, T \xrightarrow{\mathsf{WR}(x)} S \implies \exists n.\, T \neq S \wedge T \vdash \mathtt{write}(x, n) \wedge S \vdash \mathtt{read}(x, n)$;*
  - *$\forall S \in \mathcal{T}.\, \forall x.\, S \vdash \mathtt{read}(x, \_) \implies \exists T.\, T \xrightarrow{\mathsf{WR}(x)} S$;*
  - *$\forall T, T', S \in \mathcal{T}.\, \forall x.\, (T \xrightarrow{\mathsf{WR}(x)} S \wedge T' \xrightarrow{\mathsf{WR}(x)} S) \implies T = T'$.*
- *$\mathsf{WW} : \mathsf{Obj} \to 2^{\mathcal{T} \times \mathcal{T}}$ is such that for every $x \in \mathsf{Obj}$, $\mathsf{WW}(x)$ is a total order on the set $\mathsf{WriteTx}_x$;*
- *$\mathsf{RW} : \mathsf{Obj} \to 2^{\mathcal{T} \times \mathcal{T}}$ is derived from WR and WW as in Definition 5.*

PROPOSITION 7. *For any $\mathcal{X} \in \mathsf{ExecSI}$, $\mathsf{graph}(\mathcal{X}) = (\mathcal{T}_\mathcal{X}, \mathsf{SO}_\mathcal{X}, \mathsf{WR}_\mathcal{X}, \mathsf{WW}_\mathcal{X}, \mathsf{RW}_\mathcal{X})$ is a dependency graph.*

Note that the constraints on WR in Definition 6 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 $\mathcal{T} \models \text{INT}$ if a set of transactions $\mathcal{T}$ satisfies the internal consistency axiom INT in Figure 1. (Strong session) serializability can be characterised by the set of acyclic dependency graphs with internally consistent transactions [2].

THEOREM 8. *Let*

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

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(d) and 2(c) are not serializable, and they cannot be extended to acyclic dependency graphs; in particular, the graphs shown in the figures with bold edges satisfy the conditions of Definition 6, 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 $R \subseteq \mathcal{T} \times \mathcal{T}$ let $R? = R \cup \{(T, T) \mid T \in \mathcal{T}\}$. We show that (strong session) SI is characterised by dependency graphs that contain only cycles with at least two adjacent anti-dependency edges.

THEOREM 9. *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(d) is allowed by SI, and indeed the dependency graph 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(c) 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 shown in the figures contains 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(c).

To prove Theorem 9, 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 10.
(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 [19]. 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). We present this proof first.

The main challenge 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 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 11. *A tuple* $\mathcal{P} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO})$ *is a* ***pre-execution*** *if it satisfies all the conditions of Definition 3, except* CO *is a strict partial order that may not be total. We let* PreExecSI *be the set of pre-executions satisfying the SI axioms (Figure 1):*

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

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

Figure 3: Requirements on a pre-execution $\mathcal{P} = (\mathcal{H}, \mathsf{VIS}, \mathsf{CO})$ constructed from a dependency graph $\mathcal{G} = (\mathcal{H}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$.

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; for $\mathcal{P} \in \mathsf{PreExecSI}$, $\mathsf{graph}(\mathcal{P})$ is indeed a dependency graph.

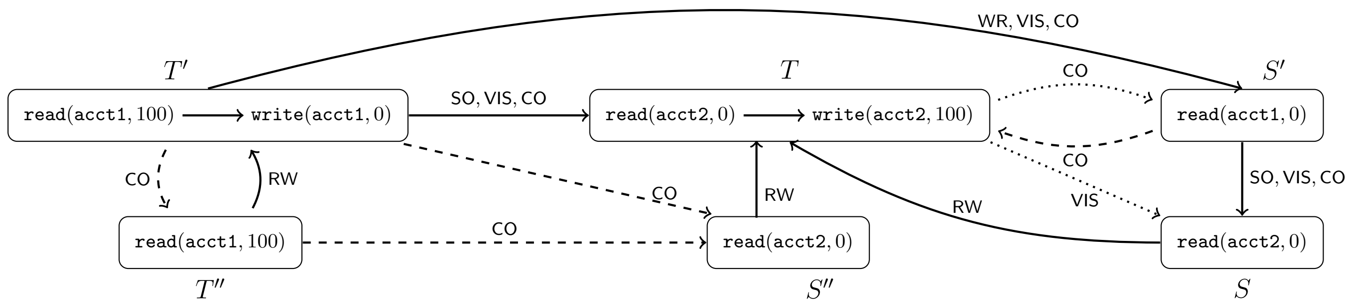
We first obtain auxiliary results that, given a dependency graph $\mathcal{G} = (\mathcal{T}, \mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{GraphSI}$, allow us to construct a pre-execution $\mathcal{P} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO}) \in \mathsf{PreExecSI}$ such that $\mathsf{graph}(\mathcal{P}) = \mathcal{G}$. Later 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 3. First, to ensure $\mathsf{graph}(\mathcal{P}) = \mathcal{G}$, by Definition 5 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 5, 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 11 from Definition 3. Inequality (S4) requires CO to be transitive; (S2) and (S3) ensure that so is VIS.

