Robustness against Consistency Models with Atomic Visibility

Giovanni Bernardi¹ and Alexey Gotsman²

1 IMDEA Software Institute, Madrid, Spain
2 IMDEA Software Institute, Madrid, Spain

Abstract
To achieve scalability, modern Internet services often rely on distributed databases with consistency models for transactions weaker than serializability. At present, application programmers often lack techniques to ensure that the weakness of these consistency models does not violate application correctness. We present criteria to check whether applications that rely on a database providing only weak consistency are robust, i.e., behave as if they used a database providing serializability. When this is the case, the application programmer can reap the scalability benefits of weak consistency while being able to easily check the desired correctness properties. Our results handle systematically and uniformly several recently proposed weak consistency models, as well as a mechanism for strengthening consistency in parts of an application.

1998 ACM Subject Classification C.2.4 Distributed Systems

Keywords and phrases Robustness, Replication, Consistency models, Transactions

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2016.7

1 Introduction
To achieve scalability and availability, modern Internet services often rely on large-scale databases that replicate and partition data across a large number of nodes and/or a wide geographical span (e.g., [14, 20, 26, 3, 6, 4, 5, 11, 22, 27, 10]). The database clients invoke transactions on the data at any of the nodes, and the nodes communicate changes to each other using message passing. Ideally, we want this distributed system to provide strong guarantees about transaction processing, such as serializability [8]: the results of concurrently executing a set of transactions could be obtained if these transactions were executed serially in some order. Serializability is useful because it allows an application programmer to easily establish desired correctness properties. For example, to check that the transactions of an application preserve a given data integrity constraint, the programmer only needs to check that every transaction does so when executed in isolation, without worrying about concurrency. Unfortunately, achieving serializability requires excessive synchronisation among database nodes, which slows down the database and even makes it unavailable if network connections between replicas fail [17, 1]. For this reason, nowadays large-scale databases often provide weak consistency guarantees, which allow non-serializable behaviours, called anomalies.

As a motivating example, consider a toy on-line auction application with transactions defined by the transactional programs in Figure 1. The program RegUser creates a new user account. It manipulates the table USERS, whose rows contain a primary key (uid) and a nickname. An invocation of RegUser uname inserts a new row in USERS only if the nickname uname does not appear in USERS, to ensure that nicknames are unique. The program ViewUsers can be used to view all the users. Some databases [22, 26, 3] may allow
executions of \textit{RegUser} and \textit{ViewUsers} such as the one sketched in Figure 2(c). There are two invocations of \textit{RegUser} that generate the transactions $T_1$ and $T_2$; these two rows of USERS, denoted by $x$ and $y$, to register the users Alice and Bob. The program \textit{ViewUsers}() then is invoked twice; the invocation in $T_3$ sees Alice but not Bob, while the invocation in $T_4$ sees Bob but not Alice. This result, called a long fork anomaly, cannot be obtained by executing the four transactions in any sequence and, hence, is not serializable.

The past few years have seen a number of proposals of new transactional consistency models for modern large-scale databases \[22, 11, 26, 3, 5\], differing in how much they weaken consistency, by exposing such anomalies, in exchange for improved performance. Unfortunately, application programmers often lack techniques to ensure that the weakness of these consistency models does not violate application correctness. This situation hinders the adoption of the novel consistency models by mainstream database developers and application programmers.

One way to address this problem is using the notion of application robustness \[16, 15, 9\]. An application is robust against a particular weak consistency model if it behaves the same whether using a database providing this model or serializability. If an application is robust against a given weak consistency model, then programmers can reap the performance benefits of using weak consistency while being able to easily check the desired correctness properties.

In this paper, we develop criteria for checking the robustness of applications against three recently proposed consistency models—causal (aka causal+) consistency (CC) \[22\], prefix consistency \[11\] (PC) and parallel snapshot isolation \[26\] (PSI, aka non-monotonic snapshot isolation \[3\]). As a corollary of our results, we also derive an existing robustness criterion \[16\] for a classical model of snapshot isolation \[6\] (SI). Our criteria also handle variants of the consistency models that allow application programmers to request that certain transactions be executed under serializability and thereby ensure the robustness of applications that are not robust otherwise.

We handle the above four consistency models in a uniform and systematic way by exploiting a recently proposed framework \[12\] for declaratively specifying their semantics (Section 2). In particular, all of the consistency models that we consider guarantee the atomic visibility of transactions: either all or none of the writes performed by a transaction can be observed by other transactions. This allows us to simplify reasoning needed to establish robustness criteria by abstracting from internals of transactions in application executions.

We first propose a dynamic robustness criterion that checks whether a given execution is serializable (Section 3). We formulate this criterion in terms of the dependency graph of the execution \[2\], describing several kinds of relationships between its transactions: an execution
is serializable if its dependency graph contains no cycles of a certain form, which we call critical. Criteria for robustness against different consistency models differ in which cycles are considered critical. We then illustrate how our dynamic robustness criteria on a single execution can be lifted to static criteria that check that all executions of a given application are serializable (Section 4).

2 Consistency Model Specifications

We start by recalling from [12] a formal model of database computations and the specifications of the consistency models that we handle. These specifications are declarative, which greatly simplifies our formal development. Nonetheless, as shown in [12], the specifications are equivalent to certain operational specifications, close to implementations.

