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 [21]. 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 [13, 26, 28] and transactional memory systems [1, 8, 14, 24].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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 [11, 18, 22, 29, 37], run-time monitoring [9, 36] and proving concurrency-control algorithms correct [15, 23, 35]. 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, 35]. But surprisingly, there is no such spec-
ification 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, 29, 34]—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, producing new behaviours (§5). When applied to long-running transactions, which ensure that the transactions will be executed in the order given, but provide no isolation guarantees. A chopping is committed: our specifications do not constrain values read inside objects, and in our representation of executions, we denote each component of transactions and simulations, which establish an ordering on the transactions.

For simplicity, all transactions in this paper are assumed to be disjoint subsets of $T \times T$ are such that $\sigma(T, SO)$ is a union of total orders defined on $T$, which correspond to transactions in different sessions. We denote components of transactions and similar structures as in $E_T$ and $\sigma(T, SO)$.

To allow transaction chopping (§5), we assume that the transactional system allows its clients to group several transactions into a session [32], which establishes an ordering on the transactions. Thus, instead of classical SI and serializability, we actually define their strong session variants [12, 13]. We represent the client-visible results of an execution of a set of sessions by a history.

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.

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.
∀(E, po) ∈ T. ∀e ∈ E. ∀x, n. op(e) = read(x, n) ∧ \{ f | op(f) = \_\(x, \_\) ∧ f \(\rightarrow\) e \} \neq \emptyset \implies \text{SO} \subseteq \text{VIS} \quad (\text{SESSION})

\text{CO} \lor \text{VIS} \subseteq \text{VIS} \quad (\text{PREFIX})

∀T ∈ T. ∀x, n. T \vdash \text{read}(x, n) \implies \text{max}_{CO}(\text{VIS}^{-1}(T) \cap \text{WriteTx}_e) \vdash \text{write}(x, n) \quad (\text{EXT})

\text{CO} = \text{VIS} \quad (\text{TotalVIS})

∀T, s ∈ T. ∀x. (T, S ∈ \text{WriteTx}_e ∧ T \neq S) \implies (T \overset{\text{VIS}}{\rightarrow} S \cup S \overset{\text{VIS}}{\rightarrow} T) \quad (\text{NoConflict})

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

(a) Session guarantees.

\begin{align*}
T_1 & \quad \text{write}(x, 1) \\
\text{SO}, \text{VIS}, \text{CO} & \quad \text{read}(x, 1)
\end{align*}

if \((\text{acct1} + \text{acct2} > 100)\):
\begin{align*}
\text{acct1} & := \text{acct1} - 100 \\
\text{acct2} & := \text{acct2} - 100
\end{align*}

(b) Lost update.

\begin{align*}
T_1 & \quad \text{acct} := \text{acct} + 50 \\
\text{read}(\text{acct}, 0) & \rightarrow \text{write}(\text{acct}, 50) \\
\text{RW}, \text{WW} & \quad \text{CO} \quad \text{RW} \\
\text{read}(\text{acct}, 0) & \rightarrow \text{write}(\text{acct}, 25) \\
T_2 & \quad \text{acct} := \text{acct} + 25
\end{align*}

\begin{align*}
T_3 & \quad \text{VIS} \\
\text{RW} & \quad \text{CO} \quad \text{RW} \\
\text{read}(\text{acct}, 25) & \rightarrow \text{write}(\text{acct}, 0)
\end{align*}

(c) Long fork.

\begin{align*}
T_1 & \quad \text{write}(x, 1) \\
\text{VIS} & \quad \text{RW} \\
\text{read}(x, 1) & \rightarrow \text{read}(y, 0) \\
\text{RW} & \quad \text{WR} \\
\text{read}(y, 1) & \rightarrow \text{write}(x, 0) \\
\text{VIS} & \quad \text{WR} \\
T_2 & \quad \text{read}(x, 0) \rightarrow \text{read}(y, 1)
\end{align*}

\begin{align*}
T_3 & \quad \text{VIS} \\
\text{RW} & \quad \text{WR} \\
\text{read}(x, 1) & \rightarrow \text{read}(y, 0)
\end{align*}

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 \(T \overset{\text{VIS}}{\rightarrow} S\) and \((T, S) \in \text{VIS}\) interchangeably, and similarly for other relations. For \(H = (T, \text{SO})\) we shorten \((T, \text{SO}, \text{VIS}, \text{CO})\) to \((H, \text{VIS}, \text{CO})\). In terms of the SI concurrency-control algorithm sketched in §1, \(T \overset{\text{VIS}}{\rightarrow} S\) means that the writes done by the transaction \(T\) are included into the snapshot taken by the transaction \(S\); \(T \overset{\text{CO}}{\rightarrow} S\) means that \(T\) commits earlier than \(S\). The constraint \(\text{VIS} \subseteq \text{CO}\) 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:

- \(\text{ExecSI} = \{X | X = \text{INT} \land \text{EXT} \land \text{SESSION} \land \text{PREFIX} \land \text{NOCONFlict}\}\);
- \(\text{ExecSER} = \{X | X = \text{INT} \land \text{EXT} \land \text{SESSION} \land \text{TOTALVIS}\}\);
- \(\text{HistSI} = \{H | \exists \text{VIS}, \text{CO} \in \text{ExecSI}\}\);
- \(\text{HistSER} = \{H | \exists \text{VIS}, \text{CO} \in \text{ExecSER}\}\).

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 \(\text{max}_R(A)\) be the element \(a \in A\) such that \(\forall b \in A. a = b \lor b \in R;\) if \(A = \emptyset\), then \(\text{max}_R(A)\) is undefined. In the following, the use of \(\text{max}_R(A)\) in an expression implicitly assumes that it is defined. We define \(\text{min}_R(A)\) similarly. For a relation \(R \subseteq A \times A\) and an element \(a \in A\), we let \(R^{-1}(a) = \{b | (b, a) \in R\}\).

We define the sequential composition of relations \(R_1\) and \(R_2\) as

\[R_1 \circ R_2 = \{(a, b) | \exists c. (a, c) \in R_1 \land (c, b) \in R_2\}\] .

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

The \text{INT} and \text{EXT} axioms in Figure 1 ensure that a transaction reads from a snapshot of object states and its own writes. The \text{internal consistency axiom} \text{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 \text{external consistency axiom} \text{EXT}. For \(T = (E, po)\), we let \(T \vdash \text{write}(x, n)\) if \(T\) writes to \(x\) and the last value written is \(n\):

\[\text{op}(\text{max}_po(e | \text{op}(e) = \text{write}(x, _))) = \text{write}(x, n)\] .
We let $T \vdash \text{read}(x, n)$ if $T$ reads from $x$ before writing to it and $n$ is the value returned by the first such read:

$$\text{op}(\min_{\text{po}} \{ e \mid \text{op}(e) = \langle x, \_ \rangle \}) = \text{read}(x, n).$$

We also let $\text{WriteTx}_X = \{ T \mid T \vdash \text{write}(x, \_ ) \}$. Then $\text{EXT}$ 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 $\text{EXT}$.

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 [17]. However, following strong session SI [12, 13], 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 $\text{SESSION}$.

The $\text{PREFIX}$ 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 $\text{PREFIX}$ and the property $\text{VIS} \subseteq \text{CO}$ in Definition 4 imply that VIS is transitive. $\text{PREFIX}$ disallows the long fork anomaly shown in Figure 2(c), which is allowed by some weakening of SI (such as parallel SI [31]). 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, $T_4$ 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. $\text{PREFIX}$ 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 $\text{PREFIX}$, either $T_1$ 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 acc$. In the SI concurrency-control this is ensured by the write-conflict detection check ($\S 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 ($\S 1$), 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 HistSER $\subseteq$ 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 $X = (H, \text{VIS}, \text{CO})$ be an execution. For $x \in \text{Obj}$, we define the following relations on $T_H$:

- **read dependency:** $T \xrightarrow{\text{WR}_X(x)} S \iff S \vdash \text{read}(x, \_ ) \land T = \text{max}_{\text{CO}}(\text{VIS}^{-1}(S) \cap \text{WriteTx}_x);$
- **write dependency:** $T \xrightarrow{\text{WW}_X(x)} S \iff T \not\subseteq S \land T, S \in \text{WriteTx}_x;$
- **anti-dependency:** $T \xrightarrow{\text{RW}_X(x)} S \iff T \neq S \land \exists T'. T' \xrightarrow{\text{WR}_X(x)} T \land T' \xrightarrow{\text{WW}_X(x)} S.$

Informally, $T \xrightarrow{\text{WR}_X(x)} S$ means that $S$ reads $T$’s write to $x$ (cf. the EXT axiom in Figure 1); $T \xrightarrow{\text{WW}_X(x)} S$ means that $S$ overwrites $T$’s write to $x$; $T \xrightarrow{\text{RW}_X(x)} S$ 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 $\text{WR}_X$ to also denote the relation $\bigcup_{x \in \text{Obj}} \text{WR}_X(x) \subseteq T_H \times T_H$, and similarly for $\text{WW}_X$ and $\text{RW}_X$.

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 $G = (T, \text{SO}, \text{WR}, \text{WW}, \text{RW})$, where $(T, \text{SO})$ is a history and

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

**Proposition 7.** For any $X \in \text{ExecSI}$, graph$(X) = (T_X, \text{SO}_X, \text{WR}_X, \text{WW}_X, \text{RW}_X)$ is a dependency graph.