As we now explain, (S5) ensures the axiom EXT. Consider the dependency graph $\mathcal{G} \in \mathsf{GraphSI}$ in Figure 4, which we use as our running example in this section and 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. The VIS and CO relations shown by the solid arrows give a pre-execution satisfying inequalities (S1)-(S4) and, in fact, all the SI axioms. Suppose we want to construct a pre-execution with a bigger CO by adding an edge $T \xrightarrow{\mathsf{CO}} S'$ (shown dotted). Since $S' \xrightarrow{\mathsf{VIS}} S$, for the resulting pre-execution to satisfy PREFIX we also need to add an edge $T \xrightarrow{\mathsf{VIS}} S$. But then the pre-execution violates EXT: S sees the write to `acct2` by T, but reads the value from the initialisation transaction (elided from Figure 4) that writes 0 to `acct2` and precedes T in CO and WW; the latter fact is witnessed by the edge $S \xrightarrow{\mathsf{RW}} T$. On the other hand, adding an edge $S' \xrightarrow{\mathsf{CO}} T$ (shown dashed), which belongs to VIS ; RW, does not violate EXT. This example illustrates a general pattern. First, as the following lemma shows, (S5) must hold in any SI execution.

LEMMA 12. $\forall \mathcal{X} \in \mathsf{ExecSI}.\ \mathsf{VIS}_\mathcal{X} \ ; \mathsf{RW}_\mathcal{X} \subseteq \mathsf{CO}_\mathcal{X}$.

Conversely, the system of inequalities in Figure 3 can be used to ensure that a pre-execution $\mathcal{P} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO})$ satisfies EXT and has other desired properties.

LEMMA 13. *Let* $\mathcal{G} = (\mathcal{T}, \mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW})$ *be a dependency graph such that* $\mathcal{T} \models \textsc{Int}$ *and* $\mathsf{VIS}, \mathsf{CO} \subseteq \mathcal{T} \times \mathcal{T}$ *be acyclic relations satisfying the system of inequalities in Figure*



```
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 }
```

Figure 4: An illustration of constructing an execution from a dependency graph (§4) and splicing an execution (§5). We omit an initialisation transaction that sets $\texttt{acct1} = 100$ **and** $\texttt{acct2} = 0$**.**

3. Then $\mathcal{P} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO})$ *is a pre-execution such that* $\mathcal{P} \in \mathsf{PreExecSI}$ *and* $\mathsf{graph}(\mathcal{P}) = \mathcal{G}$.

The proof of Lemma 12 depends on the following characterisation of anti-dependencies in terms of visibility edges.

PROPOSITION 14.

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

Informally, if we had $S \xrightarrow{\mathsf{RW}_{\mathcal{X}}} T$ and $T \xrightarrow{\mathsf{VIS}_{\mathcal{X}}} S$, then S would have to read a value of x at least as up-to-date as that written by T, contradicting the definition of $\mathsf{RW}_{\mathcal{X}}$.

PROOF OF LEMMA 12. 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 14. If $T \xrightarrow{\mathsf{CO}_{\mathcal{X}}} S'$ (see Figure 4), then by PREFIX we get $T \xrightarrow{\mathsf{VIS}_{\mathcal{X}}} S$, contradicting Proposition 14. Then, since $\mathsf{CO}_{\mathcal{X}}$ is total, we must have $S' \xrightarrow{\mathsf{CO}_{\mathcal{X}}} T$. □

PROOF OF LEMMA 13. 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 $T' \in \mathsf{VIS}^{-1}(S) \cap \mathsf{WriteTx}_x$. We now show that $T = T'$, which entails the required.

