Tolerant Constraint-Preserving Snapshot Isolation: Extended Concurrency for Interactive Transactions

Stephen J. Hegner
Umeå University
Department of Computing Science
SE-901 87 Umeå Sweden
hegner@cs.umu.se
http://hegner.people.cs.umu.se

Abstract

In a database setting involving writers, there is a delicate balance between allowing sufficient concurrency for adequate performance and adequate isolation to prevent transactions from interfering with each other. This problem is particularly acute for interactive transactions; that is, ones which involve human input, since state-of-the-art approaches, such as serializable snapshot isolation (SSI), which rely upon an abort-and-restart strategy to resolve conflicts, do not provide a suitable solution. In this work, an extension of constraint-preserving snapshot isolation (CPSI) is provided. As does CPSI, this extension provides snapshot-isolation (SI) plus constraint preservation. By employing a model in which the values of data objects, and not just their identities, are used, it provides a significantly higher level of concurrency without sacrificing isolation than do approaches without such value modelling. In addition to the theory, an operational model of transaction execution is provided.

1 Introduction

Support for concurrent database transactions has long been recognized as a difficult problem. In the *ACID* characterization [6, pp. 166-167], [7, Sec. 1.1], transactions must run in *isolation*; that is, they must not interfere with one another. In practice, enforcing the theoretical ideal *view serializability* [13, Sec. 2.4] of isolation has proven difficult to enforce efficiently. Consequently, lower levels of isolation are commonly found in practice, including *snapshot isolation* (*SI*) at an intermediate level and *read committed* (*RC*) among the lowest levels, with the level of acceptability dependent upon the application.

Strategies for managing concurrency may be classified along a pessimistic-optimistic dimension. In a *pessimistic* policy, a transaction must wait until it may be given access to the resource(s) which it needs. In an *optimistic* strategy, a transaction is given access to a (copy of a) resource immediately, with conflicts resolved before the writes of the transaction become part of the persistent database. Roughly speaking, pessimistic policies use waiting to avoid problems, while optimistic policies require transactions to abort and restart. All real policies involve both pessimism and optimism, but the classification is nevertheless a useful one.

For the purposes of this paper, an *interactive transaction* is one which requires interactive human input at certain stages in order to continue. Business processes, in which humans authorize tasks and provide input, such as the allocation of funds for a business trip [8] [11], are prime examples. The problem of providing suitable isolation while supporting adequate concurrency is particularly acute for such transactions. Human input for transaction T may not be available at the time at which it is needed, or additional time may be needed in order to reach a decision before input can be provided. If a pessimistic strategy for concurrency is employed, then another transaction T' which needs resources which cannot be allocated until T commits or at least continues, must wait, perhaps for days, until T finally completes the interaction and proceeds with its execution. If T' is also interactive, such a long delay is likely unacceptable. On the other hand, if an optimistic strategy for concurrency is employed, then in the case of such a conflict, one of T and T' must be aborted and restarted. In many cases, that may not be suitable, since further input may be required, and even if not, the state of the database will likely have changed, and the inputs may have depended upon the state. In short, neither pessimistic nor optimistic policies are appropriate for interactive transactions. On the other hand, since human interaction is many orders of magnitude slower than computer operations, there is ample time to take more complex measures in order to minimize conflicts in the first place.

Virtually all mainstream work on transaction concurrency is based upon an *object-level* model, in which conflict between two transactions is characterized entirely with respect to access to data objects, without any regard to their current values or to how the transactions might alter those values. The prospects for supporting interactive transactions within such a framework are limited. Using a finer granularity for the data objects (for example, fields instead of tuples in a relational context) may help occasionally, but often conflicts between two transactions are inherent to atomic, indecomposable attributes, such as balances in accounts. The resulting conflicts must still be addressed, by waiting or else by aborting at least one of the transactions involved, neither of which is desirable.

To maintain an adequate level of isolation while minimizing waits and aborts, another approach is to employ a finer-grained notion of conflict. In a *value-level* model, not only the identity of a data item but also its current value, as well as how the transactions involved intend to change that value, may be taken into account. In this case, the transaction manager will be more complex, but the payoffs in terms of increased concurrency may be substantial. The idea is not new; it was proposed more than thirty years ago for long-running transactions in computer-aided design [1], with a more comprehensive theory along the same lines presented in [12]. Those works, however, are focused upon sets of nested transactions, assembled to realize a single goal via cooperation, and have not seen widespread use outside of that context.

The focus of this paper is the development of a simple value-level model for concurrency of transactions, called *tolerant constraint-preserving snapshot isolation*, or *TCPSI*. It builds upon *constraint-preserving snapshot isolation* (*CPSI*) [10], a framework based upon object-level modelling, which provides the isolation and efficiency of SI, together with a guarantee of constraint preservation[1]. Although not providing true serializability, the constraint preservation

[1] Most modern relational DBMSs enforce all built-in constraints, such as primary-, secondary-, and foreign-key dependencies, internally, in real time, regardless of the level of isolation. However, this is not the case for *extended* constraints, defined using triggers or via application programs. CPSI guarantees the preservation of all constraints, including extended ones. See [10, Summary 2.4] for further explanation.

of CPSI avoids well-known anomalies such as write skew [2, A5B], [5, Ex. 2.2]. This is significant because, while lack of true serializability may be tolerable, violation of integrity constraints almost never is. Additionally, substantially fewer conflict situations arise with CPSI than with *serializable SI* (*SSI*) [4], [5]. TCPSI is a true extension of CPSI, in that all concurrency allowed under CPSI is also allowed under TCPSI.