We consider a database storing objects \( \text{Obj} = \{x, y, \ldots\} \), which we assume to be natural-valued. Clients interact with the database by issuing read and write operations on the objects, grouped into transactions. We denote each operation invocation by an event \((i, o)\), where \(i\) is an identifier from a denumerable set \(\text{EventId} \) and \(o \in \{\text{read}(x, n), \text{write}(x, n) \mid x \in \text{Obj}, n \in \mathbb{N}\}\) describes the operation invoked and its outcome: reading a value \(n\) from an object \(x\) or writing \(n\) to \(x\). We range over events by \(\_\) and denote the set of all events by Event. In the following we denote irrelevant expressions by \(_\), and write \(e \vdash \text{write}(x, n)\) if \(e = (_\_, \text{write}(x, n))\) and \(e \vdash \text{read}(x, n)\) if \(e = (_\_, \text{read}(x, n))\). A binary relation \(<\) is a strict partial order if it is transitive and irreflexive. It is total if additionally for all elements \(a, b\), we have \(a < b\), \(b < a\) or \(a = b\).

**Definition 1.** A transaction \(T, S, \ldots\) is a pair \((E, <_{\text{po}})\), where \(E \subseteq \text{Event}\) is a finite, non-empty set of events with distinct identifiers, and the program order \(<_{\text{po}}\) is a total order over \(E\). A history \(H\) is a finite set of transactions with disjoint sets of event identifiers. An annotated history \((H, \text{level})\) is a pair where \(H\) is an history and \(\text{level} : H \rightarrow \{\text{SER}, \perp\}\). An execution is a triple \(X = ((H, \text{level}), <_{\text{hb}}, <_{\text{wr}})\), where \((H, \text{level})\) is an annotated history; \(<_{\text{hb}}\) is a strict partial order over \(H\), and \(<_{\text{wr}}\) is a total order over \(H\) such that \(<_{\text{hb}} \subseteq <_{\text{wr}}\). We refer to \(<_{\text{hb}}\) and \(<_{\text{wr}}\) as happens-before and arbitration.

We denote components of an execution as in \(X.H\) and use the same notation for similar structures.

A transaction records a set of operations and the order in which the client program invoked them. A history records transactions that committed in a finite database computation. For simplicity we elide the treatment of aborted and ongoing transactions, as well as infinite database computations. Annotated histories enrich histories with a function level which records which transactions the programmer requested to execute under serializability, and which transactions under the weak consistency model offered by the underlying database. Finally, executions enrich annotated histories with a happens-before order and an arbitration order, which declaratively represent internal database processing. Intuitively, \(T <_{\text{hb}} S\) means that \(S\) is aware of the updates performed by \(T\), and thus the outcome of the operations in \(S\) may depend on the effects of \(T\). We call transactions that are not related by happens-before concurrent. The relationship \(T <_{\text{wr}} S\) means that the versions of objects written by \(S\) supersede those written by \(T\). The constraint \(<_{\text{hb}} \subseteq <_{\text{wr}}\) ensures that writes by a transaction \(T\) supersede those that \(T\) is aware of.

We use the set \{CC, PC, PSI, SI, SER\} to refer to the consistency models that we treat (Section 1), and we range over this set by \(\text{wm}\). In Figure 4 we specify these consistency models as combination of the axioms in Figure 3, constraining executions. Formally, we let
7:4 Robustness against Consistency Models with Atomic Visibility

(a) Causality violation. Disallowed by the transitivity of \( \prec_{hb} \).

\[
T_1 \begin{cases}
\text{write}(x, book) \prec_{po} \text{write}(y, empty) \\
\text{read}(x, book) \prec_{po} \text{write}(y, review)
\end{cases}
\]

\[
T_3 \begin{cases}
\text{read}(x, empty) \prec_{po} \text{read}(y, review)
\end{cases}
\]

(b) Lost update. Disallowed by CONFLICT.

\[
T_1 \begin{cases}
\text{read}(nbids, 0) \prec_{po} \text{write}(nbids, 1)
\end{cases}
\]

\[
T_3 \begin{cases}
\text{read}(nbids, 1)
\end{cases}
\]

\[
T_2 \begin{cases}
\text{read}(nbids, 0) \prec_{po} \text{write}(nbids, 1)
\end{cases}
\]

(c) Long fork. Disallowed by PREFIX.

\[
T_1 \begin{cases}
\text{write}(x, Alice) \prec_{po} \text{read}(y, -)
\end{cases}
\]

\[
T_2 \begin{cases}
\text{read}(y, Bob) \prec_{po} \text{read}(x, -)
\end{cases}
\]

\[
T_4 \begin{cases}
\text{read}(x, -) \prec_{po} \text{read}(y, Bob)
\end{cases}
\]

(d) Write skew. Disallowed by TOTALHB.

\[
\begin{cases}
\text{read}(\neg x) \prec_{po} \text{write}(Alice, y) \\
\text{read}(\neg y) \prec_{po} \text{write}(Alice, x)
\end{cases}
\]

Figure 2 Non-serializable executions illustrating anomalies. Boxes represent transactions, and thin arrows between boxes represent the happens-before relation. We omit arbitration edges to avoid clutter. The thick arrows marked \( wr/wr \) are explained in Section 3.

\[
\forall(E, \prec_{po}) \in H. \forall e \in E. \forall x, n.
\]

\[
e \vdash \text{read}(x, n) \implies (\text{before}(e, \prec_{po}, \_ x) = \emptyset \lor \text{max}(\text{before}(e, \prec_{po}, \_ x), \prec_{po}) \vdash \_ (\_ n))
\]

(INT)

\[
\forall T \in H. \forall x, n.
\]

\[
T \vdash \text{read}(x, n) \implies ((\text{before}(T, \prec_{hb}, \text{write } x) = \emptyset \land n = 0) \lor \text{max}(\text{before}(T, \prec_{hb}, \text{write } x), \prec_{aw}) \vdash \_ (\_ n))
\]