Assume the contrary: $T \neq T'$. We have $T, T' \in \mathsf{VIS}^{-1}(S) \cap \mathsf{WriteTx}_x$. Hence, T and T' are related by WW. Since $\mathsf{WW} \subseteq \mathsf{CO}$, they are related in the same way by the acyclic $\mathsf{CO}(x)$. Then by the definition of T we must have $T' \xrightarrow{\mathsf{CO}(x)} T$ and $T' \xrightarrow{\mathsf{WW}} T$. From the latter and $T' \xrightarrow{\mathsf{WR}(x)} S$ we get $S \xrightarrow{\mathsf{RW}} T$. But $T \xrightarrow{\mathsf{VIS}} S$, so from (S5) we get $T \xrightarrow{\mathsf{CO}} T$. This contradicts the assumption that CO is acyclic. Hence, we must have $T = T'$. □

According to Lemma 13, 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 3 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 $^+$ and * to denote the transitive closure and the transitive and reflexive closure of a given relation.

LEMMA 15. *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*

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

are a solution to the system of inequalities in Figure 3. 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* $\mathsf{CO} \subseteq \mathsf{CO}'$.

In particular, for $R = \emptyset$, Lemma 15 gives the smallest solution $(\mathsf{VIS}_0, \mathsf{CO}_0)$ to the system of inequalities in Figure 3.

If $\mathcal{G} \in \mathsf{GraphSI}$, then CO_0 is acyclic, and by (S3), so is VIS_0 (in fact, Lemma 15 is our motivation for defining $\mathsf{GraphSI}$ the way we did). Hence, by Lemma 13, $\mathcal{P}_0 = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}_0, \mathsf{CO}_0)$ is a pre-execution such that $\mathcal{P}_0 \in \mathsf{PreExecSI}$ and $\mathsf{graph}(\mathcal{P}_0) = \mathcal{G}$, which is what we originally set out to construct. We now proceed to prove Theorem 10(i) by extending the pre-execution $\mathcal{P}_0$ to an execution $\mathcal{X} \in \mathsf{ExecSI}$.

PROOF OF THEOREM 10(I). Assume $\mathcal{G} = (\mathcal{T}, \mathsf{SO}, \mathsf{WR}, \mathsf{WW}, \mathsf{RW}) \in \mathsf{GraphSI}$. To construct $\mathcal{X}$, we define a sequence of pre-executions $\{\mathcal{P}_i = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}_i, \mathsf{CO}_i)\}_{i=0}^n$ for some $n \geq 0$ and let $\mathcal{X} = \mathcal{P}_n$. The sequence is such that $\mathsf{VIS}_i \subseteq \mathsf{VIS}_{i+1}$ 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. That is, on every step of our construction we add edges to the commit order until it becomes total. Each pair $(\mathsf{VIS}_i, \mathsf{CO}_i)$ gives an acyclic solution to the system in Figure 3. Then by Lemma 13, $\mathcal{P}_i \in \mathsf{PreExecSI}$ and $\mathsf{graph}(\mathcal{P}_i) = \mathcal{G}$. In particular, $\mathcal{P}_n \in \mathsf{ExecSI}$ and $\mathsf{graph}(\mathcal{P}_n) = \mathcal{G}$, as required.

We start the construction of the sequence by taking as $\mathcal{P}_0$ the pre-execution that we constructed above. For example, for the dependency graph in Figure 4, VIS_0 and CO_0 consist of the solid edges in the figure and the dashed edge $S' \xrightarrow{\mathsf{CO}} T$. If the relation CO_0 is not total, then we 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 3 such that $\mathsf{CO}_1 \supseteq \{(T_1, S_1)\}$. By Lemma 15 this solution is given by (1) for $R = \{(T_1, S_1)\}$. For example, in Figure 3 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 4, then we get $\mathsf{VIS}_1 = \mathsf{VIS}_0$, and $\mathsf{CO}_1 = \mathsf{CO}_0 \cup \{(T', T'')\}$. In general, the construction continues in the same way: while CO_i is not total, we pick an arbitrary pair of transactions (T_i, S_i) unrelated by CO_i and force CO_{i+1} to include it. In our example, CO_1 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 $R = \{(T', T''), (T'', S'')\}$ in (1); this corresponds to all the solid and dashed edges in Figure 4. Note that CO_2 also includes the edge (T', S''). Since CO_2 is total in this example, the construction terminates: $\mathcal{X} = \mathcal{P}_2$.