To illustrate the main idea of this paper, consider the classical write-skew example [2, A5B], which illustrates how SI can fail to preserve constraints. Let $\mathbf{E}_0$ be a schema with two integer-valued data objects x_1 and x_2, related by the constraint $x_1 + x_2 > 0$. Let T_{01} be the transaction which reduces the value of x_1 by 1 if the result satisfies the constraint and otherwise does nothing; *i.e.*, $\mathsf{if}\ (x_1 + x_2 > 1)\ \mathsf{then}\ x_1 \leftarrow x_1 - 1$; here the values of x_1 and x_2 seen by T_{01} are those found in the snapshot taken when T_{01} starts; after that point, it never sees updates performed by other transactions. Similarly, let T_{02} be the analogous transaction for x_2, *i.e.*, $\mathsf{if}\ (x_1 + x_2 > 1)\ \mathsf{then}\ x_2 \leftarrow x_2 - 1$. Under SSI, and even under CPSI, these transactions are prevented from running concurrently, since an illegal state may arise under certain conditions. Indeed, if $x_1 = x_2 = 1$ initially, then each transaction may run successfully in isolation, but if they run concurrently under SI, the resulting state, with $x_1 = x_2 = 0$, is not legal. Each model identifies T_{01} as a writer of x_1 and a reader of x_2, as well as T_{02} a writer of x_2 and a reader of x_1. Thus, each transaction reads an object written by the other, indicating a potential conflict. However, that does not mean that there will be a conflict, only that there might be. Whether or not a violation of integrity occurs depends upon the initial state. Under TCPSI, for T_{01}, rather than prohibiting a concurrent transaction from writing x_2, a tolerance on the range of writes is stipulated. If the initial snapshot of T_{01} has $x_1 = x_2 = 2$, for example, then T_{01} will tolerate updates to x_2, as long as the final value of x_2 is at least 0. Likewise, T_{02} will tolerate updates to x_1, as long as the final value of x_1 is at least 0. In that situation, T_{01} and T_{02} would be allowed to execute concurrently under TCPSI, although such concurrency would be blocked both under CPSI and under SSI.

For more than two concurrent transactions, additional issues arise. Let $\mathbf{E}_1$ be the extension of $\mathbf{E}_0$ to three data objects. More precisely, $\mathbf{E}_1$ has three integer-valued data objects x_1, x_2, and x_3, constrained by $x_1 + x_2 + x_3 > 0$. For $i \in \{1, 2, 3\}$, define the transaction T_{1i} by $\mathsf{if}\ (x_1 + x_2 + x_3 > 1)\ \mathsf{then}\ x_i \leftarrow x_i - 1$. For an initial state with $x_1 + x_2 + x_3 = 3$ (*e.g.*, M_{10} with $x_1 = x_2 = x_3 = 1$), it is easy to see that at most two of the three transactions in $\{T_{11}, T_{12}, T_{13}\}$ may execute concurrently without a constraint violation. For T_{11} (for example), the admissibility of its update for the initial state M_{10} requires that $x_2 + x_3 > 0$, a combined condition of the objects to be updated by T_{12} and T_{13}. To retain pairwise testing, the solution forwarded in this work is to require T_{11} to place separate conditions on x_2 and x_3 which imply that $x_2 + x_3 > 0$ remains true during its lifetime. For example, it may require that $x_2 \geq 0$ and $x_3 > 0$ (in which case T_{13} is blocked from concurrent execution while T_{12} may proceed), or it may choose that $x_2 > 0$ while $x_3 \geq 0$, (in which case T_{12} is blocked but T_{13} may proceed). In return for creating some false positives and thus reducing potential concurrency, a far simpler test for admissible concurrency results. For $k > 3$ a positive integer and $0 < m < k$, this form of example extends to to k transactions, of which at most k may execute concurrently; thus, no test which does not involve all transactions simultaneously is sufficient to guarantee concurrency without false positives. The reader who is curious about these details now is invited to look at 4.4 for a detailed example of the concurrency problems, at 4.10 for details of the proposed solution sketched above, and at 5.7 for a detailed operational example of how

the approach works. Although some aspects will require reading other parts of the paper, it should be possible to grasp the main ideas with only the above example as background.

The paper is organized as follows. Section 2 provides the database framework used, while CPSI is reviewed in Sec. 3. Both sections are summaries of ideas developed in detail in [10]. In Sec. 4, the theory of TCPSI is developed in detail, while Sec. 5 provides an operational model of how the entire process proceeds. Finally, Sec. 6 contains conclusions and further directions.

2 The Database Framework

In this section, the database framework which is used throughout this paper is sketched. With few exceptions, the concepts are taken from [10], to which the reader is referred for details and examples. Although [10] is based upon the earlier paper [9], the frameworks differ substantially; the journal article [10] should in all cases be taken as the primary reference.

2.1 Notation $f(x)\downarrow$ indicates that the partial function f is defined on x. $f(x)\downarrow \in Y$ indicates that both $f(x)\downarrow$ and $f(x) \in Y$. $\mathbb{Z}$ denotes the set of integers. For $i, j \in \mathbb{Z}$, $[i, j] = \{n \in \mathbb{Z} \mid i \leq n \leq j\}$.