Note that the constraints on WR in Definition 6 ensure that it uniquely determines the values read by transactions. For $H = (T, \text{SO})$ we write $(H, \text{WR}, \text{WW}, \text{RW})$ for $(T, \text{SO}, \text{WR}, \text{WW}, \text{RW})$.

We write $T \models \text{INT}$ if a set of transactions $T$ 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

$$\text{GraphSER} = \{ G \mid (T_G \models \text{INT}) \land ((\text{SO}_G \cup \text{WR}_G \cup \text{WW}_G \cup \text{RW}_G) \text{ is acyclic}) \}. $$
Then
\[ \text{HistSER} = \{ H \mid \exists \text{WR}, \text{WW}, \text{RW}. \}
\]
\[ (H, \text{WR}, \text{WW}, \text{RW}) \in \text{GraphSER}. \]

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 \( T \) and a relation \( R \subseteq T \times T \) let \( R^? = R \cup \{(T, T) \mid T \in T\} \). 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
\[ \text{GraphSI} = \{ G \mid \langle T_0 \mid \text{INT} \rangle \wedge \langle (SO_G \cup WR_G \cup WW_G) ; \text{RW}_G \rangle \text{ is acyclic} \} \}

Then
\[ \text{HistSI} = \{ H \mid \exists \text{WR}, \text{WW}, \text{RW}. (H, \text{WR}, \text{WW}, \text{RW}) \in \text{GraphSI} \}. \]

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., \( T_1 \rightarrow \text{RW} \rightarrow T_2 \rightarrow \text{RW} \rightarrow T_1 \)). 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 contain cycles without these: e.g., \( T_1 \rightarrow \text{WW} \rightarrow T_2 \rightarrow \text{RW} \rightarrow T_1 \) in Figure 2(b) and \( T_1 \rightarrow \text{WR} \rightarrow T_3 \rightarrow \text{RW} \rightarrow T_2 \rightarrow \text{RW} \rightarrow T_4 \rightarrow \text{RW} \rightarrow T_1 \) 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:** \( \forall G \in \text{GraphSI}. \exists \chi \in \text{ExecSI}. \text{graph}(\chi) = G \).
(ii) **Completeness:** \( \forall \chi \in \text{ExecSI}. \text{graph}(\chi) \in \text{GraphSI} \).

As we explain in §7, the easier completeness direction of this theorem actually follows from existing results [18]. 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 \( \chi \) from the dependencies given by \( G \) 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 \( P = (T, \text{SO}, \text{VIS}, \text{CO}) \) 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 \( \text{PreExecSI} \) be the set of pre-executions satisfying the SI axioms (Figure 1):
\[
\text{PreExecSI} = \{ P \mid P \models \text{INT} \land \text{EXT} \land \text{SESSION} \land \text{PREFIX} \land \text{NOCONFLICT} \}.
\]

\[
\begin{align*}
\{ & \text{SO} \cup \text{WR} \cup \text{WW} \subseteq \text{VIS} \} \\
& \{ \text{CO} \land \text{VIS} \subseteq \text{VIS} \} \\
& \{ \text{VIS} \subseteq \text{CO} \} \\
& \{ \text{CO} \land \text{CO} \subseteq \text{CO} \} \\
& \{ \text{VIS} \land \text{RW} \subseteq \text{CO} \}
\end{align*}
\]

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

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 \( P \in \text{PreExecSI} \), graph(\( P \)) is indeed a dependency graph.

We first obtain auxiliary results that, given a dependency graph \( G = (T, \text{SO}, \text{WR}, \text{WW}, \text{RW}) \in \text{GraphSI} \), allow us to construct a pre-execution \( P = (T, \text{SO}, \text{VIS}, \text{CO}) \in \text{PreExecSI} \) such that graph(\( P \)) = \( G \). Later we show how to extend \( P \) to a desired execution \( \chi \in \text{ExecSI} \). We start by restating the requirements on the pre-execution in a way more suitable for guiding its construction; these are given by the system of inequalities in Figure 3. First, to ensure graph(\( P \)) = \( G \), by Definition 5 at the very least we must have \( \text{WR} \cup \text{WW} \subseteq \text{VIS} \). For \( P \) to satisfy the SESSION axiom we must also have \( \text{SO} \subseteq \text{VIS} \). These two observations motivate (S1). This inequality also implies that \( P \) satisfies NOCONFLICT, since according to Definition 5, \( \text{WW} \) is total over transactions that write to a given object. Inequality (S2) is equivalent to PREFIX, and inequality (S3) states a relationship between \( \text{VIS} \) and \( \text{CO} \) inherited by Definition 11 from Definition 3. Inequality (S4) requires \( \text{CO} \) to be transitive; (S2) and (S3) ensure that so is \( \text{VIS} \).

As we now explain, (S5) ensures the axiom EXT. Consider the dependency graph \( G \in \text{GraphSI} \) 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 1000kp1 and 1000kp2 see the initial state of the database, while the transactions arising from 1000kp11 see its state in the middle of a transfer. The \( \text{VIS} \) and \( \text{CO} \) 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 \( \text{CO} \) by adding an edge \( T \ \text{CO} \rightarrow S' \) (shown dotted). Since \( S' \ \text{VIS} \rightarrow S \), for the resulting pre-execution to satisfy PREFIX we also need to add an edge \( T \ \text{VIS} \rightarrow S \). 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 \( \text{CO} \) and \( \text{WW} \); the latter fact is witnessed by the edge \( S \ \text{RW} \rightarrow T \). On the other hand, adding an edge \( S' \ \text{CO} \rightarrow T \) (shown dashed), which belongs to \( \text{VIS} \); \( \text{RW} \), 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.** \( \forall \chi \in \text{ExecSI}. \text{VIS}_\chi \land \text{RW}_\chi \subseteq \text{CO}_\chi \).

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

**Lemma 13.** Let \( G = (T, \text{SO}, \text{WR}, \text{WW}, \text{RW}) \) be a dependency graph such that \( T \models \text{INT} \) and \( \text{VIS}, \text{CO} \subseteq T \times T \) be acyclic relations satisfying the system of inequalities in Figure
3. Then \( P = (T, SO, VIS, CO) \) is a pre-execution such that \( P \in \text{PreExecSI} \) and \( \text{graph}(P) = \mathcal{G} \).

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

**Proposition 14.**
\[
\forall X \in \text{ExecSI}. \forall T, S \in T_X. S^\text{RW} \times_X T \iff S \neq T \land \\
\exists x. S \vdash \text{read}(x, n) \land T \vdash \text{write}(x, n) \land (T \xrightarrow{\text{VIS}}_X S).
\]

Informally, if we had \( S \xrightarrow{\text{RW}} T \) and \( T \xrightarrow{\text{VIS}}_X S \), 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 \( \text{RW} \).

**Proof of Lemma 12.** Consider \( X \in \text{ExecSI} \) and \( T, S', S \in T_X \) such that \( S' \xrightarrow{\text{VIS}}_X S \xrightarrow{\text{RW}} T \). If \( T = S' \), then \( S' \xrightarrow{\text{VIS}}_X S \), contradicting Proposition 14. If \( T \xrightarrow{\text{CO}}_X S' \) (see Figure 4), then by \text{PREFIX} we get \( T \xrightarrow{\text{VIS}}_X S \), contradicting Proposition 14. Then, since \( CO_X \) is total, we must have \( S' \xrightarrow{\text{CO}}_X T \).

**Proof of Lemma 13.** We only prove that \( P \models \text{EXT} \) and \( WR_P = \text{WR} \); discharging the other obligations is straightforward. Consider \( S \in T \) such that \( S \vdash \text{read}(x, n) \). Then there exists a unique \( T' \) such that \( T' \xrightarrow{\text{WR}} S \). Let \( T = \max_{CO} (\text{VIS}^{-1}(S) \cap \text{WriteTx}_x) \). This is defined because: \( CO \) is acyclic; by (S1) and (S3) we have \( WW \subseteq CO \), so that \( CO \) is total over \( \text{WriteTx}_x \); and by (S1) we have \( WR \subseteq VIS \), so that \( T' \in \text{VIS}^{-1}(S) \cap \text{WriteTx}_x \). We now show that \( T = T' \), which entails the required.

Assume the contrary: \( T \neq T' \). We have \( T, T' \in \text{VIS}^{-1}(S) \cap \text{WriteTx}_x \). Hence, \( T \) and \( T' \) are related by \( WW \). Since \( WW \subseteq CO \), they are related in the same way by the acyclic \( CO \). Then by the definition of \( T \) we must have \( T' \xrightarrow{\text{CO}}(x) \) and \( T' \xrightarrow{\text{CO}}(x) \).

From the latter and \( T \xrightarrow{\text{WR}} S \) we get \( S \xrightarrow{\text{RW}} T \). But \( T \xrightarrow{\text{VIS}} S \), so from (S5) we get \( T \xrightarrow{\text{CO}} T \). This contradicts the assumption that \( CO \) is acyclic. Hence, we must have \( T = T' \).