(EXT)

\[
\forall T, S \in H. (\exists x. T \vdash \text{write}(x, \_) \land S \vdash \text{write}(x, \_)) \implies T = S \lor T \prec_{hb} S \lor S \prec_{hb} T
\]

(CONFLICT)

\[
\prec_{aw} : \prec_{hb} \subseteq \prec_{aw}
\]

(PREFIX)

\[
\forall T, S \in H. (\text{level}(T) = \text{level}(S) = \text{SER}) \implies T = S \lor T \prec_{hb} S \lor S \prec_{hb} T
\]

(TOTALHB)

\[
\forall T, S \in H. (\text{level}(T) = \text{level}(S) = \text{SER})
\]

(SERTotal)

Figure 3 Consistency axioms constraining an execution \(((H, \text{level}), \prec_{hb}, \prec_{aw})\).

the set of annotated histories allowed by a consistency model \( wm \) be given by \( \text{hist}(wm) = \{ (X, H, X, \text{level}) \mid X \models wm \} \). We now explain the axioms and the anomalies that they (dis)allow. We summarise these anomalies in Figure 2.

Given a total order \( \prec \subseteq A \times A \) and a set \( B \subseteq A \), we write \( \text{max}(B, \prec) \) for the element \( b \in B \) such that \( \forall a \in B. a \leq b \); if \( A = \emptyset \), then \( \text{max}(B, \prec) \) is undefined. We define min in the obvious dual manner. In the following, when we write \( \text{max}(B, \prec) \) or \( \text{min}(B, \prec) \), we assume that they are defined. Given a partial order \( \prec \subseteq A \times A \) and an \( a \in A \), we define the downset of \( a \) as \( \text{before}(a, \prec) = \{ a' \in A \mid a' < a \} \), and let \( \text{before}(a, \prec, \text{op } x) = \text{before}(a, \prec) \cap \{ a' \in A \mid a' \vdash \text{op } (x, \_ \}) \).

The internal consistency axiom INT ensures that, within a transaction, the database provides sequential semantics: in a transaction \((E, \prec_{po})\), a read event \( e \) on an object \( x \) returns the value of the last event on \( x \) preceding \( e \). The events on \( x \) preceding \( e \) are given by the set \( \text{before}(e, \prec_{po}, \_ x) \). If in \((E, \prec_{po})\) a read \( e \) on \( x \) is not preceded by an operation on the same object (i.e., \( \text{before}(e, \prec_{po}, \_ x) = \emptyset \)), then its value is determined in terms of writes by other transactions, using the external consistency axiom EXT. To formulate
CC \equiv \text{Int} \land \text{Ext} \land \text{SerTotal}

\text{PSI} \equiv \text{Ext} \land \text{Conflict}

\text{PC} \equiv \text{CC} \land \text{Prefix}

\text{SI} \equiv \text{PSI} \land \text{Prefix}

\text{SER} \equiv \text{Int} \land \text{Ext} \land \text{TotalHB}

\textbf{Figure 4} Consistency model definitions.

\text{Ext} we lift the \triangleright notation to transactions. For given \(T = (E, <_{\text{po}}), x \in \text{Obj} \) and \(n \in \mathbb{N}\), we write:

- \(T \triangleright \text{write}(x, n)\) if \(\max\{e \in E \mid e \triangleright \text{write}(x, \_\_), <_{\text{po}}\} \triangleright \text{write}(x, n)\); and
- \(T \triangleright \text{read}(x, n)\) if \(\min\{e \in E \mid e \triangleright \_\_-(x, \_\_), <_{\text{po}}\} \triangleright \text{read}(x, n)\).

According to \text{Ext}, if a transaction \(T\) reads an object \(x\) before writing to it, then the value returned by the read is determined by the transactions that happen before \(T\) and that write to \(x\); the set of such transactions is given by \text{before}(T, <_{\text{bb}}, \text{write}\ x).\) If this set is empty, then \(T\) reads the initial value 0; otherwise it reads the value written by the transaction from the set that is the last one in \(<_{\text{al}}.\) \text{Ext} guarantees the atomic visibility of a transaction: either all or none of its writes can be visible to another transaction. A detailed discussion on the matter can be found in [12, Section 3].

The axiom \text{SerTotal} formalises the additional guarantees provided to transactions that the application programmer required to execute on serializability, as recorded by \text{level}. We discuss this axiom in more detail below; for now we assume \text{level} = (\lambda T. \bot) for all executions.

The axioms \text{Int}, \text{Ext} and \text{SerTotal} define causal consistency (CC) [22]. This forbids the causality violation anomaly in Figure 2(a), where a user sees the review, but not the book it was associated with. This anomaly is forbidden because \(<_{\text{bb}}\) is transitive, so we must have \(T_1 <_{\text{bb}} T_3\). Since \(T_1 <_{\text{al}} T_2\), the writes by \(T_2\) supersede those by \(T_1\), and thus \text{Ext} implies \(T_3 \triangleright \text{read}(y, \text{review})\) and \(T_3 \triangleright \text{read}(x, \text{book})\).