2.2 Data objects, constraints, and schemata A *data object* x is a mutable object; that is, an object whose value may be altered. A *simple data object* x is indecomposable; it is characterized by the set $\mathsf{States}\langle x\rangle$ of its *states*. A *compound data object* (or just *data object*) is a set $\mathbf{x}$ of simple data objects. A *database* over $\mathbf{x}$ is a function $M : \mathbf{x} \to \bigcup_{x\in\mathbf{x}} \mathsf{States}\langle x\rangle$ with the property that $M(x) \in \mathsf{States}\langle x\rangle$ for each $x \in \mathbf{x}$, with the set of all databases over $\mathbf{x}$ denoted $\mathsf{DB}(\mathbf{x})$. Put another way, M defines an $\mathbf{x}$ tuple $\mathbf{s} \in \prod_{x\in\mathbf{x}} \mathsf{States}\langle x\rangle$ of values, with $\pi_x(\mathbf{s}) = M(x)$.

An *unconstrained database schema* $\mathbf{d}$ is just a data object $\mathsf{DObj}\langle\mathbf{d}\rangle$. A *database* of $\mathbf{d}$ is a database over $\mathsf{DObj}\langle\mathbf{d}\rangle$. The set of all databases of $\mathbf{d}$ is denoted $\mathsf{DB}(\mathbf{d})$. Thus, $\mathsf{DB}(\mathbf{d})$ is shorthand for $\mathsf{DB}(\mathsf{DObj}\langle\mathbf{d}\rangle)$.

A *constrained database schema* is a triple $\mathbf{D} = \langle\mathsf{DObj}\langle\mathbf{D}\rangle, \mathsf{LDB}(\mathbf{D}), \mathsf{ELDB}(\mathbf{D})\rangle$ in which $\mathsf{DObj}\langle\mathbf{D}\rangle$ is a set of data objects, $\mathsf{LDB}(\mathbf{D})$ is a subset of $\mathsf{DB}(\mathsf{DObj}\langle\mathbf{D}\rangle)$, the set of *legal databases* of $\mathbf{D}$, and $\mathsf{ELDB}(\mathbf{D})$ is a subset of $\mathsf{LDB}(\mathbf{D})$, the set of *extended legal databases*, or *x-legal databases*, of $\mathbf{D}$. Think of $\mathsf{DB}(\mathbf{D})$ as the set of all databases, regardless of constraints. The reason for the distinction between $\mathsf{LDB}(\mathbf{D})$ and $\mathsf{ELDB}(\mathbf{D})$ is that, as noted in the footnote of Sec. 1, most modern relational DBMSs enforce all built-in constraints. $\mathsf{LDB}(\mathbf{D})$ represents the databases which satisfy all such internal constraints, while $\mathsf{ELDB}(\mathbf{D})$ represents those databases which satisfy all constraints, including those defined by means such as triggers; for example, $x_1 + x_2 > 0$ of $\mathbf{E}_0$, described in Sec. 1. The work in this paper, as well as the earlier work on CPSI, [10], is concerned with isolation which preserves not only membership in $\mathsf{LDB}(\mathbf{D})$ but also in $\mathsf{ELDB}(\mathbf{D})$.

Define $\mathsf{SubObj}\langle\mathbf{D}\rangle = \{\mathbf{y} \mid \mathbf{y} \subseteq \mathsf{DObj}\langle\mathbf{D}\rangle\}$ to be the set of *subobjects* of $\mathbf{D}$. Let $\mathbf{x}, \mathbf{y} \in \mathsf{SubObj}\langle\mathbf{D}\rangle$. For $M \in \mathsf{DB}(\mathbf{x})$, the *restriction* of M to $\mathbf{y}$ is the database on $\mathbf{x} \cap \mathbf{y}$ defined by $M_{|\mathbf{x}\cap\mathbf{y}}$, the function M restricted to $\mathbf{x} \cap \mathbf{y}$. As a slight abuse of notation, this restriction will also be written as simply $M_{|\mathbf{y}}$, with the understanding that subobjects in $\mathbf{y}$ which do not apply to M (*i.e.*, which are not in $\mathbf{x}$ also) are ignored. For $\mathbf{M} \subseteq \mathsf{DB}(\mathbf{x})$, $\mathbf{M}_{|\mathbf{y}} = \{M_{|\mathbf{y}} \mid M \in \mathbf{M}\}$.

The database schema $[\![\mathbf{D}|\mathbf{y}]\!] = \langle \mathbf{y}, \mathsf{LDB}\langle\mathbf{D}|\mathbf{y}\rangle, \mathsf{ELDB}\langle\mathbf{D}|\mathbf{y}\rangle\rangle$, in which $\mathsf{LDB}\langle\mathbf{D}|\mathbf{y}\rangle = \{M_{|\mathbf{y}} \mid M \in \mathsf{LDB}(\mathbf{D})\}$ and $\mathsf{ELDB}\langle\mathbf{D}|\mathbf{y}\rangle = \{M_{|\mathbf{y}} \mid M \in \mathsf{ELDB}(\mathbf{D})\}$.

A compound data object may be the empty set $\emptyset$. In that case $\mathsf{DB}(\emptyset)$ is a function on domain $\emptyset$. There is only one such function, so the empty database object has just one possible database, which will be denoted by ϕ_{DB}. It is always the case that $\phi_{\mathsf{DB}} \in \mathsf{ELDB}\langle\mathbf{D}|\emptyset\rangle$.

2.3 Notational convention Throughout the rest of this paper, unless stated specifically to the contrary, take $\mathbf{D} = \langle \mathsf{DObj}\langle\mathbf{D}\rangle, \mathsf{LDB}(\mathbf{D}), \mathsf{ELDB}(\mathbf{D})\rangle$ to be a (constrained) database schema.