According to Lemma 13, to construct a desired pre-execution \( P = (T, SO, VIS, CO) \in \text{PreExecSI} \) from a dependency graph \( G = (T, SO, WR, WW, RW) \), it is sufficient to find a solution to the system of inequalities from 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 \( G = (T, SO, WR, WW, RW) \) be a dependency graph. For any \( R \subseteq T \times T \), the relations
\[
\text{VIS} = (((SO \cup WR \cup WW) \setminus R^+) \cap R^*) \\
\text{CO} = (((SO \cup WR \cup WW) \setminus R^+) \cap R^*)
\]
are a solution to the system of inequalities in Figure 3. They also are the smallest solution to the system for which \( CO \subseteq R \): for any other solution \( (VIS', CO') \) with \( CO' \subseteq R \) we have \( VIS' \subseteq VIS \) and \( CO \subseteq CO' \).

In particular, for \( R = \emptyset \), Lemma 15 gives the smallest solution \( (VIS_0, CO_0) \) to the system of inequalities in Figure 3.

If \( G \in \text{GraphSI} \), then \( CO_0 \) is acyclic and, by (S3), so is \( VIS_0 \) (in fact, Lemma 15 is our motivation for defining \( \text{GraphSI} \) the way we did). Hence, by Lemma 13, \( P_0 = (T, SO, VIS_0, CO_0) \) is a pre-execution such that \( P_0 \in \text{PreExecSI} \) and \( \text{graph}(P_0) = G \), which is what we originally set out to construct. We now proceed to prove Theorem 10(i) by extending the pre-execution \( P_0 \) to an execution \( X' \in \text{ExecSI} \).

**Proof of Theorem 10(i).** Assume \( G = (T, SO, WR, WW, RW) \in \text{GraphSI} \). To construct \( X' \), we define a sequence of pre-executions \( P_i = (T, SO, VIS_i, CO_i) \) for some \( n \geq 0 \), where \( X' = P_n \).

The sequence is such that \( VIS_n \subseteq VIS_{n+1} \) and \( CO_i \subseteq CO_{i+1} \), for \( i = 0, \ldots, n-1 \); furthermore, \( CO_n \) is total, so that \( P_n \) is an execution. That is, on every step of our construction we add edges to the commit order until it becomes total. Each pair \( (VIS_i, CO_i) \) gives an acyclic solution to the system in Figure 3. Then by Lemma 13, \( P_n \in \text{PreExecSI} \) and \( \text{graph}(P_n) = G \). In particular, \( P_n \in \text{ExecSI} \) and \( \text{graph}(P_n) = G \), 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 $S' \xrightarrow{CO_0} T$. 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 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 VIS$_1 = $ VIS$_0$, and CO$_1 = CO_0 \cup \{(T', T'')\}$ In general, the construction continues in the same way: while CO$_0$ is not total, we pick an arbitrary pair of transactions $(T_1, S_1)$ unrelated by CO$_0$ and force CO$_{i+1}$ to include it. In our example, CO$_0$ does not relate the transactions $T'$ and $T''$. 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$_1$ and CO$_1$, we construct sets $R_i = \{(T_k, S_k) \mid k = 1 \ldots i\}$, $i = 0 \ldots n$ that accumulate the edges enforced in the commit order at every step. The relations VIS$_i$, CO$_i$, and the sets $R_i$ are defined recursively as follows: we let VIS$_1$, CO$_1$, and $R_0 = 0$ 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$_1$ is not total. By Lemma 15, each (VIS$_i$, CO$_i$) is a solution to the system of inequalities in Figure 3. It is easy to check that CO$_{i+1} = (CO_i \cup \{(T_i, S_i)\})^{i+1}$, $i = 0 \ldots (n - 1)$. Since CO$_0$ is acyclic, by the choice of the edges $(T_i, S_i)$ it follows that CO$_i$, $i = 1 \ldots n$ 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.

**Proof of Theorem 10(ii).** Consider $\mathcal{X} = (T, SO, VIS, CO) \in \text{ExecSI}$. As follows from Lemma 12, VIS and CO give a solution to the system of inequalities of Figure 3 for WR = WR$_X$, WW = WW$_X$, RW = RW$_X$. We now apply Lemma 15 for $R = \emptyset$; the minimality of the solution given by Lemma 15 implies that $(SO \cup WR \cup WW) : RW?^{n} \subseteq CO$. Then $(SO \cup WR \cup WW) : RW?^{n+1}$ is acyclic because so is CO. This establishes graph($\mathcal{X}$) $\in$ GraphSI.

**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 [29] into sessions of smaller transactions without introducing new behaviours (the sessions are also called chains in this context [37]). 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 $H$, let $\approx_H = SO_H \cup SO_H^{-1} \cup \{(T, T) \mid T \in T_H\}$ be the equivalence relation grouping transactions from the same session. We let $\quad \sum_H$ be the result of splicing all transactions in the session to which $T$ belongs in $H$ into a single transaction: $\quad \sum_H = (E, po)$, where $E = (\{[E_S \mid S \approx_H T\})$ and

$$po = \{(e, f) \mid (\exists S. e, f \in E_S \land e \cdash RW_S f \land S \approx_H T) \lor (\exists S'. e \in E_S \land f \in E_{S'} \land S \cdash SO_H S' \land S' \approx_H T)\}.$$ We let splice($H$) be the history resulting from splicing all sessions in a history $H$: splice($H$) = \{[\quad \sum_H \mid T \in T_H\}, \emptyset\}. A dependency graph $G \in$ GraphSI is spliceable if there exists a dependency graph $G' \in$ GraphSI such that $H_{G'} = $ splice($H_G$). For a dependency graph $G$, we let $\approx_g = \approx_K$ and $\quad \sum_g =$ $\quad \sum_H$.

For example, the graph $G_1$ in Figure 4 is not spliceable, because splice($H_{G_1}$) $\notin$ HistSI: informally, $\quad \sum_{G_1}$ observes the write by $\quad \sum_H$ to acct1, but not its write to acct2. On the other hand, $G_2$ is spliceable, as witnessed by the graph $G_2 \in$ GraphSI with $H_{G_2} = $ splice($H_{G_2}$) and only the edges $\quad \sum_g \approx_g = \text{RW}_2$ $\quad \sum_g$, $\quad \sum_g$, $\quad \sum_g$. Given a dependency graph $G$, we let the dynamic chopping graph corresponding to $G$ be the graph DCG($G$) obtained by removing WR$_G$, WW$_G$ and RW$_G$ edges between transactions related by $\approx_g$, and by extending $G$ with edges in the reverse of the session order: $SO_G^{-1}$. We refer to the latter edges as predecessors, edges, to those in $SO_G$ as successor, edges, and to those in ($WR_G \cup WW_G \cup RW_G$) $\approx_g$ as conflict edges. A cycle in a chopping graph DCG($G$) 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 ($RW_G \approx_g$ or write ($WW_G \approx_g$) dependency edge. Our dynamic chopping criterion is as follows.

**Theorem 16.** For $G \in$ GraphSI, if DCG($G$) contains no critical cycles, then $G$ is spliceable.

For example, the above graph $G_2$ (Figure 4) contains no critical cycles, and the graph $G_1$ contains a critical cycle

$$\quad \sum_g \approx_g = \text{RW}_G \quad \sum_g \approx_g = \text{SO}_G^{-1} \quad \sum_g \approx_g,$$ $T'$. To prove Theorem 16, we exhibit a particular dependency graph splice($G$) such that splice($G$) $\in$ GraphSI and $H_{\text{splice}(G)} = $ splice($H_G$). We define read dependencies WR$_{\text{splice}(G)}$ by lifting those in WR$_G$ to spliced transactions:

$$\forall T, S \in T_G, \forall x \in \text{Obj}, \quad \sum_g \approx_g \implies \quad \sum_g \implies \quad \sum_g \iff S.$$

We define WW$_{\text{splice}(G)}$ similarly and derive RW$_{\text{splice}(G)}$ from WR$_{\text{splice}(G)}$ and WW$_{\text{splice}(G)}$ as in Definition 5. As the following lemma shows, RW$_{\text{splice}(G)}$ defined in this way can be decomposed into a form similar to (2).

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

$$\forall T, S \in T_G, \forall x \in \text{Obj}, \quad \sum_g \approx_g \implies \quad \sum_g \implies \quad \sum_g \implies S.$$

To prove Theorem 16, we assume splice($G$) $\notin$ GraphSI and use Theorem 9 to obtain a cycle in ($WR_{\text{splice}(G)} \cup WW_{\text{splice}(G)}$) ;
We then use (2) and Lemma 17 to decompose this cycle into a critical cycle in $DCG(G)$, yielding a contradiction.

We now derive a static analysis from Theorem 16. Assume a set of programs $P = \{P_1, P_2, \ldots\}$, 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, 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. [29], 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 $H$ can be produced by the programs $P$, if there is a one-to-one correspondence between each session in $H$ and a program $P_i \in P$ 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 $P$ is correct if every dependency graph $G \in \text{GraphSI}$, where $H$ can be produced by $P$, is spliceable.

We check the correctness of $P$ using its static chopping graph $\text{SCG}(P)$. It is a directed graph whose nodes are pairs of indices identifying the pieces in $P$: $\{(i, j) \mid i = 1, \ldots, |P|, \ j = 1, \ldots, k_i\}$. 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 $W_{j_1}^i \cap R_{j_2}^i \neq \emptyset$ (a read dependency edge); $i_1 \neq i_2$ and $W_{j_1}^i \cap W_{j_2}^i \neq \emptyset$ (a write dependency edge); or $i_1 \neq i_2$ and $R_{j_1}^i \cap W_{j_2}^i \neq \emptyset$ (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 $\text{SCG}(P)$ over-approximates the edge sets of dynamic graphs $DCG(G)$ corresponding to dependency graphs $G$ produced by the programs $P$. From this observation and Theorem 16 we easily get our static analysis.