Causal consistency allows the lost update anomaly, illustrated by the execution in Figure 2(b). This execution may arise from the programs \text{ViewItem} and \text{StoreBid} in Figure 1, which respectively let a user query the information about an item and bid on an item. They access a table ITEMS, whose rows represent items and contain a primary key (iId), an item description (desc) and the number of existing bids (nbids). The anomaly in Figure 2(b) is caused by two invocations of the program \text{StoreBid} that generate the transactions \(T_1\) and \(T_2\), meant to increase the number of bids of an item. The two transactions read the initial number of bids for the item, namely 0, and concurrently modify it, resulting in one addition getting lost. This is observed by a third transaction \(T_3\) generated by \text{ViewItem}. The lost update anomaly is disallowed by the axiom \text{Conflict}, which guarantees that transactions updating the same object are not concurrent. This axiom rules out any execution with the history in Figure 2(b). We specify parallel snapshot isolation (PSI) [26] by strengthening causal consistency with the axiom \text{Conflict}. This consistency model allows the long fork anomaly given in Figure 2(c), which we discussed in Section 1.

We specify prefix consistency (PC) [11] and snapshot isolation (SI) [6] by strengthening respectively CC and PSI via the axiom \text{Prefix}: if \(T\) observes \(S\), then it also observes all \(<_{\text{al}}\)-predecessors of \(S\), which is formalised using sequential composition \(;\) of relations. The axiom \text{Prefix} disallows any execution with the history in Figure 2(c): \(T_1\) and \(T_2\) have to be related by \(<_{\text{al}}\) one way or another; but then by \text{Prefix}, either \(T_3\) has to observe Alice or \(T_3\) has to observe Bob. Like causal consistency, prefix consistency allows the lost update
Our first goal is to define criteria to check whether a single execution $X$ in one of the consistency models that we consider has a serializable history: $X.H \in \text{hist}(\text{SER})$. From these dynamic robustness criteria, in the next section we derive static criteria to check whether this is the case for all executions of a given application.

Our dynamic criteria are formulated in terms of dependency graphs, widely used in the database literature [2]. Let the set of labels $L$ be defined as follows: $D = \{(\text{wr}, x), (\text{ww}, x) \mid x \in \text{Obj}\}$, $L = D \cup \{(\text{rw}, x) \mid x \in \text{Obj}\}$. We use $\lambda$ to range over $L$ and $s, t$ to range over
A graph $G$ is a pair $(H, \rightarrow)$, where $H \in \text{Hist}$ and $\rightarrow \subseteq H \times L \times H$. We write $T \xrightarrow{\lambda} S \in G$ in place of $(T, \lambda, S) \in \rightarrow$. We also use some graph-theoretic notions. A path $\pi$ in $G$ is a non-empty finite sequence of edges $T_0 \xrightarrow{\lambda_0} T_1 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n-1}} T_n$. In this case we write $\pi \in G$. The path is a cycle if $T_0 = T_n$, and it is a simple cycle if all other pairs of transactions on it are distinct. We also write $T \xrightarrow{\lambda} S \in \pi$ to mean that the edge $T \xrightarrow{\lambda} S$ appears in the path $\pi$. Given a graph $G = (H, \rightarrow)$, we denote by $\rightarrow^+$ the least relation such that

(i) $T \xrightarrow{\lambda^+} S$ if $T \xrightarrow{\lambda} S \in G$; and

(ii) $T \xrightarrow{\lambda S} S$ whenever $T \xrightarrow{\lambda} T' \in G$ and $T' \xrightarrow{S} S$ for some $T' \in H$.

We denote with $\rightarrow^*$ the reflexive closure of $\rightarrow^+$, so that $T \xrightarrow{\rightarrow^*} T$ for every $T \in H$. We now define a map from executions into dependency graphs.

**Definition 2.** The dynamic dependency graph of an execution $X = ((H, _), \ll_{hb}, \ll_{ar})$ is $\text{DDG}(X) = (H, \rightarrow)$, where for every $x \in \text{Obj}$ the relation $\rightarrow$ contains the following triples:

- **read-dependency:** $T \xrightarrow{\text{wr}_x} S$ if $T \xrightarrow{\text{read}(x, _)}$ and $T = \max(\text{before}(S, \ll_{hb}, \text{write} x), \ll_{ar})$;

- **write-dependency:** $T \xrightarrow{\text{ww}_x} S$ if $T \xrightarrow{\text{write}(x, _)}$ and $T \xrightarrow{\text{write}(x, _)}$ and $T \xrightarrow{<_{ar} S}$;

- **anti-dependency:** $T \xrightarrow{\text{ar}_x} S$ if $T \xrightarrow{\text{write}(x, _)}$ and $T \xrightarrow{\text{write}(x, _)}$ and $T \xrightarrow{<_{ar} S}$.

Thus, $T \xrightarrow{\text{wr}_x} S$ means that $S$ reads $T$’s write to $x$ (cf. EXT in Figure 3), and $T \xrightarrow{\text{ww}_x} S$ means that $S$ overwrites $T$’s write to $x$. The relation $T \xrightarrow{\text{ar}_x} S$ means that $S$ overwrites the write to $x$ read by $T$ (the initial value of an object is overwriten by any write to this object). In Figures 2(c), 2(d) and 5 we draw the dependency graphs with thick edges.

Dependency graphs provide a way to show that executions have serializable histories [2].

**Lemma 3.** For every $X$, if $X \models \text{INT} \land \text{EXT}$ and $\text{DDG}(X)$ is acyclic, then $X \cdot H \in \text{hist}(\text{SER})$.

For instance, the history in Figure 5 is serializable. The graphs of the executions in Figures 3(c, d) contain cycles and, in fact, the histories of these executions are not serializable.

As we now show, to ensure that the history of an execution $X$ arising from a particular consistency model is serializable, it is enough to check that $\text{DDG}(X)$ does not contain cycles of a particular form, which we call *critical*. This more precise characterisation is instrumental in obtaining our static robustness criteria (Section 4).