2.4 Updates and updateable objects A *(syntactic) update* on $\mathbf{D}$ is a pair $u = \langle u^{(1)}, u^{(2)}\rangle \in \mathsf{LDB}(\mathbf{D}) \times \mathsf{DB}(\mathbf{D})$. $u^{(1)}$ is the current or old state before the update; $u^{(2)}$ the new state afterwards. $\mathsf{SynUpdates}(\mathbf{D})$ denotes the set of all syntactic updates on $\mathbf{D}$. $u \in \mathsf{SynUpdates}(\mathbf{D})$ is *legal* if $u^{(2)} \in \mathsf{LDB}(\mathbf{D})$, with the set of all legal updates on $\mathbf{D}$ denoted $\mathsf{LUpdates}(\mathbf{D})$. It is *extended legal* (or *x-legal*) if $u^{(1)}, u^{(2)} \in \mathsf{ELDB}(\mathbf{D})$. The set of all extended legal updates on $\mathbf{D}$ is denoted $\mathsf{ELUpdates}(\mathbf{D})$. Note that $\mathsf{ELUpdates}(\mathbf{D}) \subseteq \mathsf{LUpdates}(\mathbf{D}) \subseteq \mathsf{SynUpdates}(\mathbf{D})$. $\mathbf{u} \subseteq \mathsf{SynUpdates}(\mathbf{D})$ is *complete* if for every $M \in \mathsf{LDB}(\mathbf{D})$, there is a $u \in \mathbf{u}$ with $u^{(1)} = M$. Define $u_1 \circ u_2 = \{(M_1, M_3) \mid (\exists M_2 \in \mathsf{LDB}(\mathbf{D}))((M_1, M_2) \in u_1 \wedge (M_2, M_3) \in u_2)\}$. The *identity update* on $\mathbf{x} \in \mathsf{DObj}\langle\mathbf{D}\rangle$ is $1_{\mathbf{x}} = \{(N, N) \mid N \in \mathsf{DB}(\mathbf{x})\}$.

For $u \in \mathsf{SynUpdates}(\mathbf{D})$, $\mathbf{y} \subseteq \mathsf{DObj}\langle\mathbf{D}\rangle$, define $u_{|\mathbf{y}} = \langle {u^{(1)}}_{|\mathbf{y}}, {u^{(2)}}_{|\mathbf{y}}\rangle$. For $\mathbf{u} \subseteq \mathsf{SynUpdates}(\mathbf{D})$, define $\mathbf{u}_{|\mathbf{y}} = \{u_{|\mathbf{y}} \mid u \in \mathbf{u}\}$. The *trimming* of $\mathbf{u}$ to $M \in \mathsf{LDB}(\mathbf{D})$ is $\mathsf{Trim}_M\langle\mathbf{u}\rangle = \{u \in \mathbf{u} \mid u^{(1)} = M\}$. If $\mathbf{x}, \mathbf{y} \subseteq \mathsf{DObj}\langle\mathbf{D}\rangle$, $M \in \mathsf{LDB}\langle\mathbf{D}|\mathbf{x}\rangle$, and $\mathbf{u} \subseteq \mathsf{SynUpdates}([\![\mathbf{D}|\mathbf{y}]\!])$, then $\mathsf{Trim}_M\langle\mathbf{u}\rangle$ is shorthand for $\mathsf{Trim}_{M_{|\mathbf{y}}}\langle\mathbf{u}\rangle = \mathsf{Trim}_{M_{|\mathbf{x}\cap\mathbf{y}}}\langle\mathbf{u}\rangle$. In general, for any $M \in \mathsf{LDB}(\mathbf{D})$, $\mathbf{u}(M)$ denotes $\{u^{(2)} \mid (u \in \mathbf{u}) \wedge (u^{(1)} = M)\} = \{u^{(2)} \mid u \in \mathsf{Trim}_M\langle\mathbf{u}\rangle\}$. Call $\mathbf{u} \subseteq \mathsf{SynUpdates}(\mathbf{D})$ *functional* if for every $M \in \mathsf{LDB}(\mathbf{D})$, $\mathbf{u}(M)$ contains at most one element, and for $\mathbf{M} \subseteq \mathsf{LDB}(\mathbf{D})$, define $\mathbf{u}(\mathbf{M}) = \{\mathbf{u}(M) \mid M \in \mathbf{M}\}$.

An *updateable object* over $\mathbf{D}$ is a pair $\langle\mathbf{c}, \mathbf{u}\rangle$ in which $\mathbf{c} \subseteq \mathsf{DObj}\langle\mathbf{D}\rangle$, and $\mathbf{u} \subseteq \mathsf{SynUpdates}([\![\mathbf{D}|\mathbf{c}]\!])$, with $\langle\mathbf{c}, \mathbf{u}\rangle$ *functional*, (resp. *complete*, resp. *legal*, resp. *x-legal*) precisely in the case that $\mathbf{u}$ has that property; it is called *singleton* if $\mathbf{u}$ consists of just one update. The updateable object $\langle\mathsf{DObj}\langle\mathbf{D}\rangle, \mathbf{u}\rangle$ is abbreviated to $\langle\mathbf{D}, \mathbf{u}\rangle$. For $\langle\mathbf{c}, \mathbf{u}\rangle$, $\mathbf{x} \subseteq \mathsf{DObj}\langle\mathbf{D}\rangle$ with $\mathbf{c} \subseteq \mathbf{x}$, and $M \in \mathsf{LDB}\langle\mathbf{D}|\mathbf{x}\rangle$, define $\mathsf{Trim}_M\langle\langle\mathbf{c}, \mathbf{u}\rangle\rangle = \langle\mathbf{c}, \mathsf{Trim}_M\langle\mathbf{u}\rangle\rangle$; and for $\mathbf{y} \subseteq \mathbf{c}$, define $\langle\mathbf{c}, \mathbf{u}\rangle_{|\mathbf{y}} = \langle\mathbf{y}, \mathbf{u}_{|\mathbf{y}}\rangle$.