Formally, in addition to VIS_i and CO_i we construct sets $R_i = \{(T_k, S_k) \mid k = 1..i\}$, $i = 0..n$ that accumulate the edges enforced in the commit order at every step. The relations VIS_i and CO_i and the sets R_i are defined recursively as follows: we let VIS_i and CO_i be defined by (1) for $R = R_i$; we let $R_0 = \emptyset$ and $R_{i+1} = R_i \cup \{(T_i, S_i)\}$, where (T_i, S_i) is an arbitrary pair of transactions unrelated by CO_i; such a pair must exist if CO_i is not total. By Lemma 15, each $(\mathsf{VIS}_i, \mathsf{CO}_i)$ is a solution to the system of inequalities in Figure 3. It is easy to check that $\mathsf{CO}_{i+1} = (\mathsf{CO}_i \cup \{(T_i, S_i)\})^+$, $i = 0..(n-1)$. Since CO_0 is acyclic, by the choice of the edges (T_i, S_i) it follows that CO_i, $i = 1..n$ are acyclic as well. Hence, the sequence $\{\mathcal{P}_i\}_{i=0}^n$ constructed above satisfies the properties stated at the beginning of the proof, as required. $\square$

Proof of Theorem 10(ii). Consider $\mathcal{X} = (\mathcal{T}, \mathsf{SO}, \mathsf{VIS}, \mathsf{CO}) \in \mathsf{ExecSI}$. As follows from Lemma 12, VIS and CO give a solution to the system of inequalities of Figure 3 for $\mathsf{WR} = \mathsf{WR}_\mathcal{X}$, $\mathsf{WW} = \mathsf{WW}_\mathcal{X}$, $\mathsf{RW} = \mathsf{RW}_\mathcal{X}$. We now apply Lemma 15 for $R = \emptyset$; the minimality of the solution given by Lemma 15 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 $\mathsf{graph}(\mathcal{X}) \in \mathsf{GraphSI}$. $\square$

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*** [30] into sessions of smaller transactions without introducing new behaviours (the sessions are also called ***chains*** in this context [38]). 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 $\mathcal{G}_1$ in Figure 4 is not spliceable, because $\mathsf{splice}(\mathcal{H}_{\mathcal{G}_1}) \notin \mathsf{HistSI}$: informally, $\boxed{S}_{\mathcal{G}_1}$ observes the write by $\boxed{T}_{\mathcal{G}_1}$ to `acct1`, but not its write to `acct2`. On the other hand, let $\mathcal{G}_2$ be the graph obtained by removing the transactions S and S' from $\mathcal{G}_1$. Then $\mathcal{G}_2$ is spliceable, as witnessed by the graph $\mathcal{G}_2 \in \mathsf{GraphSI}$ with $\mathcal{H}_{\mathcal{G}_2} = \mathsf{splice}(\mathcal{H}_{\mathcal{G}_2})$ and only the edges $\boxed{T''}_{\mathcal{G}_2} \xrightarrow{\mathsf{RW}_{\mathcal{G}_2}} \boxed{T}_{\mathcal{G}_2}$ and $\boxed{S''}_{\mathcal{G}_2} \xrightarrow{\mathsf{RW}_{\mathcal{G}_2}} \boxed{T}_{\mathcal{G}_2}$.

Given a dependency graph $\mathcal{G}$, we let the ***dynamic chopping graph*** corresponding to $\mathcal{G}$ be the graph $\mathsf{DCG}(\mathcal{G})$ obtained by removing $\mathsf{WR}_\mathcal{G}, \mathsf{WW}_\mathcal{G}$ and $\mathsf{RW}_\mathcal{G}$ edges between transactions related by $\approx_\mathcal{G}$, and by extending $\mathcal{G}$ with edges in the reverse of the session order: $\mathsf{SO}_\mathcal{G}^{-1}$. We refer to the latter edges as ***predecessor*** edges, to those in $\mathsf{SO}_\mathcal{G}$ as ***successor*** edges, and to those in $(\mathsf{WR}_\mathcal{G} \cup \mathsf{WW}_\mathcal{G} \cup \mathsf{RW}_\mathcal{G}) \setminus \approx_\mathcal{G}$ as ***conflict*** edges. A cycle in a chopping graph $\mathsf{DCG}(\mathcal{G})$ is ***critical*** if: *(i)* it does not contain two occurrences of the same vertex; *(ii)* it contains a fragment of three consecutive edges of the form "conflict, predecessor, conflict"; and *(iii)* any two anti-dependency edges ($\mathsf{RW}_\mathcal{G} \setminus \approx_\mathcal{G}$) on the cycle are separated by at least one read ($\mathsf{WR}_\mathcal{G} \setminus \approx_\mathcal{G}$) or write ($\mathsf{WW}_\mathcal{G} \setminus \approx_\mathcal{G}$) dependency edge. Our ***dynamic chopping criterion*** is as follows.