A path $\pi$ in a dynamic dependency graph $G$ is **chord-free** if, whenever $u \xrightarrow{S} v \in \pi$ for some $S \in L^n$ with $n \geq 2$, we have $\neg(u \rightarrow v \in G)$. A path $\pi$ is *rw-minimal* if, whenever $u \xrightarrow{\text{rw}} v \in \pi$ and $u \xrightarrow{\lambda} v \in G$, we have $\lambda = (\text{rw}, _)$. The last notion forces us exclude an $\text{rw}$ edge from $\pi$ if there is another option.

**Definition 4.** Given an execution $X$, an edge $T \xrightarrow{\lambda} S \in \text{DDG}(X)$ is *unprotected* if either $X\cdot \text{level}(T) \neq \text{SER}$ or $X\cdot \text{level}(S) \neq \text{SER}$. A cycle $\pi \in \text{DDG}(X)$ among transactions $T_0, T_1, \ldots, T_n$ (where $T_0 = T_n$) that is simple, chord-free and rw-minimal is:

- **CC-critical**, if $\pi$ contains an unprotected edge $T_i \xrightarrow{\text{rw}} T_{i+1}$ and an unprotected edge $T_j \xrightarrow{\lambda} T_{j+1}$ with $i \neq j$ and $\lambda \in \{\text{ww}, _\}$;

- **PC-critical**, if $\pi$ contains an unprotected edge $T_i \xrightarrow{\text{rw}} T_{i+1}$ and at least two adjacent unprotected edges with labels in $\{\text{ww}, _\}$;

- **PSI-critical**, if:

  1. $\pi$ contains at least two unprotected $\text{rw}$ edges; and
Robustness against Consistency Models with Atomic Visibility

2. for every $T_i \xrightarrow{rw,x} T_{i+1}$, $T_j \xrightarrow{rw,y} T_{j+1} \in \pi$, if $i \neq j$, then $x \neq y$;

SI-critical, if:

1. $\pi$ contains at least two adjacent unprotected $rw$ edges; and

2. for every $T_i \xrightarrow{rw,x} T_{i+1}$, $T_j \xrightarrow{rw,y} T_{j+1} \in \pi$, if $i \neq j$, then $x \neq y$.

The graphs of the executions in Figure 2 (with level = $(\lambda T_\bot)$) contain critical cycles: (c) contains a PSI-critical cycle, and (d) contains an SI-critical cycle.

**Theorem 5.** For every $wm$ and $X$, if $X \models wm$, then $DDG(X)$ contains a cycle if and only if it contains a $wm$-critical cycle.

From Theorem 5 and Lemma 3 we obtain our dynamic robustness criterion.

**Corollary 6.** For every $wm$ and every $X$, if $X \models wm$ and $DDG(X)$ contains no $wm$-critical cycle then $X.H \in \text{hist}(SER)$.

We note that the above robustness criterion for SI is a variant of an existing one [16, 15]. In our setting, it is just a consequence of our novel criterion for PSI.

To prove Theorem 5 we show how the axioms in Figure 3 impact the properties of edges and paths in dependency graphs. First, observe that there is a relation between $wr$, $ww$ edges and the orders $<_hb$ and $<_wr$: for every $X$ and every $T, S \in X.H$, the definitions ensure that

\[
T \xrightarrow{wr} S \in DDG(X) \implies T <_{hb} S;
\]

\[
(T \xrightarrow{wr} S \in DDG(X) \lor T \xrightarrow{ww} S \in DDG(X)) \implies T <_{wr} S;
\]

\[
(X \models \text{Conflict} \land T \xrightarrow{wr} S \in DDG(X)) \implies T <_{hb} S.
\]

These implications let us show that, under certain conditions, if two transactions in a dependency graph are connected by a path, then they are also related by happens-before or arbitration.

**Lemma 7.** For any $X$, $s \in L^+$ and $T \xrightarrow{\Delta^+} S \in DDG(X)$, if $X \models \text{SERTotal}$ then:

1. if all the $rw$ and $ww$ edges in $T \xrightarrow{\Delta^+} S$ are protected then $T <_{hb} S$;

2. if all the $rw$ edges in $T \xrightarrow{\Delta^+} S$ are protected then $T <_{wr} S$;

3. if $X \models \text{Conflict}$ and all the $rw$ edges in $T \xrightarrow{\Delta^+} S$ are protected then $T <_{hb} S$.

The following lemma shows that, if $T \xrightarrow{rw,x} S$, then $T$ cannot happen-before $S$: in this case $T$ would have to read a value at least as up-to-date as that written by $S$, contradicting the definition of anti-dependencies.

**Lemma 8.** $\forall X. \forall x \in \text{Obj.} \forall T, S \in X.H. T \xrightarrow{rw,x} S \in DDG(X) \implies S \not\nsubseteq_{hb} T \land T \vdash \text{read}(x,_-) \land S \vdash \text{write}(x,_-)$.

**Proof of Theorem 5.** The if implication is obvious, so let us prove the only if implication. Suppose that the graph $DDG(X)$ contains a cycle $\pi'$. From $\pi'$ we can easily build a cycle

\[
\pi = T_0 \xrightarrow{\lambda_0} T_1 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_{n-1}} T_n \quad \text{(where } T_0 = T_n, \ n \geq 2)\]

in $DDG(X)$ that is simple, chord-free and $rw$-minimal. The argument now is a case analysis on the $wm$. Here we consider only CC and PSI and defer the full proof to [7].