2.5 Transactions A *black-box transaction* T over $\mathbf{D}$ is represented by an updateable object $\langle\mathbf{D}, \mathcal{U}_T\rangle$ which is functional, complete, and x-legal. The set of all black-box transactions over $\mathbf{D}$ is denoted $\mathsf{BBTrans}_{\mathbf{D}}$. The notation $\langle\mathbf{D}, \mathcal{U}_T\rangle$ will be used throughout the rest of this paper to denote the updateable object which underlies the transaction T.

2.6 The contexts of a transaction The contexts of a transaction are central to this work. Here, only brief illustration via example is provided. For a full discussion, see [10, Disc. 3.18].

Let $\mathbf{E}_2$ be the schema with four integer-valued data objects: x_1, x_2, y_1, and y_2, constrained by $x_1 + x_2 > 0$. Let T_{21} be the transaction defined by the rule $\mathsf{if}\ (x_1 + x_2) - y_1 > 0\ \mathsf{then}\ x_1 \leftarrow x_1 - y_1$; in other words, the value of x_1 is reduced by the value of y_1, provided the result will satisfy the constraints. Otherwise, no update is performed. For M_{20}, defined by $(x_1 = 3, x_2 = 3, y_1 = 1, y_2 = 0)$, the *grounded write* of this update is $\{3 \overset{x_1}{\rightsquigarrow} 2\}$, representing the update on $\mathsf{ELDB}(\mathbf{E}_2)$

which changes x_1 from 3 to 2, leaving the other three data objects fixed. The *write context* is the set of all data objects which are written; in this case $\{x_1\}$. The grounded write specifies a change to the write context, without referring to other data objects. For M_{21}, defined by $(x_1 = 3, x_2 = -1, y_1 = 2, y_2 = 2)$, the ground write is $\emptyset$; *i.e.*, there is no change to the state, with the write context is $\emptyset$ as well. For full details on ground updates, see [10, Def. 3.9].

Continuing with the example, the *read context* is $\{x_2, y_1\}$, and the members of this set are further subclassified into the the *integrity context*, consisting of $\{x_2\}$, and the *grounding context*, consisting of $\{y_1\}$. The integrity context consists of those reads which are necessary to verify that the integrity constraints will be satisfied after the update; it will be formalized via the notion of guard in 3.4. The grounding context consists of those reads which are necessary to determine the grounded write. The grounding and integrity contexts need not be disjoint. Formally, the read and write contexts are always taken to be disjoint. If a data object is written by a transaction, its old value (before the update) is irrelevant to constraint satisfaction after the update, while the new value is automatically considered in checking integrity constraints.

2.7 Write sets and trimming The *write set* $\mathsf{WSet}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$ of the updateable object $\langle \mathbf{c}, \mathbf{u}\rangle$ is the set of all $y \in \mathbf{c}$ for which there is a legal $u \in \mathbf{u}$ which alters the state of y. Formally, $\mathsf{WSet}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle = \{y \in \mathbf{c} \mid (\forall x \in \mathbf{x})(\exists u \in \mathbf{u} \cap \mathsf{LUpdates}([\![\mathbf{D}|\mathbf{c}]\!]))({u^{(1)}}_{|\{x\}} \neq {u^{(2)}}_{|\{x\}})\}$. $\mathsf{WUpd}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle = \mathbf{u}_{|\mathsf{WSet}\langle\langle \mathbf{c},\mathbf{u}\rangle\rangle}$ is the set of *write updates* of $\langle \mathbf{c}, \mathbf{u}\rangle$, while $\mathsf{WObj}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle = \langle \mathsf{WSet}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle, \mathsf{WUpd}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle\rangle$ is its set of *write objects*. $\langle \mathbf{c}, \mathbf{u}\rangle$ is a *full write object* if $\langle \mathbf{c}, \mathbf{u}\rangle = \mathsf{WObj}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$.

For $M \in \mathsf{LDB}(\mathbf{D})$, the *write trim* of $\langle \mathbf{c}, \mathbf{u}\rangle$ to M is $\mathsf{WTrim}_M\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle = \langle \mathbf{c}, \mathbf{u}\rangle_{|\mathsf{WSet}\langle\mathsf{Trim}_M\langle\langle \mathbf{c},\mathbf{u}\rangle\rangle\rangle}$. $\mathsf{WTrim}_M\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$ is a full write object, and if $\langle \mathbf{c}, \mathbf{u}\rangle$ is functional and complete, then $\mathsf{WUpd}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$ consists of exactly one update. If $\mathbf{c} = \emptyset$, that update must be $\langle \phi_{\mathsf{DB}}, \phi_{\mathsf{DB}}\rangle$.

For a transaction T whose initial snapshot is $M \in \mathsf{LDB}(\mathbf{D})$, the update which it performs is represented by the (single) update of the updateable object $\mathsf{WTrim}_M\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle$. Data objects not in $\mathsf{WTrim}_M\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle$ are left unchanged.