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

We now consider another type of 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. ROBUSTNESS CRITERIA FOR SI**

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

\[
\text{var1 = acct1} \xrightarrow{\text{RW}} \text{acct1 = acct1 - 100} \xrightarrow{\text{WR}} \text{var2 = acct2} \xrightarrow{\text{RW}} \text{acct2 = acct2 + 100} \xrightarrow{\text{WR}} \text{var1 = acct1}.
\]

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 $\{\text{transfer,lookup1,lookup2}\}$. 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 §B, splicing abstract executions directly would be problematic.

In §B, we also show that the conditions on chopping required by Corollary 18 are laxer than those of the analysis for serializability [29], but stricter than those for parallel SI [11]. 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.

\[\text{acct1 = acct1 - 100} \xrightarrow{\text{RW}} \text{acct2 = acct2 + 100} \xrightarrow{\text{WR}} \text{return acct1} \xrightarrow{\text{RW}} \text{return acct2}.
\]
THEOREM 19. For any \( G \), we have \( G \in \text{GraphSI} \setminus \text{GraphSER} \) if and only if \( T_G \models \text{INT} \), \( G \) 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 [18]. 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 \( G_4 \) of the write skew anomaly in Figure 2(d) contains a cycle: \( T_1 \xrightarrow{\text{RW}_4} T_2 \xrightarrow{\text{RW}_5} T_1 \). Furthermore, it is easy to see that all its cycles have two adjacent anti-dependencies, so that \( G_4 \in \text{GraphSI} \setminus \text{GraphSER} \). The dependency graph \( G_4 \) from Figure 4 does not contain any cycles, so that \( G_4 \notin \text{GraphSI} \setminus \text{GraphSER} \). For instance, \( G_4 \in \text{GraphSER} \). Finally, the dependency graph \( G_4 \) of the long fork anomaly in Figure 4 contains a cycle, but without two adjacent anti-dependencies; hence, \( G_4 \notin \text{GraphSI} \setminus \text{GraphSER} \), and in fact, \( G_4 \notin \text{GraphSI} \).

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 \( P \) 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 \( P \). 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 \( P \) produce no histories in \( \text{HistSI} \setminus \text{HistSER} \), 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 [31] 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 \( \text{PREFIX} \), while still requiring visibility to be transitive, a property that we refer to as \( \text{TRANSVIS} \) [10].

**DEFINITION 20.** The sets of executions and histories allowed by parallel SI are:

\[
\text{ExecPSI} = \{ \chi \mid \chi \models \text{INT} \land \text{EXT} \land \text{SESSION} \land \text{TRANSVIS} \land \text{NOConflict} \};
\]

\[
\text{HistPSI} = \{ \mathcal{H} \mid \exists \text{VIS}, \text{CO}, (\mathcal{H}, \text{VIS}, \text{CO}) \in \text{ExecSI} \}.
\]

Note that this specification essentially does not use the commit order \( \text{CO} \) according to \( \text{NOConflict} \), its edges used in \( \text{EXT} \) are uniquely determined by \( \text{VIS} \).

The axiom \( \text{TRANSVIS} \) ensures that transactions ordered by \( \text{VIS} \) are observed by others in this order. However, it allows two transactions unrelated by \( \text{VIS} \) to be observed in different orders; in particular, parallel SI allows the long fork anomaly of Figure 2(c), disallowed by the axiom \( \text{PREFIX} \) in SI.

Following [11, extended version, Lemma 14], we can give a characterisation of parallel SI in terms of dependency graphs.

**THEOREM 21.** Let \( \text{GraphPSI} = \{ G \mid (T_G \models \text{INT}) \land ((\text{SO}_G \cup \text{WR}_G \cup \text{WW}_G)^+ : \text{RW}_G?) \text{ is irreflexive} \} \).

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

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 \( G_4 \) in Figure 2(c). It is easy to see that all its cycles contain at least two anti-dependencies, and therefore \( G_4 \in \text{GraphPSI} \). On the other hand, let \( G_5 \) be the dependency graph in Figure 2(b). The graph \( G_5 \) contains a cycle with exactly one anti-dependency \( (T_1 \xrightarrow{\text{WW}_5} T_2 \xrightarrow{\text{RW}_5} T_1) \), and therefore \( G_5 \notin \text{GraphPSI} \). As a corollary of Theorems 9 and 21, we obtain a dynamic robustness criterion that checks whether a given dependency graph is in \( \text{GraphPSI} \setminus \text{GraphSI} \).

**THEOREM 22.** For any \( G \), we have \( G \in \text{GraphPSI} \setminus \text{GraphSI} \) if and only if \( T_G \models \text{INT} \), \( G \) 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 \( G_4 \) of the long fork anomaly all cycles have at least two anti-dependencies. Furthermore, \( G_4 \) also has a cycle with no adjacent anti-dependencies: \( T_1 \xrightarrow{\text{WR}_4} T_3 \xrightarrow{\text{WR}_5} T_2 \xrightarrow{\text{WR}_6} T_4 \xrightarrow{\text{WR}_7} T_1 \), so that \( G_4 \in \text{GraphPSI} \) \& \( \text{GraphSI} \). The dependency graph \( G_5 \) of the write skew anomaly in Figure 2(d) contains only cycles with at least two adjacent anti-dependencies, so that \( G_4 \notin \text{GraphPSI} \) \& \( \text{GraphSI} \). For instance, \( G_4 \in \text{GraphPSI} \) \& \( \text{GraphSI} \). The dependency graph \( G_5 \) of the lost update anomaly contains a cycle with exactly one anti-dependency, so that \( G_5 \notin \text{GraphPSI} \) \& \( \text{GraphSI} \).

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, 27], 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. [18] 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, 36] and proving the correctness of concurrency-control algorithms [15, 35]. 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 [33].

The constraint on dependency graphs that we use to characterise SI also arose in the work of Lin et al. [23], 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 [25, 35, 37]. There have also been proposals of consistency models for transactional memory that weaken consistency guarantees in a way similar to chopping [4, 19, 34]. Our chopping analysis enables bringing these benefits to transactional systems providing SI. We have previously proposed a chopping analysis for parallel SI [11], 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
APPENDIX

A. ADDITIONAL PROOFS

A.1 Proof of Proposition 7

We prove a more general result, from which Proposition 7 follows immediately.

**Proposition 23.** Let \( \mathcal{X} = \langle T, SO, VIS, CO \rangle \) be an execution such that \( \mathcal{X} \models EXT \). Then \( \mathcal{G} = \langle T, SO, WR_X, WW_X, RW_X \rangle \) is a dependency graph.

**Proof.** It suffices to show that \( \mathcal{G} \) satisfies all the constraints imposed by Definition 6:

\[ \forall T, S \in T. \forall x. T \xrightarrow{WR_X(x)} S \implies \exists n. T \neq S \land T \vdash write(x, n) \land S \vdash read(x, n). \]

Let \( T, S, x \) be such that \( T \xrightarrow{WR_X(x)} S \). By Definition 5, \( S \vdash read(x, n) \) for some \( n \in \mathbb{N} \) and \( T = \max_{CO}(VIS^{-1}(S) \cap WriteTx_x) \). Then \( T \neq S \). Since \( \mathcal{X} \models EXT \), we have \( T \vdash write(x, n) \).

\[ \forall S \in T. \forall x. S \vdash read(x, \_ \_ \_ \_ \_ \_ ) \implies \exists T. T \xrightarrow{WR_X(x)} S. \]

Suppose that \( S \vdash read(x, \_ \_ \_ \_ \_ \_ ) \). Since \( \mathcal{X} \models EXT \), the set \( VIS^{-1}(S) \cap WriteTx_x \) is non-empty, so that \( T = \max_{CO}(VIS^{-1}(S) \cap WriteTx_x) \) is defined. By Definition 5, \( T \xrightarrow{WR_X(x)} S \).

\[ \forall T, T'. S \in T. \forall x. (T \xrightarrow{WR_X(x)} S \land T' \xrightarrow{WR_X(x)} S) \implies T = T'. \]

This holds because by Definition 5 we have \( T = \max_{CO}(VIS^{-1}(S) \cap WriteTx_x) = T' \).