**Case of $wm = \text{CC}$.** Lemma 7(2) implies that $\pi$ contains at least one unprotected $rw$ edge, for otherwise $T_0 <_{ar} T_0$, contradicting the irreflexivity of $X <_{ar}$. Let this edge be $T_i \xrightarrow{rw} T_{i+1}$. Then Lemma 8 ensures that $T_{i+1} \not\nsubseteq_{hb} T_i$. Since $\pi$ contains the non-empty
path $T_{i+1} \xrightarrow{rw,x} T_i$, Lemma 7(1) implies that on this path there is at least one unprotected edge $T_j \xrightarrow{rw,x} T_{j+1}$ with $\lambda \in \{ww,\_\}, \{rw,\_\}$ and $i \neq j$. It follows that $\pi$ is CC-critical.

Case of $wm = PSI$. First we prove that the cycle $\pi$ contains at least two unprotected edges $rw$ edges. Since $X \models PSI$, we know that $X \models CC$. Thus, the previous argument ensures that the cycle $\pi$ contains at least one unprotected $rw$ edge, say $T_i \xrightarrow{rw,x} T_{i+1}$. Suppose that $\pi$ contains exactly one such edge. Since $X \models CONFLICT$, Lemma 7(3) now ensures $T_{i+1} \not<_{hb} T_i$. But by Lemma 8, $T_i \xrightarrow{rw,x} T_{i+1}$ implies $T_{i+1} \not<_{hb} T_i$. The resulting contradiction shows that $\pi$ must contain at least two unprotected $rw$ edges.

Now we have to prove

$$\forall T_i \xrightarrow{rw,x} T_{i+1}, T_j \xrightarrow{rw,y} T_{j+1} \in \pi. i \neq j \implies x \neq y.$$  \hfill (2)

Suppose that $\pi$ does not satisfy (2). Then, as $\prec_{rw}$ is total, Definition 2 guarantees that we have either $T_{i+1} \xrightarrow{wm,z} T_{j+1}$ or $T_{j+1} \xrightarrow{wm,z} T_{i+1}$. Since $\pi$ is a simple cycle, in the first case we contradict either that $\pi$ is chord-free or that $\pi$ is $rw$-minimal. In the second case, we have either (a) $T_{j+1} = T_i$ or (b) $T_{j+1} \neq T_i$. If (a) holds, then we contradict that $\pi$ is $rw$-minimal, because $T_i \xrightarrow{rw,x} T_{i+1} \in \pi$ and $T_i \xrightarrow{wm,z} T_{i+1}$. If (b) holds, then the sub-path $T_{j+1} \xrightarrow{rw,x} T_{i+1}$ of $\pi$ contains at least two edges and it is chord-free by construction. But this contradicts $T_{j+1} \xrightarrow{wm,z} T_{i+1}$. It follows that $\pi$ satisfies (2) above, and thus it is a PSI-critical cycle.

4 Static Robustness Criteria

We now illustrate how the dynamic robustness criteria (Corollary 6) can be lifted to static criteria, which allow programmers to analyse the behaviour of their applications and which can serve as a basis for static analysis tools.

We define an application $A$ by a set of transactional programs $f_i$, giving the code of its transactions: $A = \{f_1, \ldots, f_n\}$ (e.g., see Figure 1). As is standard in the database literature [16], this abstracts from the rest of the application logic to focus on the parts that directly interact with the database. We call a pair $I = (f, v)$ of a program and a vector of its actual parameters a program instance. An application instance $I$ is a set of program instances, and an annotated application instance is a pair $(I, levels)$, where $levels : I \to \{SER, \perp\}$ defines which programs the programmer requested to execute under serializability. We first formulate criteria for checking the robustness of a particular annotated application instance, resulting from running a set of transactional programs with given parameters. We then sketch how these criteria can be generalised to whole applications.

We aim to illustrate the ideas for lifting dynamic robustness criteria to static ones in the simplest form. To this end, we abstract from the syntax of the programming language and assume that we are only given approximate information about the set of objects read or written by each transactional program. Namely, we assume a function $rwsets$ that maps every program instance $I$ to a triple $rwsets(I) = (R^O, W^O, W^C)$. Informally, $R^O$ and $W^O$ are the sets of all the objects that may be read or written in some execution of $I$, and $W^C$ is a set of the objects that must be written in any execution of $I$, with the proviso that $W^C \subseteq W^O$. For instance, for $I = \langle StoreBid, (iId_1, 7) \rangle$ (Figure 1) we have $rwsets(I) = \{(ITEMS(iId_1).nbids), \{ITEMS(iId_1).nbids, BIDS(*,*)\}, \{ITEMS(iId_1).nbids\}$ where $*$ means “all fields” or “all rows”.

To formalise the meaning of the read/write sets, we define a relation that determines if a history can be produced by a given $I$. We let $T \models I$ for $rwsets(I) = (R^O, W^O, W^C)$, if:
We now formulate our static robustness criteria by using the same notions of paths and cycles for static dependency graphs as for dynamic ones (Section 3). Given a pair \((\mathcal{I}, \text{level}_\mathcal{I})\) and a cycle in \(\pi \in \text{SDG}(\mathcal{I})\) among program instances \(I_0, I_1, \ldots, I_n\) (where \(I_0 = I_n\), we
say that an \( rw \) edge \( I_t \xrightarrow{rw,x} I_{t+1} \in \pi \) is critical in \( \pi \), if for all \( I_1, I_m \) in \( \pi \) such that \( t \neq m \) and for all \( t, t' \in D^* \) such that \( I_t \xrightarrow{\ldots \cdot x} I_{t'} \in \pi \) and that \( I_{t+1} \xrightarrow{\ldots \cdot y} I_m \in \pi \), we have \( \neg (I_t \xrightarrow{\text{ww}, \text{xy}} I_m) \). For example, the graph in Figure 6 contains the following cycle \( \pi \), in which the left-most \( rw \) edge is critical, while the right-most \( rw \) edge is not critical:

\[
\begin{array}{c}
\text{ViewItem(id)} \xrightarrow{rw} \text{StoreBid(id, 7)} \xrightarrow{\text{ww}} \text{ViewItem(id)} \xrightarrow{rw} \text{StoreBid(id, 7)} \xrightarrow{\text{ww}} \text{ViewItem(id)}
\end{array}
\]  