2.8 Lifting of updates Let $\langle \mathbf{c}, \mathbf{u}\rangle$ be an updateable object on $\mathbf{D}$, and let $\mathbf{x} \subseteq \mathsf{DObj}\langle\mathbf{D}\rangle$ with $\mathbf{c} \subseteq \mathbf{x}$. $\langle \mathbf{c}, \mathbf{u}\rangle$ may be *lifted* to an update $\mathbf{x}$ by requiring that it be the identity on all data objects in $\mathbf{c} \setminus \mathbf{x}$. Formally, $\mathsf{Lift}_{[\![\mathbf{D}|\mathbf{x}]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle = \{v \in \mathsf{SynUpdates}([\![\mathbf{D}|\mathbf{x}]\!]) \mid (v_{|\mathbf{c}} \in \mathbf{u})$ and $({v^{(1)}}_{|\mathbf{x}\setminus\mathbf{c}} = {v^{(2)}}_{|\mathbf{x}\setminus\mathbf{c}})\}$ is the *lifting* of $\langle \mathbf{c}, \mathbf{u}\rangle$ (or just of $\mathbf{u}$) from $[\![\mathbf{D}|\mathbf{c}]\!]$ to $[\![\mathbf{D}|\mathbf{x}]\!]$. If $\langle \mathbf{c}, \mathbf{u}\rangle$ is functional, then so too is $\mathsf{Lift}_{[\![\mathbf{D}|\mathbf{x}]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$. In that case, $\mathsf{FLift}_{[\![\mathbf{D}|\mathbf{x}]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle : \mathsf{LDB}\langle\mathbf{D}|\mathbf{x}\rangle \to \mathsf{DB}(\mathbf{x})$ is the partial function defined by $M \mapsto M'$ if $\langle M, M'\rangle \in \mathsf{Lift}_{[\![\mathbf{D}|\mathbf{c}]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$ and is undefined otherwise. If $\mathbf{u}$ consists of a single update, then $\langle \mathbf{c}, \mathbf{u}\rangle$ is functional, so $\mathsf{Lift}_{[\![\mathbf{D}|\mathbf{x}]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$ is functional as well. When $x = \mathsf{DObj}\langle\mathbf{D}\rangle$, a simpler notation is used: $\mathsf{Lift}_{[\![\mathbf{D}|\mathsf{DObj}\langle\mathbf{D}\rangle]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$ is abbreviated to $\mathsf{Lift}_{\mathbf{D}}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$, and $\mathsf{FLift}_{[\![\mathbf{D}|\mathsf{DObj}\langle\mathbf{D}\rangle]\!]}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$ is abbreviated to $\mathsf{FLift}_{\mathbf{D}}\langle\langle \mathbf{c}, \mathbf{u}\rangle\rangle$.

3 SI and CPSI

Tolerant CPSI is an extension of ordinary CPSI; thus, it is first necessary to present the core ideas of the latter in a manner which is precise enough to allow their application and extension to the former. Due to space limitations, the presentation is abbreviated; for full details, the reader is referred to [10].

3.1 Transaction Schedules and Snapshot Isolation Both ordinary and tolerant CPSI are built upon *snapshot isolation* (*SI*) [14, 12.5], [15, Sec. 10.6.2], a commonly used level of isolation in modern DBMSs which employ *multiversion concurrency control* (*MVCC*) [3, Ch. 5], [14, Ch. 12], [15, Ch. 5]. Under SI, each transaction T receives, when it starts, a private copy of the database, called its *snapshot*. All operations by T, both reads and writes, during its duration, are on this private copy. T is allowed to commit its writes to the *global database* (the version of the database consisting of committed values, which is visible to other transactions) only if none of them writes a data object which another, concurrent transaction has already written.[2]

Each transaction T has a start time and an end time. Actual times are not important; rather, it is only the order of events which is significant. Let $\mathbf{T}$ be a finite set of transactions, and let $\mathsf{SCSet}\langle\mathbf{T}\rangle = \{T^s \mid T \in \mathbf{T}\} \cup \{T^c \mid T \in \mathbf{T}\}$, with the members of $\mathsf{SCSet}\langle\mathbf{T}\rangle$ just symbols. An *SI-schedule* on $\mathbf{T}$ is a total order $\leq_{\mathbf{T}}$ on $\mathsf{SCSet}\langle T\rangle$ with the property that for each $T \in \mathbf{T}$, $T^s <_{\mathbf{T}} T^c$. ($x <_{\mathbf{T}} y$ denotes that $x \leq_{\mathbf{T}} y$ but $x \neq y$.) The schedule indicates the temporal order of events, with T^s (resp. T^c) representing the relative start time (resp. commit time) of T. For $T_1, T_2 \in \mathbf{T}$, if neither $T_1^c <_{\mathbf{T}} T_2^s$ nor $T_2^c <_{\mathbf{T}} T_1^s$ holds, then T_1 and T_2 *execute concurrently* and $\{T_1, T_2\}$ forms a *concurrent pair*.