Theorem 16. *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 $\mathcal{G}_2$ (Figure 4) contains no critical cycles, and the graph $\mathcal{G}_1$ contains a critical cycle

$$T' \xrightarrow{\mathsf{WR}_{\mathcal{G}_1}} S' \xrightarrow{\mathsf{SO}_{\mathcal{G}_1}} S \xrightarrow{\mathsf{RW}_{\mathcal{G}_1}} T \xrightarrow{\mathsf{SO}_{\mathcal{G}_1}^{-1}} T'.$$

To prove Theorem 16, 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:

$$\forall T, S \in \mathcal{T}_\mathcal{G}.\, \forall x \in \mathsf{Obj}.\, \boxed{T}_\mathcal{G} \xrightarrow{\mathsf{WR}_{\mathsf{splice}(\mathcal{G})}(x)} \boxed{S}_\mathcal{G} \iff \boxed{T}_\mathcal{G} \neq \boxed{S}_\mathcal{G} \wedge T \xrightarrow{\approx_\mathcal{G} \,;\, \mathsf{WR}_\mathcal{G}(x) \,;\, \approx_\mathcal{G}} S. \quad (2)$$

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 5. As the following lemma shows, $\mathsf{RW}_{\mathsf{splice}(\mathcal{G})}$ defined in this way can be decomposed into a form similar to (2).

Lemma 17. *Let $\mathcal{G} \in \mathsf{GraphSI}$ be such that $\mathsf{DCG}(\mathcal{G})$ contains no critical cycles. Then*

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

To prove Theorem 16, we assume $\mathsf{splice}(\mathcal{G}) \notin \mathsf{GraphSI}$ and use Theorem 9 to obtain a cycle in $(\mathsf{WR}_{\mathsf{splice}(\mathcal{G})} \cup \mathsf{WW}_{\mathsf{splice}(\mathcal{G})})$;

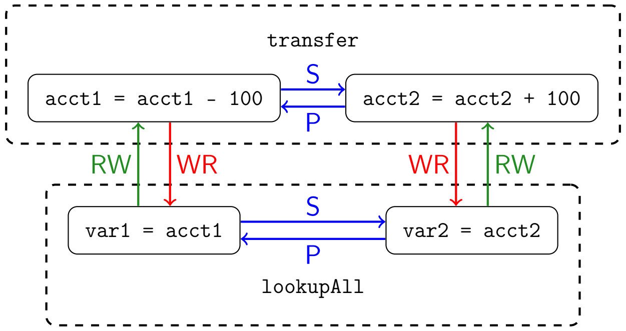


Figure 5: The static chopping graph of the programs {transfer, lookupAll} from Figure 4. Dashed boxes group program pieces into sessions.

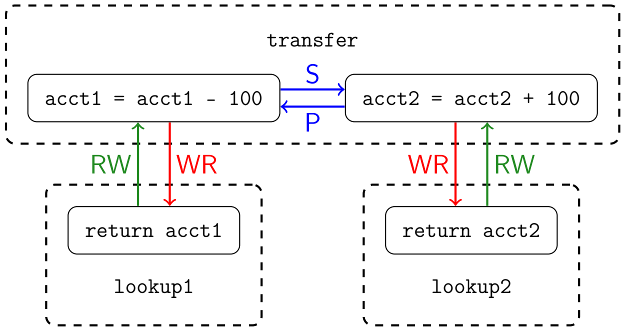


Figure 6: The static chopping graph of the programs {transfer, lookup1, lookup2} from Figure 4.

$\mathsf{RW}_{\mathsf{splice}(\mathcal{G})}$?. We then use (2) and Lemma 17 to decompose this cycle into a critical cycle in $\mathsf{DCG}(\mathcal{G})$, yielding a contradiction.