\( (4) \)

**Definition 11.** Given a pair \((I, \text{levels})\), an edge \( I_t \xrightarrow{\ldots \cdot x} I_{t+1} \in \text{SDG}(I) \) is unprotected if either \( \text{levels}(I_t) \neq \text{SER} \) or \( \text{levels}(I_{t+1}) \neq \text{SER} \). A cycle \( \pi \in \text{SDG}(I) \) among program instances \( I_0, I_1, \ldots, I_n \) (where \( I_0 = I_n \)) is:

- **CC-critical**, if \( \pi \) contains an unprotected edge \( I_t \xrightarrow{\ldots \cdot x} I_{t+1} \) and an unprotected edge \( I_j \xrightarrow{\ldots \cdot y} I_{j+1} \) with \( i \neq j \) and \( x, y \in \{\text{ww}, \text{xy}, \text{yx}, \text{yy}\} \);  

- **PC-critical**, if \( \pi \) contains an unprotected edge \( I_t \xrightarrow{\ldots \cdot x} I_{t+1} \) and at least two adjacent unprotected edges with labels in \( \{\text{ww}, \text{xy}, \text{yx}, \text{yy}\} \);  

- **PSI-critical**, if:
  1. \( \pi \) contains at least two unprotected critical \( rw \) edges; and  
  2. for every \( I_t \xrightarrow{\ldots \cdot x} I_{t+1} \), \( I_t \xrightarrow{\ldots \cdot x} I_{t+1} \in \pi \), if \( i \neq j \), then \( x \neq y \);  

- **SI-critical**, if:
  1. \( \pi \) contains at least two adjacent unprotected critical \( rw \) edges; and  
  2. for every \( I_t \xrightarrow{\ldots \cdot x} I_{t+1} \), \( I_t \xrightarrow{\ldots \cdot x} I_{t+1} \in \pi \), if \( i \neq j \), then \( x \neq y \).  

Note that, unlike a critical cycle in a dynamic dependency graph (Definition 4), a critical cycle in a static graph does not have to be simple. The following lemma states that \( \lhd \) preserves critical cycles.

**Lemma 12.** For every \( wm \), \((G, \text{level}_G)\) and \((F, \text{level}_F)\) such that \((G, \text{level}_G) \lhd (F, \text{level}_F)\), if \( G \) contains a \( wm \)-critical cycle, then \( F \) contains a \( wm \)-critical cycle.

Lemmas 10 and 12 (which are proven in [7]) and Corollary 6 establish our static criteria.

**Theorem 13.** For every \((H, \text{level}_H)\), \((I, \text{level}_I)\) and \(wm\), if \( \text{SDG}(I) \) contains no \( wm \)-critical cycles and \((H, \text{level}_H) \in \text{hist}(wm)\), then whenever \((H, \text{level}_H) \vdash (I, \text{level}_I)\), we have \( H \in \text{hist(SER)} \).

For example, let \( \text{level}_I = (\lambda I, \bot) \) and consider \( I \) defined by (3). The corresponding static dependency graph in Figure 6 contains PSI-critical cycles, one of which is obtained by following twice the loop RegUser(Alice) \( \xrightarrow{\text{rw,USERS(\_).name}} \) RegUser(Alice). Indeed, as we explained in Section 2, the annotated instance \((I, \text{level}_I)\) is not robust against PSI, because it can produce the write skew anomaly in Figure 2(d). Now let \( \text{level}_I'(\text{RegUser}, \text{Alice}) = \text{SER} \) and \( \text{level}_I'(\_) = \bot \) otherwise. Figure 6 contains the static dependency graph corresponding to the annotated instance \((I, \text{level}_I)\). This graph does not contain PSI-critical cycles. To see why, observe that in the graph there are only two kinds of cycles: the ones due to the self-loop on the node RegUser(Alice), and the ones that connect nodes in \{StoreBid(idl1, 7), StoreBid(idl1, 10), ViewItem(idl1)\}. The cycles of the first kind contain only protected \( rw \) edges thanks to \( \text{level}_I'(\_) \), while the cycles of the second kind contain at most one critical \( rw \) edge, as sketched in (4) above. It follows that no cycle is PSI-critical, and thus by executing only the RegUser transaction on serializability, we make \( I \) robust. However, the graph contains a CC-critical cycle, namely the one shown in (4) above. It is CC-critical for its two \( rw \) edges are unprotected. As we explained in Section 2, under CC \( I \) may produce the lost update anomaly in Figure 2(b), and it is unsafe to run \( I \) over a CC database.
Analysing Whole Applications. The static criteria in Theorem 13 allow a programmer to analyse the robustness of a given application instance. Analysing an application completely using the theorem requires considering an infinite number of its instances, a task best done by an automatic static analysis tool. We now sketch how ideas from abstract interpretation [13, 25] can be used to finitely represent and analyse the set of all instances of an application. In the future, this can pave the way to automating our robustness criteria in static analysis tools. Due to space constraints, we only present the concepts by an example.