- for any \( x \), \( WW_G(x) \) is a total order over \( WriteTx_x \). Fix an object \( x \) and recall that \( T \xrightarrow{WW_G(x)} S \) if and only if \( T, S \in WriteTx_x \) and \( T \xrightarrow{CO} S \). Since \( CO \) is transitive and irreflexive, so is \( WW_G(x) \). Since \( CO \) is total over \( T \times T \), so is \( WW_G(x) \) over \( T \cap WriteTx_x \).

\[ T \xrightarrow{WR_X(x)} S \iff T \neq S \land \exists T'. T \xrightarrow{WR_X(x)} T \land T' \xrightarrow{WR_X(x)} S. \]

This follows directly from Definition 5.

\[ \square \]

A.2 Proof of Proposition 14

Fix \( \mathcal{X} = \langle T, SO, VIS, CO \rangle \in ExecSl \) and \( T, S \in T \). Let graph(\( \mathcal{X} \)) = \( \langle T, SO, WR, WW, RW \rangle \). We illustrate the proof in Figure 7.

\[ \text{“} \rightarrow \text{“}. \]

Assume \( S \xrightarrow{RW} T \). Then for some \( T' \in T \) we have \( T' \xrightarrow{WR_X(x)} S \) and \( T' \xrightarrow{WW(x)} T \). This implies \( S \neq T, S \vdash read(x, \_ \_ \_ \_ \_ \_ ) \) and \( T \vdash write(x, \_ \_ \_ \_ \_ \_ ) \). Assume that \( T \xrightarrow{VIS} S \). Then \( T \in VIS^{-1}(S) \cap WriteTx_x \). Since \( T \xrightarrow{WW(x)} T \), we have \( T \xrightarrow{CO} T \). But then \( T' \neq \max_{CO}(VIS^{-1}(S) \cap WriteTx_x) \), contradicting \( T \xrightarrow{WR_X(x)} S \). Hence, we cannot have \( T \xrightarrow{VIS} S \).

\[ \text{“} \leftarrow \text{“}. \]

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

\[ \square \]

A.3 Proof of Lemma 15

We first prove that the relations in the statement of the lemma are indeed a solution to the system of inequalities in Figure 3:

1.

\[ SO \cup WR \cup WW = (SO \cup WR \cup WW) \]

\[ \subseteq ((SO \cup WR \cup WW) \cdot RW?) \cup R) \]

\[ = VIS \]

2.

\[ CO \cdot VIS = (((SO \cup WR \cup WW) \cdot RW?) \cup R)^+ \]

\[ = (((SO \cup WR \cup WW) \cdot RW?) \cup R)^+ \cdot (SO \cup WR \cup WW) \]

\[ \subseteq (((SO \cup WR \cup WW) \cdot RW?) \cup R)^+ \cdot (SO \cup WR \cup WW) \]

\[ = VIS \]

3.

\[ VIS = (((SO \cup WR \cup WW) \cdot RW?) \cup R)^+ \cdot (SO \cup WR \cup WW) \]

\[ \subseteq (((SO \cup WR \cup WW) \cdot RW?) \cup R)^+ \cdot (SO \cup WR \cup WW) \cdot RW? \]

\[ \subseteq (((SO \cup WR \cup WW) \cdot RW?) \cup R)^+ \cdot (SO \cup WR \cup WW) \cdot RW? \]

\[ = VIS \]

\[ = CO \]
Since $R \subseteq V$, this implies
$((\text{SO} \cup \text{WR} \cup \text{WW}) \cdot \text{RW}) \cup R \subseteq \text{CO}',$
and by (S4) we get
$((\text{SO} \cup \text{WR} \cup \text{WW}) \cdot \text{RW}) \cup R \subseteq \text{CO}'.$

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

To prove that $\text{VIS} \subseteq \text{VIS}'$, we rewrite $\text{VIS}$ as
$\text{VIS} = (\text{SO} \cup \text{WR} \cup \text{WW}) \cup (((\text{SO} \cup \text{WR} \cup \text{WW}) \cdot \text{RW}) \cup R)^+ \cdot (\text{SO} \cup \text{WR} \cup \text{WW})$,
and we prove that both parameters of the union are included in $\text{VIS}'. This establishes $\text{VIS} \subseteq \text{VIS}'$ because, according to (S2) and (S3), $\text{VIS}'$ is transitive. We have already proved (3). We also have
$(((\text{SO} \cup \text{WR} \cup \text{WW}) \cdot \text{RW}) \cup R)^+ \cdot (\text{SO} \cup \text{WR} \cup \text{WW}) = \text{CO} \cdot (\text{SO} \cup \text{WR} \cup \text{WW})$.
Since $\text{CO} \subseteq \text{CO}'$ and $\text{SO} \cup \text{WR} \cup \text{WW} \subseteq \text{VIS}'$, by (S2) we get
$\text{CO} \cdot (\text{SO} \cup \text{WR} \cup \text{WW}) \subseteq \text{CO}' \cdot \text{VIS}' \subseteq \text{VIS}'$,
as required. □

A.4 Proof of Lemma 17

Let $G \in \text{GraphI}$ and suppose that $T \xrightarrow{\text{RW}_{\text{null}}(x)} S \xrightarrow{\text{SO}} T \yrightarrow{\text{WR}} S$, for some $T, S \in T_G$ and $x \in \text{Obj}$. By definition, $T \neq S$, and there exists $V \in T_G$ such that $V \xrightarrow{\text{RW}_{\text{null}}(x)} T \xrightarrow{\text{WR}} V \xrightarrow{\text{WW}_{\text{null}}(x)} S$. That is, there exist $T', V', \approx_G V', V'' \approx_G V$ and $S'' \approx_G S$ such that $V \xrightarrow{\text{RW}_{\text{null}}(x)} T'$, $V' \xrightarrow{\text{WW}_{\text{null}}(x)} S''$. We show that $T' \xrightarrow{\text{RW}_{\text{null}}(x)} S''$, so that $T \approx_G ; \approx_G S$.

First note that from $V \xrightarrow{\text{WR}_{\text{null}}(x)} T'$ and $V' \xrightarrow{\text{WW}_{\text{null}}(x)} S''$ we can infer $V' \vdash \text{write}(x, \_)$, $V'' \vdash \text{write}(x, \_)$ and $S'' \vdash \text{write}(x, \_)$.
Since $V'' \approx_G V'$ and $V''$, one of the following must be true: $V'' = V''$, $V' \approx_G V''$ or $V'' \approx_G V'$.
A.5 Proof of Theorem 16

We use \( \gamma, \gamma', \ldots \) to range over cycles in a dependency graph \( G \), i.e., paths of the form \( T_1 \xrightarrow{R_1} T_2 \xrightarrow{R_2} \cdots \xrightarrow{R_{n-1}} T_n \) such that \( T_1 = T_n \) and for any \( i = 1, \ldots, n \) we have \( T_i \in T_G \) and \( \gamma_i \in \{SO_G, WR_G, WW_G\} \). For a given cycle \( \gamma = T_1 \xrightarrow{R_1} T_2 \xrightarrow{R_2} \cdots \xrightarrow{R_{n-1}} T_n \), we let \( \text{rep}(\gamma) = |\{T_i \mid \exists j, 1 \leq i < j \leq n \land i \neq j \land T_i = T_j\}| \) be the number of repeated vertices in it. Note that \( \gamma \) has no repeated vertices if and only if \( \text{rep}(\gamma) = 0 \). We call such a cycle simple.

**Lemma 24.** Let \( G \) be a dependency graph such that the relation \((SO_G \cup WR_G \cup WW_G) ; RW_G?\) contains a cycle. Then this relation contains a simple cycle.

**Proof.** Let \( \gamma \) be a cycle in \((SO_G \cup WR_G \cup WW_G) ; RW_G?\), i.e., a cycle without adjacent \( RW_G \) edges. If \( \text{rep}(\gamma) = 0 \), then there is nothing to prove, so let us assume that \( \text{rep}(\gamma) > 0 \). Below we show how to extract a sub-cycle \( \gamma' \) of \( \gamma \) such that \( \text{rep}(\gamma') < \text{rep}(\gamma) \) and \( \gamma' \) has no adjacent \( RW_G \) edges. By applying this procedure repeatedly, we obtain a cycle \( \gamma'' \) with no adjacent \( RW_G \) edges and such that \( \text{rep}(\gamma'') = 0 \), as required.
For some $T, S \in T_G$ and $\{R_i\}_{i=1}^4 \subseteq \{SO_G, WR_G, WW_G, RW_G\}$. A graphical representation of $\gamma$ is given in Figure 9.

From $\gamma$ we can derive the cycles

$$\gamma_1 = S \rightarrow \cdots \rightarrow R_1 \rightarrow T \rightarrow R_2 \rightarrow \cdots \rightarrow S$$

which are contained inside the dashed boxes in the picture to the right.

Since $\gamma$ contains no adjacent $RW_G$ edges, we can have two adjacent $RW_G$ edges in $\gamma_1$ only if $R_1 = RW_G$ and $R_2 = RW_G$; similarly, we have two adjacent $RW_G$ edges in $\gamma_2$ only if $R_2 = RW_G$ and $R_3 = RW_G$. Therefore, if either $R_1 \neq RW_G$ or $R_2 \neq RW_G$, then we know that $\gamma_1$ has no adjacent $RW_G$ edges, and we choose $\gamma' = \gamma_1$. Otherwise $R_3 = RW_G$; since $\gamma$ contains no adjacent $RW_G$ edges, it follows that $R_2 \neq RW_G$; therefore $\gamma_2$ contains no adjacent $RW_G$ edges, and we choose $\gamma' = \gamma_2$.

**Proposition 25.** Let $G$ be a dependency graph. For any $T \in T_G$:

1. $T \vdash read(x, n)$ if and only if $\min_{SO_G} \{S \mid S \approx_G T \land T \vdash \_\approx \_\} \vdash read(x, n)$;
2. $T \vdash write(x, n)$ if and only if $\max_{SO_G} \{S \mid S \approx_G T \land T \vdash write(x, n)\} \vdash write(x, n)$.