We now derive a static analysis from Theorem 16. Assume a set of ***programs*** $\mathcal{P} = \{P_1, P_2, \dots\}$, 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***, defining the code of the transactions in the sessions. We further assume that we are given the sets R^i_j and W^i_j of all objects that can respectively be read and written by the j-th piece of P_i. For example, the program `transfer` in Figure 4 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. [30], 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.

A history $\mathcal{H}$ ***can be produced*** by the programs $\mathcal{P}$, if there is a one-to-one correspondence between every session in $\mathcal{H}$ and a program $P_i \in \mathcal{P}$ whose read and write sets cover the sets of objects read or written by the corresponding transactions in the session. For example, the history in Figure 4 can be produced by the programs in the figure. 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 $\mathcal{P}$ using its ***static chopping graph*** $\mathsf{SCG}(\mathcal{P})$. It is a 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); $i_1 = i_2$ and $j_1 > j_2$ (a ***predecessor*** edge); $i_1 \neq i_2$ and $W^{i_1}_{j_1} \cap R^{i_2}_{j_2} \neq \emptyset$ (a ***read dependency*** edge); $i_1 \neq i_2$ and $W^{i_1}_{j_1} \cap W^{i_2}_{j_2} \neq \emptyset$ (a ***write dependency*** edge); or $i_1 \neq i_2$ and $R^{i_1}_{j_1} \cap W^{i_2}_{j_2} \neq \emptyset$ (an ***anti-dependency*** edge). 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 16 we easily get our static analysis.

COROLLARY 18. *The chopping defined by $\mathcal{P}$ is correct if $\mathsf{SCG}(\mathcal{P})$ contains no critical cycles.*

In Figure 5 we show the static chopping graph of the programs {`transfer`, `lookupAll`}, which contains a critical cycle:

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

In fact, since the dependency graph in Figure 4 is not spliceable, the chopping defined by the above programs is incorrect. In Figure 6 we show the static chopping graph of the programs {`transfer`, `lookup1`, `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.

Our SI characterisation is instrumental in deriving the above static analysis due to the ease of splicing a dependency graph (cf. (2)). As we explain in [11, §B], splicing abstract executions directly would be problematic.

In [11, §B], we also show that the conditions on chopping required by Corollary 18 are laxer than those of the analysis for serializability [30], but stricter than those for parallel SI [12]. In particular, this implies that the classical transaction chopping analysis for serializability is also sound for SI. 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.

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 [19][1] (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 8 and 10.

[1]Our analysis does not take into account the notion of a *vulnerable* dependency from [19].

THEOREM 19. *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.*

Fekete et al. previously established a result corresponding to the "only if" direction of the above theorem [19]. 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.

The dependency graph $\mathcal{G}_3$ of the write skew anomaly in Figure 2(d) 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 $\mathcal{G}_1$ from Figure 4 does not contain any cycles, so that $\mathcal{G}_1 \notin \mathsf{GraphSI} \setminus \mathsf{GraphSER}$; in fact, $\mathcal{G}_1 \in \mathsf{GraphSER}$. Finally, the dependency graph $\mathcal{G}_4$ of the long fork anomaly in Figure 4 contains a cycle, but without two adjacent anti-dependencies; hence, $\mathcal{G}_4 \notin \mathsf{GraphSI} \setminus \mathsf{GraphSER}$, and in fact, $\mathcal{G}_4 \notin \mathsf{GraphSI}$.

We can derive a static analysis from Theorem 19 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 $\mathcal{P}$ with given read and write sets. Based on these sets, it constructs a ***static dependency graph***, over-approximating possible dependencies that can exist in executions of the programs $\mathcal{P}$. The analysis then checks that the graph has no cycles with at least two adjacent anti-dependency edges. By Theorem 19 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. Note that the dependency graphs characterisation of consistency models greatly facilitates deriving the above static 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 [32] behaves the same as when executing under the classical SI (robustness ***against parallel SI towards SI***). 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 [10].

DEFINITION 20. *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. 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(c), disallowed by the axiom PREFIX in SI.

Following [12, extended version, Lemma 14], we can give a characterisation of parallel SI in terms of dependency graphs.