Fix an SI-schedule $\leq_{\mathbf{T}}$ for $\mathbf{T} \subseteq \mathsf{BBTrans}_{\mathbf{D}}$ and let $M \in \mathsf{ELDB}(\mathbf{D})$. As the transactions in $\mathbf{T}$ execute according to $\leq_{\mathbf{T}}$, the global database is updated by the transactions, as they commit. Given $M \in \mathsf{ELDB}(\mathbf{D})$ as the initial global database, (*i.e.*, before any transaction in $\mathbf{T}$ commits), there are three classes of values of interest for the global database which occur as the transactions run. $\mathsf{InitSnap}_{\langle\leq_{\mathbf{T}}: M\rangle}\langle T\rangle$ is the initial snapshot of T, the database which T reads at the beginning of its execution. Observe that $\mathsf{InitSnap}_{\langle\leq_{\mathbf{T}}: M\rangle}\langle T\rangle = M$ iff no transaction of $\mathbf{T}$ commits before T starts. $\mathsf{BeforeCmt}_{\langle\leq_{\mathbf{T}}: M\rangle}\langle T\rangle$ is the state of the global database immediately before T commits. It may differ from $\mathsf{InitSnap}_{\langle\leq_{\mathbf{T}}: M\rangle}\langle T\rangle$ because other transactions, running concurrently with T, may have committed before T and made changes to the global database. $\mathsf{AfterCmt}_{\langle\leq_{\mathbf{T}}: M\rangle}\langle T\rangle$ is the state of the global database immediately after T commits. It differs from $\mathsf{BeforeCmt}_{\langle\leq_{\mathbf{T}}: M\rangle}\langle T\rangle$ to the extent that T made changes to the global database upon its commit. It suffices to model only transactions which commit, since those which do not commit have no effect upon the global database, and so can simply be removed. Call $\leq_{\mathbf{T}}$ *constraint preserving for (initial state)* $M \in \mathsf{ELDB}(\mathbf{D})$ if for every $T \in \mathbf{T}$, $\mathsf{AfterCmt}_{\langle\leq_{\mathbf{T}}: M\rangle}\langle T\rangle \in \mathsf{ELDB}(\mathbf{D})$. In other words, *constraint preservation*, as used in the remainder of this paper, entails preservation of both internal and external constraints, as distinguished in 2.2.

For a more comprehensive presentation of SI, see [10, Sec. 2 and 4].

3.2 Notational convention In 3.3 and 3.4, take $\mathbf{T}$ to be a finite subset of $\mathsf{BBTrans}_{\mathbf{D}}$ and $\leq_{\mathbf{T}}$ an SI-schedule for $\mathbf{T}$.

3.3 State assignment Let $\mathbf{T}' \subseteq \mathbf{T}$. A *state assignment* for $\mathbf{T}'$ is a function $\iota : \mathbf{T}' \to \mathsf{LDB}(\mathbf{D})$. The most important use is the *state assignment for* $\leq_{\mathbf{T}}$ *with initial state* $M \in$

[2] In practice, some details may differ; in particular, internal constraints are always enforced immediately, and checks for concurrent writes to the same data object may be made earlier. However, those details do not affect the theory developed here materially. See [10, Summaries 2.2-2.4] for details.

$\mathsf{LDB}(\mathbf{D})$, the function $\mathsf{StAssign}_{\langle \leq_{\mathbf{T}} : M\rangle} : \mathbf{T} \to \mathsf{LDB}(\mathbf{D})$ given on elements by $T \mapsto \mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle$, identifying the initial snapshot of each $T \in \mathbf{T}$, when the entire schedule begins with database state M. For $\mathbf{T}' \subseteq \mathbf{T}$, $\mathsf{StAssign}^{|\mathbf{T}'}_{\langle \leq_{\mathbf{T}} : M\rangle} : \mathbf{T}' \to \mathsf{LDB}(\mathbf{D})$ is the function $\mathsf{StAssign}_{\langle \leq_{\mathbf{T}} : M\rangle}$ restricted to $\mathbf{T}'$. In particular, for $T_i, T_j \in \mathbf{T}$, $\mathsf{StAssign}^{|\{T_i,T_j\}}_{\langle \leq_{\mathbf{T}} : M\rangle}$ is the function $\mathsf{StAssign}_{\langle \leq_{\mathbf{T}} : M\rangle}$ restricted to $\{T_i, T_j\}$. When working with a general state assignment ι, think of it as a (possible) restriction of $\mathsf{StAssign}_{\langle \leq_{\mathbf{T}} : M\rangle}$ for an SI-schedule $\leq_{\mathbf{T}}$ with initial state M.

The state assignment $\iota : \mathbf{T}' \to \mathsf{LDB}(\mathbf{D})$ is nonoverlapping if none of its transactions writes a common data object (as required for concurrent transactions under SI). Formally, ι is *nonoverlapping* if for every $T_1, T_2 \in \mathbf{T}'$ with $T_1 \neq T_2$,

$$\mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathcal{U}_{T_1}\rangle\rangle\rangle \cap \mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_2)}\langle\langle \mathbf{D}, \mathcal{U}_{T_2}\rangle\rangle\rangle = \emptyset.$$

Given $M \in \mathsf{ELDB}(\mathbf{D})$, ι is *extendedly legal* (or *x-legal*) for M if, for every $T \in \mathbf{T}'$, $\mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T)}\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle\rangle(M)\downarrow \in \mathsf{ELDB}(\mathbf{D})$.

3.4 Guards and guard functions Database constraints are frequently quite localized in nature. If a data object $\mathbf{c}$ is to be written, only a small subset of the remaining data objects $\mathbf{y}$ must be checked to determine whether or not that write will preserve the integrity constraints. The notion of a guard object formalizes this. Let $\langle \mathbf{c}, \{u\}\rangle$ be a singleton full write object over $\mathbf{D}$; that is, a full write object with just one update. A *guard object* for $\langle \mathbf{c}, \{u\}\rangle$ is a $\mathbf{y} \in \mathsf{DObj}\langle \mathbf{D}\rangle$ which satisfies the following two properties.

(go-i) $\mathbf{y} \cap \mathbf{c} = \emptyset$.

(go-ii) For every $M \in \mathsf{ELDB}(\mathbf{D})$ with $M_{|\mathbf{c}} = u^{(1)}$,
$\mathsf{FLift}_{\mathbf{D}}\langle\langle \mathbf{c}, \{u\}\rangle\rangle(M)\downarrow \in \mathsf{ELDB}(\mathbf{D}) \Leftrightarrow \mathsf{FLift}_{[\![\mathbf{D}|\mathbf{y}\cup\mathbf{c}]\!]}\langle\langle \mathbf{c}, \{u\}\rangle\rangle(M_{|\mathbf{y}\cup\mathbf{c}})\downarrow \in \mathsf{ELDB}([\![\mathbf{D}|\mathbf{y} \cup \mathbf{c}]\!])$.