**Proof.** We only prove Statement i; the proof for Statement ii is analogous. To maintain the notation easy, we let $[T]_G = (E_T, \text{po}_T)$.

Suppose that $[T]_G \vdash read(x, n)$; by definition, the event $e_{rd} = \min_{\text{po}_T} \{e \in E_T \mid \text{op}(e) = \_\approx \_\}$ exists, and $\text{op}(e_{rd}) = read(x, n)$. Recall that $E_T = (\bigcup \{E_S \mid S \approx_G T\})$, from which it follows that $e_{rd} \in E_{Td}$ for some $T_{td} \approx_{td} T$. Also, $\text{po}_T = \{(e, f) \mid \exists S, S \approx_G T \land (e, f) \in E_{rd} \land e \text{ po}_T f\} \cup (\exists S', e \in E_{rd} \land f \in E_{S'} \land S \xrightarrow{SO_G} S')$. A first consequence of this fact is that, for any event $f \in E_{rd}$ such that $\text{op}(f) = \_\approx \_\wedge e_{rd} \text{ po}_T f$, hence $T_{td} \vdash read(x, n)$. A second consequence is that, for any transaction $S \approx_G T_{td}$ with $S \neq T_{td}$, and for any event $f \in E_{S}$ such that $\text{op}(f) = \_\approx \_\wedge e_{rd} \text{ po}_T f$, then $T_{td} \vdash read(x, n)$; alternatively, we can write that if $S \approx_G T_{td}, S \neq T_{td}$, and either $S \vdash read(x, n)$ or $S \vdash write(x, n)$, then $T_{td} \vdash \min_{SO_G} \{S \mid S \approx_G T \land (S \vdash \_\approx \_\lor S \vdash \_\approx \_)\}$.

Next, suppose that $T_{td} = \min_{SO_G} \{S \mid S \approx_G T \land S \vdash \_\approx \_\}$ is defined, and $T_{td} \vdash \_\approx \_$. By definition, there exists an event $e_{rd} \in E_{Td}$ such that $\text{op}(e_{rd}) = read(x, n)$, and for any $f \in E_{Td}$ such that $\text{op}(f) = \_\approx \_\wedge e_{rd} \text{ po}_T f$, we have that $e_{rd} \vdash \_\approx \_$. Note that $e_{rd} \in E_{Td}$. We prove that, for any event $f \in E_{Td}$, if $\text{op}(f) = \_\approx \_\wedge e_{rd} \text{ po}_T f$, from which it follows that $[T]_G \vdash read(x, n)$. First note that we have already shown that if $\text{op}(f) = \_\approx \_$ and $f \in E_{Td}$, then $e_{rd} \vdash \_\approx \_$. We have proved that if $\text{op}(f) = \_\approx \_\wedge e_{rd} \text{ po}_T f$, from which $e_{rd} \vdash \_\approx \_\wedge e_{rd} \text{ po}_T f$, follows. Suppose then that $f \in E_{S}$ for some $S \neq T_{td}, S \approx_G T_{td}$ if $\text{op}(f) = \_\approx \_$ then either $S \vdash \_\approx \_$ (if there exists no event $f' \in E_{S}$ such that $\text{op}(f') = \_\approx \_\lor S \vdash \_\approx \_$); if $\text{op}(f) = \_\approx \_$, then $S \vdash \_\approx \_$. We have proved that if $\text{op}(f) = \_\approx \_$, then $S \vdash \_\approx \_\lor S \vdash \_\approx \_$, so that it cannot be $S \xrightarrow{SO_G} T_{td}$; therefore, it has to be the case that $T_{td} \vdash \_\approx \_\wedge e_{rd} \text{ po}_T f$, from which $e_{rd} \vdash \_\approx \_\wedge e_{rd} \text{ po}_T f$, follows. Without loss of generality, let $G \in \text{GraphSI}$ such that $\text{DCG}(G)$ contains no critical cycles. Then $\text{splice}(G)$ is a dependency graph.

**Proof.** We prove that, if the chopping graph of $G$ contains no critical cycles, then splice($G$) satisfies all the constraints of Definition 6. 

![Figure 9: A cycle with a repeated transaction T](image-url)
Figure 10: Graphical representation of different cases in the proof of Lemma 26.
Consider \( T, S \in T_{spliceg} \) such that \( T \xrightarrow{\text{WR}_{spliceg}(x)} S \). By definition \( T \neq S \) and there exist two transactions \( T', S' \in T_G \) such that \( T \approx_G T' \xrightarrow{\text{WR}_G(x)} S' \approx_G S \). Hence, for some \( n \) we have \( T' \vdash \text{write}(x, n) \) and \( S' \vdash \text{read}(x, n) \). We now prove that (i) \( T \vdash \text{write}(x, n) \) and (ii) \( S \vdash \text{read}(x, n) \).

(i) Consider an arbitrary transaction \( T'' \in T \) such that \( T'' \vdash \text{write}(x, n) \) and \( T'' \approx_G T' \) (Figure 10(a)). We show that it cannot be the case that \( T' \xrightarrow{\text{SO}_G} T'' \). Then \( T'' = \max_{SO_G} \{ S \mid S \approx_G T \land S \vdash \text{write}(x, n) \} \), and by Proposition 25(ii) it follows that \( T' \vdash \text{write}(x, n) \), as required.

Assume \( T' \xrightarrow{\text{SO}_G} T'' \); then \( T'' \not\approx T' \). Since \( T', T'' \vdash \text{write}(x, n) \), either \( T' \xrightarrow{\text{WW}_G(x)} T'' \) or \( T'' \xrightarrow{\text{WW}_G(x)} T' \). However, the latter case would lead to the cycle \( T'' \xrightarrow{\text{SO}_G} T' \xrightarrow{\text{SO}_G} T'' \), which cannot exist because \( G \in \text{GraphSI} \). Therefore \( T' \xrightarrow{\text{WW}_G(x)} T'' \). Together with \( T'' \xrightarrow{\text{WR}_G(x)} S' \), this yields the anti-dependency \( S' \xrightarrow{\text{RW}_G(x)} T'' \). Since \( S' \approx_G S \not\approx_G T \approx_G T'' \), this implies \( S' \vdash \text{read}(x, n) \) in \( DCG(G) \), which is critical. This contradicts the assumption of the lemma.

(ii) We show that, for any transaction \( S'' \approx_G S' \) such that \( S'' \xrightarrow{\text{SO}_G} S' \) we have \( \neg S'' \vdash \text{write}(x, n) \), and if \( S'' \vdash \text{read}(x, m) \), then \( m = n \). As a consequence, \( \min_{SO_G} \{ V \mid V \approx_G S \land V \vdash \text{write}(x, n) \} \vdash \text{read}(x, n) \), and hence, by Proposition 25(i) we have \( S \vdash \text{read}(x, n) \).

Let \( S'' \) be a transaction such that \( S'' \xrightarrow{\text{SO}_G} S' \); we prove that \( \neg (S'' \vdash \text{write}(x, n)) \) by contradiction. Assume \( S'' \vdash \text{write}(x, n) \). Then by the definition of \( \text{WW}_G(x) \), either \( T'' \xrightarrow{\text{WW}_G(x)} S' \) or \( S'' \xrightarrow{\text{WW}_G(x)} T'' \); the case \( S'' \approx T'' \) is ruled out because \( S'' \approx_G S \not\approx_G T \approx_G T'' \).

We cannot have \( T'' \xrightarrow{\text{WW}_G(x)} S'' \) (Figure 10(b)), since together with \( T'' \xrightarrow{\text{WR}_G(x)} S' \), this would imply the anti-dependency \( S' \xrightarrow{\text{RW}_G(x)} S'' \). But then we have a cycle \( S' \xrightarrow{\text{RW}_G(x)} S'' \xrightarrow{\text{SO}_G} S', \) contradicting \( G \in \text{GraphSI} \). We cannot have \( S'' \xrightarrow{\text{WW}_G(x)} T' \) either (Figure 10(c)); in this case the chopping graph of \( G \) contains the critical cycle \( S'' \xrightarrow{\text{WW}_G(x)} T'' \xrightarrow{\text{WR}_G(x)} S' \xrightarrow{\text{SO}_G} S'' \). We have thus established \( \neg (S'' \vdash \text{write}(x, n)) \).

Suppose now that \( S'' \xrightarrow{\text{SO}_G} S' \) and \( S'' \vdash \text{read}(x, m) \) for some \( m \). Then there exists a transaction \( V' \in T_G \) such that \( V' \xrightarrow{\text{WR}_G(x)} S'' \) and \( V' \vdash \text{write}(x, m) \). Since \( T'' \vdash \text{write}(x, n) \), we have \( V' = T'' \), \( V \xrightarrow{\text{WW}_G(x)} T'' \) or \( T'' \xrightarrow{\text{WW}_G(x)} V \). We show that the latter two cases are impossible, so that \( T'' = V' \) and, hence, \( m = n \), as required. If \( T'' \xrightarrow{\text{WW}_G(x)} V \), then we have the anti-dependency \( S' \xrightarrow{\text{RW}_G(x)} V' \) (Figure 10(d)). Then the graph \( G \) contains the cycle \( S' \xrightarrow{\text{RW}_G(x)} V \xrightarrow{\text{WR}_G(x)} S'' \xrightarrow{\text{SO}_G} S' \), which contradicts \( G \in \text{GraphSI} \).