We associate an application $A$ with a summary dependency graph $SDG(A)$ that summarises the static graphs of all the instances of $A$. In Figure 7 we show the summary dependency graph $SDG(A)$ for the application in Figure 1. Every program in $A$ yields a summary node in the graph $SDG(A)$, representing all instances of the program. Every edge in $SDG(A)$ is a summary edge, summarising the possible dependencies between the corresponding programs. It is annotated by a constraint relating the actual parameters of the incident programs between which the dependency exists. For example, the summary edge from ViewItem to StoreBid in Figure 7 means that for every instance $I$ of $A$ we have $StoreBid(id_1, \_)$ $\xrightarrow{rw, x}$ $ViewItem(id_2)$ in $SDG(I)$ iff $x \in \{BIDS(\_), iid, BIDS(\_), val, ITEMS(iid_1).nbids\}$ and $iid_1 = iid_2$. Similarly, the $ww$ edge incident to $StoreBid$ means that we have $StoreBid(id_1, \_)$ $\xleftarrow{ww, x}$ $StoreBid(id_2, \_)$ in $SDG(I)$ iff $x = ITEMS(iid_1).nbids$ and $iid_1 = iid_2$.

Definition 11 carries over to summary graphs of applications by taking into account the constraints on summary edges when checking whether a given $rw$ edge is critical in a given cycle $\pi$, and whether the objects that appear on the $rw$ edges of $\pi$ are different. For example, consider the following cycle in the graph in Figure 7:

$$\xrightarrow{wr, ITEMS(iid_1).nbids; \_}$ $\xleftarrow{rr, x}$ $ViewItem(iid_1)$ $\xrightarrow{rr, x}$ $StoreBid(iid_2, \_)$ $\xleftarrow{rr, x}$ $ViewItem(iid_2)$ $\xrightarrow{rr, x}$ $StoreBid(iid_2, \_)$ $\xleftarrow{rr, x}$ $StoreBid(iid_2, val)$ $\xrightarrow{ww, x}$ $StoreBid(iid_2, \_)$ $\xleftarrow{ww, x}$ $ViewItem(iid_2)$ $\xrightarrow{ww, x}$ $StoreBid(iid_2, val)$

We consider the $rw$ edge on this cycle not critical. This is because the constraints on the edges in the cycle imply $iid_1 = iid_2$, which satisfies the constraint on the must $ww$ edge between $StoreBid$ programs in Figure 7. For any annotated instance $(I, levels)$ of the application $A$ in Figure 1, using the adjusted Definition 11 we can check that, if $levels$ maps the instances of $RegUser$ and $ViewItem$ in $I$ to $SER$, then $(I, levels)$ is robust against PSI.

\[\text{Marking $ViewItem$ as $SER$ is actually unnecessary to make this application robust under PSI, because the graph $SDG(A)$ contains an edge $ViewItem(\_)$ $\rightarrow$ $RegUser(uname)$ which does not exist in the dependency graph of any execution of $A$. This can be addressed by a more precise static analysis.}\]
5 Related Work

In the setting of databases, application robustness was first investigated by Fekete et al. [16], who proposed a criterion for robustness against snapshot isolation (SI) [6]. Fekete then extended the criterion to SI databases allowing the programmer to request serializability for certain transactions [15], a mechanism that we also consider. Our criterion is formulated in a way similar to that of Fekete et al., using dependency graphs [2]. However, in contrast to their work, we consider more subtle models of parallel snapshot isolation, prefix consistency and causal consistency, which allow more anomalies than SI. The method we use is also different from that of Fekete et al. They consider an operational specification of SI [6], which makes the proof of the robustness criterion highly involved. In contrast, we benefit from using declarative specifications that achieve conciseness by exploiting atomic visibility of transactions [12]. This allows us to come up with robustness criteria more systematically.

Robustness has also been investigated for applications running on weak shared-memory models of common multiprocessors and programming languages (e.g., [9]). However, this line of work has not considered applications using transactions. Transactions complicate the consistency model semantics, which makes establishing robustness criteria more challenging.

Serializability of transactions in an application simplifies establishing its correctness properties, but is not necessary for this. Thus, an alternative approach to establishing application correctness is to prove its desired properties directly, without requiring the transactions to produce only serializable behaviours. Corresponding methods have been proposed for ANSI SQL isolation levels and SI by Lu et al. [23], and for PSI and some of other recent models by Gotsman et al. [18]. Such methods are complementary to ours: the conditions they require can be satisfied by more applications, but are more difficult to check than robustness.

6 Conclusion

In this paper we have made the first steps towards understanding the impact of recently-proposed transactional consistency models for large-scale databases on the correctness properties of applications using them. To this end, we have proposed criteria for checking when an application using a weak consistency model exhibits only strongly consistent behaviours. This enables programmers to check that application correctness will be preserved for a particular choice of a consistency model or transactions to be executed under serializability.

The robustness result of Fekete et al. for SI has previously given rise to automatic tools for statically detecting anomalies in applications [19]. Our work could form a basis for similar advances in databases providing weaker consistency models. Our dynamic robustness criteria are also of an independent interest: apart from serving as a basis for static analysis, such criteria can be used to optimise run-time monitoring algorithms [24, 28].

In establishing our robustness criteria, we have followed a systematic approach that exploits axiomatic specifications [12]: using the axioms of a consistency model, we have characterised the cycles allowed in dependency graphs of executions on the model, and exploited the characterisations to provide sound static analysis techniques. We hope that this method will be applicable to other consistency models being proposed for large-scale databases.

References

1 D. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. IEEE Computer, 45(2), 2012.
Robustness against Consistency Models with Atomic Visibility


9 A. Bouajjani, E. Derevenetc, and Roland M. Checking and enforcing robustness against TSO. In ESOP, 2013.