THEOREM 21. *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 $\mathcal{G}_4$ in Figure 2(c). It is easy to see that all its cycles contain at least two anti-dependencies, and therefore $\mathcal{G}_4 \in \mathsf{GraphPSI}$. On the other hand, let $\mathcal{G}_5$ be the dependency graph in Figure 2(b). The graph $\mathcal{G}_5$ contains a cycle with exactly one anti-dependency ($T_1 \xrightarrow{\mathsf{WW}_{\mathcal{G}_5}} T_2 \xrightarrow{\mathsf{RW}_{\mathcal{G}_5}} T_1$), and therefore $\mathcal{G}_5 \notin \mathsf{GraphPSI}$. As a corollary of Theorems 9 and 21, we obtain a dynamic robustness criterion that checks whether a given dependency graph is in $\mathsf{GraphPSI} \setminus \mathsf{GraphSI}$.

THEOREM 22. *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 $\mathcal{G}_4$ of the long fork anomaly all cycles have at least two anti-dependencies. Furthermore, $\mathcal{G}_4$ also has a cycle with no adjacent anti-dependencies: $T_1 \xrightarrow{\mathsf{WR}_{\mathcal{G}_4}} T_3 \xrightarrow{\mathsf{RW}_{\mathcal{G}_4}} T_2 \xrightarrow{\mathsf{WR}_{\mathcal{G}_4}} T_4 \xrightarrow{\mathsf{RW}_{\mathcal{G}_4}} T_1$, so that $\mathcal{G}_4 \in \mathsf{GraphPSI} \setminus \mathsf{GraphSI}$. The dependency graph $\mathcal{G}_3$ of the write skew anomaly in Figure 2(d) 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 $\mathcal{G}_5$ of the lost update anomaly contains a cycle with exactly one anti-dependency, so that $\mathcal{G}_5 \notin \mathsf{GraphPSI} \setminus \mathsf{GraphSI}$; in fact, $\mathcal{G}_5 \notin \mathsf{GraphPSI}$.

From Theorem 22 it follows that the desired static analysis can check that the static dependency graph of an application contains no cycles where there are at least two anti-dependency edges and no two anti-dependency edges are adjacent.

7. RELATED WORK

Snapshot isolation was originally defined by an idealised algorithm formulated in terms of implementation-level concepts [6]. Since then there have been proposals of more declarative SI specifications [2, 10, 28], one of which [10] 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. [19] 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 10(ii)), but they did not establish an analogue of our soundness result (Theorem 10(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 [9, 37] and proving the correctness of concurrency-control algorithms [16, 36]. 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* [34].

The constraint on dependency graphs that we use to characterise SI also arose in the work of Lin et al. [24], 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 [26, 36, 38]. There have also been proposals of consistency models for transactional memory that weaken consistency guarantees in a way similar to chopping [4, 20, 35]. Our chopping analysis enables bringing these benefits to transactional systems providing SI. We have previously proposed a chopping analysis for parallel SI [12], which also relies on a dependency graph characterisation of this consistency model (Theorem 21). 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 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.

References

[1] The Clojure language: Refs and transactions. http://clojure.org/refs.

[2] A. Adya. Weak consistency: A generalized theory and optimistic implementations for distributed transactions. PhD thesis, MIT, 1999.

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

[4] Y. Afek, H. Avni, and N. Shavit. Towards consistency oblivious programming. In *OPODIS*, 2011.

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

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

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

[8] A. Bieniusa and T. Fuhrmann. Consistency in hindsight: A fully decentralized STM algorithm. In *IPDPS*, 2010.

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

[10] A. Cerone, G. Bernardi, and A. Gotsman. A framework for transactional consistency models with atomic visibility. In *CONCUR*. Dagstuhl, 2015.

[11] A. Cerone and A. Gotsman. Analysing snapshot isolation (extended version). Available from www.software.imdea.org/~gotsman, 2016.

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

[13] K. Daudjee and K. Salem. Lazy database replication with ordering guarantees. In *ICDE*, 2004.

[14] K. Daudjee and K. Salem. Lazy database replication with snapshot isolation. In *VLDB*, 2006.

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

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

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

[18] S. Elnikety, W. Zwaenepoel, and F. Pedone. Database replication using generalized snapshot isolation. In *SRDS*, 2005.

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

[20] P. Felber, V. Gramoli, and R. Guerraoui. Elastic transactions. In *DISC*, 2009.

[21] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, 2008.

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

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

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

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

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

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

[28] M. Saeida Ardekani, P. Sutra, M. Shapiro, and N. Preguiça. On the scalability of snapshot isolation. In *Euro-Par*, 2013.

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

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

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

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

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

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

[35] L. Xiang and M. L. Scott. Software partitioning of hardware transactions. In *PPoPP*, 2015.

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

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

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