If \( V' \xrightarrow{\text{WW}_G(x)} T'' \), then we have the anti-dependency \( S' \xrightarrow{\text{RW}_G(x)} T'' \) (Figure 10(e)). In this case \( DCG(G) \) contains the critical cycle \( S'' \xrightarrow{\text{RW}_G(x)} T'' \xrightarrow{\text{WR}_G(x)} S' \xrightarrow{\text{SO}_G} S'' \), and again we obtain a contradiction.

Let \( S \), \( T \), \( V \) be transactions such that \( T \vdash \text{read}(x, ...) \). By Proposition 25(i), there exists a transaction \( S' \approx_G S \) such that \( T \vdash \text{read}(x, ...) \).

Since \( G \) is a dependency graph, there exists a transaction \( T \in T_G \) such that \( T \xrightarrow{\text{WR}_G(x)} S' \). Then \( T \not\approx S' \). We cannot have \( T \xrightarrow{\text{SO}_G} S' \), because \( T \vdash \text{write}(x, ...) \) and \( S' = \min_{SO_G} \{ S \mid S \approx_G S' \land S \vdash \text{write}(x, ...) \} \) by Proposition 25(i). Finally, we cannot have \( T' \xrightarrow{\text{SO}_G} T \), because this would contradict the hypothesis \( G \in \text{GraphSI} \) due to the cycle \( S' \xrightarrow{\text{SO}_G} T \xrightarrow{\text{WR}_G(x)} S' \). As a consequence, it cannot be \( T \approx_G S' \). Then \( T \not\approx_G S' \).

Let \( S, T, V \in T_{spliceg} \) be such that \( T \xrightarrow{\text{WR}_{spliceg}(x)} S \) and \( V \xrightarrow{\text{WR}_{spliceg}(x)} S \). Then \( S \not\approx_G T \), \( S \not\approx_G V \) and there exist transactions \( S', S'', T', V' \) such that \( S' \approx_S S \approx S'' \), \( T' \xrightarrow{\text{WR}_G(x)} S' \), \( V' \xrightarrow{\text{WR}_G(x)} S'' \). Note that if \( S' = S'' \), then \( T' = V' \), because \( G \) is a dependency graph; hence \( T \not\approx_G S' \), and there is nothing left to prove.

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

We prove that \( \text{WW} \) is transitive, irreflexive, and whenever \( T \vdash \text{write}(x, ...) \) and is total over \( \text{WriteTx} \).
**Lemma 27.** Let $G$ be a dependency graph whose chopping graph has no critical cycle. Then $T_{\text{splice}(G)} \models \text{INT}.$

**Proof.** Proceeding by contradiction, let us assume that INT is violated. Then there exist: $T \in T_G; e \in E_T$ such that $\text{op}(e) = \text{read}(x, n)$ for some $x \in \text{Obj}, n \in \mathbb{N};$ and, letting $\text{po}_{\text{op}} = \text{po}_{\text{read}},$

$$f = \max \{\text{po}_{\text{op}}(e') | e' \preceq (x, n) \land e' \preceq f\}$$

(4)
such that $\text{po}_{\text{op}}(f) = (x, m)$ for some $m \neq n.$ Since $T_G \models \text{INT},$ we cannot have $f \in E_T.$ Therefore, there exists a transaction $T' = \text{splicex}(x, n)$ such that $f \in E_{T'}.$ We now make a case split on whether $T' \models \text{write}(x, n)$.

1. $T' \models \text{write}(x, n).$ Then there exists an event $g \in E_{T'}$ such that $\text{op}(g) = \text{write}(x, n).$ Without loss of generality, let $g$ be the last write to object $x$ in $E_{T'}.$

If $f \preceq g,$ then $f \preceq g \preceq g \preceq e,$ contradicting (4). Thus, either $g = f$ or $g \preceq f.$ In both cases, we show that $\text{op}(g) = \text{write}(x, m).$ If $g = f,$ we have $(x, n) = \text{op}(f) = \text{op}(g) = \text{op}(e), \text{so that } \text{op}(g) = \text{write}(x, m).$ If $g \preceq f,$ then $\text{op}(f) = \text{read}(x, m),$ since $g$ is the last write to $x$ in $T'.$ Also, for any other event event $h$ such that $\text{op}(h) = (x, n)$ and $g \preceq h \preceq f,$ we have $\text{op}(h) = \text{read}(x, n).$ Then because $T_G \models \text{INT}$ and $\text{op}(f) = \text{read}(x, m),$ we must have $\text{op}(h) = \text{read}(x, m).$ But then $T_G \models \text{INT}$ again ensures $\text{op}(g) = \text{write}(x, m).$

We have proved that $T' \models \text{write}(x, m).$ By hypothesis $T \models \text{read}(x, n)$ for some $n \neq m,$ so that there exists a transaction $S \neq T'$ such that $S \models \text{write}(x, n)$ and $S \overset{\text{WW}(x)}{\rightarrow} T.$
Next we prove that it cannot be either $S \xrightarrow{SO_G} T \xrightarrow{T'} \xrightarrow{SO_G} T$, nor $T' \xrightarrow{SO_G} S \xrightarrow{SO_G} T$, nor $T' \xrightarrow{SO_G} T \xrightarrow{SO_G} S$. Since we have already proved that $S \neq T'$ and $T' \xrightarrow{SO_G} T$, these imply $T' \not\ni_G S$.

- If $S \xrightarrow{SO_G} T \xrightarrow{T'} \xrightarrow{SO_G} T$, then since $S, T' \ni_G \text{write}(x,\_)$ and $S \neq T'$, it has to be either $S \xrightarrow{WW_G(x)} T'$ or $T' \xrightarrow{WW_G(x)} S$. However, the last case is impossible because it would lead to a cycle $S \xrightarrow{SO_G} T \xrightarrow{WW_G(x)} S$, contradicting $G \in \text{GraphS}$. Therefore, $S \xrightarrow{WW_G(x)} T'$.

- If $T' \xrightarrow{SO_G} S \xrightarrow{SO_G} T$, since $S \ni \text{write}(x,\_)$, this implies that there exists an event $h \in E_G$ such that $\text{op}(h) = \text{write}(x,\_)$. We can show that $f \xrightarrow{\text{point}_T} h \xrightarrow{\text{point}_T} e$, contradicting the hypothesis that $f = \text{max}_{\text{point}_T}(f | \text{op}(f) = _\gamma(x,\_) \land f \xrightarrow{T} e)$.

- If $T' \xrightarrow{SO_G} T \xrightarrow{SO_G} S$, then we have the cycle $T \xrightarrow{SO_G} S \xrightarrow{WW_G(x)} T$, which is not allowed because we are assuming that $G \in \text{GraphS}$.

We have proved that $T' \not\ni_G S$. Next, we observe that since $T' \ni \text{write}(x, m)$, $S \ni \text{write}(x, n)$ and $T' \neq S$, we must have either $T' \xrightarrow{WW_G(x)} S$ or $S \xrightarrow{WW_G(x)} T'$. We show that both of these cases lead to a contradiction. If $S \xrightarrow{WW_G(x)} T'$, then since $S \xrightarrow{WW_G(x)} T$, we get $T \xrightarrow{WW_G(x)} T'$, causing the cycle $T \xrightarrow{WW_G(x)} T' \xrightarrow{SO_G} T'$; this contradicts $G \in \text{GraphS}$. On the other hand, if $T' \xrightarrow{WW_G(x)} S$, then we have a critical cycle $T' \xrightarrow{WW_G(x)} S \xrightarrow{WW_G(x)} T \xrightarrow{SO_G} T'$ in $\text{DCG}(G)$, contradicting the hypothesis of the lemma.

2. $(T' \ni \text{write}(x,\_))$. In this case there does not exist an event $g \in E_{\gamma'}$ such that $\text{op}(g) = \text{write}(x,\_)$. Using the fact that $T_\gamma = \text{INT}$, we can easily show that for any $g \in E_{\gamma'}$ such that $\text{op}(g) = \text{read}(x,\_)$, we have $\text{op}(g) = \text{read}(x,\_)$. Then $T' \ni \text{read}(x,\_)$.

Since $G$ is a dependency graph, there exist two transactions $S, V$ such that $S \xrightarrow{WR_G(x)} T$ and $V \xrightarrow{WR_G(x)} T'$. We have $S \ni \text{write}(x, n)$ and $V \ni \text{write}(x, m)$, so that $S \neq V$. Then either $S \xrightarrow{WW_G(x)} V$ or $V \xrightarrow{WW_G(x)} S$. We show that neither of these cases is possible.

If $S \xrightarrow{WW_G(x)} V$, then $T \xrightarrow{WW_G(x)} V$. This causes a cycle $T \xrightarrow{WW_G(x)} V \xrightarrow{WR_G(x)} T' \xrightarrow{SO_G} T$, contradicting $G \in \text{GraphS}$.

On the other hand, if $V \xrightarrow{WW_G(x)} S$, then observe that $T' \neq S$, since $(T' \ni \text{write}(x,\_))$ and $S \ni \text{write}(x,\_)$. Therefore $T' \xrightarrow{WW_G(x)} S$. We can show that $S \not\ni_G T$ in a way similar to the one above, proving that neither of the cases $S \xrightarrow{SO_G} T$ or $T' \xrightarrow{SO_G} S$ or $T' \xrightarrow{SO_G} T$ or $T' \xrightarrow{SO_G} S$ or $T' \xrightarrow{SO_G} T$ or $T' \xrightarrow{SO_G} S$ in $\text{DCG}(G)$, contradicting the assumptions of the lemma. □

**Proof of Theorem 16.** Let $G \in \text{GraphS}$ be a dependency graph such that $\text{DCG}(G)$ contains no critical cycle. We prove that $\text{splice}(G) \in \text{GraphS}$. First, Lemmas 26 and 27 ensure that $\text{splice}(G)$ is indeed a dependency graph and $\text{GraphS}(\text{splice}(G)) = \text{INT}$. Since $\text{SO} \text{splice}(G) = 0$, by Theorem 10 it remains to prove that the relation $((\text{WR} \text{splice}(G) \cup \text{WW} \text{splice}(G)) : \text{RW} \text{splice}(G), \gamma)$ is acyclic. The proof goes by contradiction: we assume that this relation contains a cycle and exhibit a critical cycle in $\text{DCG}(G)$. Let

\[ \gamma = T_0 \xrightarrow{c_0} \ldots \xrightarrow{c_{n-1}} T_n \xrightarrow{c_n} (n \geq 1) \]

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

\[ T_0, \ldots, T_n \in T_\gamma, \quad c_0, \ldots, c_{n-1} \in \{\text{WR} \text{splice}(G), \text{WW} \text{splice}(G), \text{RW} \text{splice}(G)\} \]

(the letter $C$ stands for conflict). $T_n = T_0$ (in particular, $T_n \approx_G T_0$), and there is no index $i = 0..(n-1)$ such that $C_i = \text{RW} \text{splice}(G)$ and $C_i \mod n = \text{RW} \text{splice}(G)$. By Lemma 24, we can assume that $\gamma$ is simple. Thus, for any $i, j = 0..(n-1)$ we have $T_{i+j} \approx_G T_j$ (equivalently, $T_i \approx_G T_j$) only if $i = j$.

By applying Lemma 17, we can convert $\gamma$ into the following path:

\[ T_0 \approx_G T_0' \xrightarrow{c_0'} T_1' \approx_G T_1'' \xrightarrow{c_1''} \ldots \xrightarrow{c_{n-1}'} T_n' \approx_G T_n'' \]

(5)

where for any $i = 0..n$, $T_i' \approx_G T_i$ (note that because $T_n \approx_G T_0$, this implies $T_n'' \approx_G T_0'$), and for any $i = 0..(n-1)$, $C_i'$ is the relation in $\gamma$ corresponding to the relation $C_i$ in $\text{splice}(G)$ (e.g., if $C_i = \text{RW} \text{splice}(G)$, then $C_i' = (\text{WR} \text{splice}(G) \mod n = \text{RW} \text{splice}(G))$). We also know that

\[-\exists i = 0..(n-1). C_i' = (\text{RW} \text{splice}(G) \mod n = \text{RW} \text{splice}(G)). \]

(6)
Since the cycle $\gamma$ is simple, the only possibility for vertices to be repeated on the path (5) is when they are adjacent: $T'_n = T''_n$ for some $i = 0..(n - 1)$. Recall that whenever $T \not\approx S$, for some transaction $T, S \in T_D$, then one of the following holds: $T = S$, $T \xrightarrow{SO_i} S$ or $T \xrightarrow{SO_i^{-1}} S$. Also, we know that $T'_n \approx T_n \approx T_0 \approx T''_n$. Therefore, we can rewrite the path (5) as follows:

$$T'_n \xrightarrow{S_1} T''_n \xrightarrow{c_0} T'_1 \xrightarrow{S_1} T''_1 \xrightarrow{c_0} \ldots \xrightarrow{S_{n-1}} T''_{n-1} \xrightarrow{c_0} T'_n,$$

(7)

where $S_0, \ldots, S_{n-1} \in \{SO_i, SO_i^{-1}\}$ (the letter $S$ stands for siblings). In this cycle repeated vertices are always adjacent and connected by a $SO_i$?-edge. By removing such edges from the cycle we obtain a simple cycle, where all the occurrences of $SO_i$?-edges are actually $SO_i$-edges; this is a cycle in $DCG(G)$. Due to (6), 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 $DCG(G)$, it remains to show that there exists an index $i = 0..(n - 1)$ such that $S_i = SO_i^{-1}$. This holds because, if we had $S_i = SO_i$ for all $i = 0..(n - 1)$, then we would obtain a cycle in $((SO_i \cup WR_i \cup WW_i) \cup RW_i)^+$, contradicting the assumption that $G \in GraphSI$. □

B. ADDITIONAL MATERIAL ON TRANSACTION CHOPPING UNDER SI

In this section we compare our chopping criterion for SI to criteria that have been proposed for other consistency models: serializability (§B.1) and parallel SI (§B.2). We also discuss why we defined splicing over dependency graphs, rather than over executions (§B.3).

B.1 Comparison with Transaction Chopping under Serializability

We now compare our chopping criterion for SI to the one previously proposed for serializability. For clarity, we refer to critical cycles of §5 as SI-critical. The following is an improved version of the chopping criterion by Shasha et al. [29].

**Definition 28.** A cycle in $SCG(P)$ is SER-critical if: (i) it does not contain two occurrences of the same vertex; and (ii) it contains three consecutive edges of the form “conflict, predecessor, conflict”.

**Theorem 29.** The chopping defined by programs $P$ is correct under serializability if $SCG(P)$ contains no SER-critical cycles.

Note that any SI-critical cycle is also SER-critical. As a consequence, any set of programs that is chopped correctly under SI is also chopped correctly under serializability. In particular, the chopping defined by the programs $P_2$ considered in Figure 6 is correct under serializability. On the other hand, the chopping defined by $P_3$ from Figure 5 is incorrect, and in fact $SCG(P_1)$ contains a SER-critical cycle:

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

(8)

The programs $P_3 = \{write1, write2\}$ in Figure 11 define a correct chopping under SI, but not under serializability. Their static chopping graph $SCG(P_3)$ is depicted on the bottom left of the same figure. There is only one cycle that contains no repeated vertices and three consecutive edges of the form “conflict, predecessor, conflict”:

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

(9)

This cycle is not SI-critical, and by Corollary 18, the chopping defined by $P_3$ is correct under SI. However, it is incorrect under serializability. Indeed, consider the execution $H_6 \in GraphSER$ depicted in Figure 11, which can be produced by $P_3$. It is immediate to observe that splice($H_6$) is a write skew: splice($H_6$) \notin HistSER.

B.2 Comparison with Transaction Chopping under Parallel SI

Next, we compare our chopping criterion for SI to the one that we recently proposed for PSI [11].

**Definition 30.** Let $P$ be a transactional application. A cycle in $SCG(P)$ is PSI-critical if: (i) it does not contain two instances of the same vertex; (ii) it contains three consecutive edges of the form “conflict, predecessor, conflict”; and (iii) it contains at most one anti-dependency edge.

**Theorem 31.** The chopping defined by programs $P$ is correct under PSI if $SCG(P)$ contains no PSI-critical cycles.

Note that any cycle that is PSI-critical, is also SI-critical; as a consequence, the sets of programs $P_2$ and $P_3$ considered in this section define a correct chopping under PSI. On the other hand, it is easy to see that $P_1$ cannot be chopped correctly under PSI: the cycle (8) for $SCG(P_1)$ is PSI-critical.

The programs $P_4 = \{write1, write2, read1, read2\}$ in Figure 12 define a correct chopping under PSI, but not SI. The static chopping graph $SCG(P_4)$ is depicted on the bottom left of the same figure. The graph contains exactly one cycle with no repeated vertices, and three consecutive vertices of the form “conflict, predecessor, conflict”:

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

(10)

This cycle is not PSI-critical, so that $P_4$ indeed define a correct chopping under PSI. On the other hand, this cycle is SI-critical and $P_4$ do not define a correct chopping under SI. Indeed, consider the dependency graph $G_7 \in GraphSI$ in Figure 12, which can be produced by $P_4$. Splicing the history $H_{G_7}$ results in a long fork anomaly: splice($H_{G_7}$) \notin HistSI. It follows that $P_4$ does not define a correct chopping under SI.
session write1 { tx { var1 = x }; tx { y = var1 } }
session write2 { tx { var2 = y }; tx { x = var2 } }

Figure 11: Example of a chopping correct under SI, but not under SER.

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); }

Figure 12: Example of a chopping correct under PSI, but not under SI.
B.3 Challenges in Splicing Executions

In §5 we claimed that splicing abstract executions, rather than dependency graphs, is challenging. Here we illustrate why via a simple example. Consider the abstract execution $X$ of Figure 13, which is in ExecSI. A straightforward way to define $\text{splice}(X)$ is by letting

$$\text{T}_{X} \xrightarrow{\text{CO}_{\text{splice}(X)}} \text{S}_{X} \iff \exists T', S'. T \approx_{H_X} T' \xrightarrow{\text{CO}_{X}} S' \approx_{H_X} S,$$

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

$$\text{T}_{X} \xrightarrow{\text{CO}_{\text{splice}(X)}} \text{S}_{X} \xrightarrow{\text{CO}_{\text{splice}(X)}} \text{T}_{X},$$

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

Figure 13: An attempt to splice an execution directly.