Condition (go-ii) states that the update $\langle \mathbf{c}, \{u\}\rangle$ is x-legal when lifted to all of $\mathbf{D}$ iff it is x-legal when lifted to just $[\![\mathbf{D}|\mathbf{y} \cup \mathbf{c}]\!]$, Thus, a global test for constraint satisfaction may be replaced by a much more localized one.

A guard function for a transaction provides a guard object for the write defined by each snapshot. Formally, a *guard function* for a transaction T is a $g : \mathsf{LDB}(\mathbf{D}) \to \mathsf{SubObj}\langle \mathbf{D}\rangle$ which assigns to each $N \in \mathsf{ELDB}(\mathbf{D})$ a guard object $g(N)$ for $\mathsf{WTrim}_N\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle$, subject to the additional condition that the guard depends only upon the update, and not the initial snapshot which induced it:

$$(\forall N_1, N_2 \in \mathsf{ELDB}(\mathbf{D}))((\mathsf{WTrim}_{N_1}\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle = \mathsf{WTrim}_{N_2}\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle) \Rightarrow (g(N_1) = g(N_2))).$$

Given $N \in \mathsf{ELDB}(\mathbf{D})$, $g(N)$ is called the *guard object* for N. The set of all guard functions for T is denoted $\mathsf{Guards}_{\mathbf{D}}\langle T\rangle$. This set is always nonempty; *i.e.*, a guard function always exists. See [10, 5.10] for details.

A *guarded black-box transaction* T is represented by a pair $\langle\langle \mathbf{D}, \mathcal{U}_T\rangle, \mathcal{G}_T\rangle$ in which $\langle \mathbf{D}, \mathcal{U}_T\rangle \in \mathsf{FUpdObj}(\mathbf{D})$ and $\mathcal{G}_T \in \mathsf{Guards}_{\mathbf{D}}\langle T\rangle$, with T represented by $\langle \mathbf{D}, \mathcal{U}_T\rangle$ in the latter. The set of all guarded black-box transactions over $\mathbf{D}$ is denoted $\mathsf{GBBTrans}_{\mathbf{D}}$.

The notion of a guard is closely related to, but not the same as, the idea of integrity context, as presented in 2.6. Consider again the schema $\mathbf{E}_2$ and the transaction T_{21} of 2.6. For snapshot M_{20}, the guard object is exactly the integrity context $\{x_2\}$. However, for snapshot M_{21}, although the integrity context is also $\{x_2\}$, the guard is $\emptyset$. The difference is that while the integrity context is used to determine whether or not a candidate ground write will preserve the integrity constraints (*i.e.*, whether that update would satisfy the *consistency* property of

ACID [6, p. 166]), the purpose of the guard is to identify a read set which must be protected from change in order to ensure *isolation* of ACID, once a consistent ground write has been selected. For an extensive set of examples of guards, see [10, 5.11-5.14].

3.5 Notational convention From now on, unless stated explicitly to the contrary, augment 3.2 so that $\mathbf{T}$ is taken to be a finite subset of $\mathsf{GBBTrans}_{\mathbf{D}}$, and not just of $\mathsf{BBTrans}_{\mathbf{D}}$. In other words, assume that every transaction in $\mathbf{T}$ has a guard function associated with it. Furthermore, as explained in 3.4 above, the guard function of $T \in \mathbf{T}$ will be denoted $\mathcal{G}_T$, while $<_{\mathbf{T}}$ continues to be an SI-schedule for $\mathbf{T}$.

3.6 Independent pairs of guarded transactions A pair of nonoverlapping concurrent transactions is guard independent if at least one does not write the guard of the other. Formally, given $\{T_1, T_2\} \subseteq \mathsf{GBBTrans}_{\mathbf{D}}$, a state assignment $\iota : \{T_1, T_2\} \to \mathsf{ELDB}(\mathbf{D})$ is *guard independent* if it is nonoverlapping and at least one of $\mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathcal{U}_{T_1}\rangle\rangle\rangle \cap \mathcal{G}_{T_2}(\iota(T_2)) = \emptyset$ or $\mathsf{WSet}\langle \mathsf{Trim}_{\iota(T_2)}\langle\langle \mathbf{D}, \mathcal{U}_{T_2}\rangle\rangle\rangle \cap \mathcal{G}_{T_1}(\iota(T_1)) = \emptyset$ holds, See [10, Thm. 5.17] for a proof of the following result, which is central to the results of Sec. 4.

3.7 Theorem — guard independence ⇒onstraint preservation *Let $M \in \mathsf{ELDB}(\mathbf{D})$. If $\mathsf{StAssign}^{|\{T,T'\}}_{\langle \leq_{\mathbf{T}} : M\rangle}$ is guard independent for every concurrent pair $\{T, T'\}$ of $\mathbf{T}$, then $\leq_{\mathbf{T}}$ is constraint preserving for initial state M.* □

3.8 Guard-write dependencies and CPSI *Constraint-preserving snapshot isolation*, or *CPSI*, is the application of guard independence to establish that a schedule of transactions, run under SI, will not result in any constraint violations, internal or external. In other words, under CPSI, for the schedule $\leq_{\mathbf{T}}$ to be constraint preserving, every pair $\{T_1, T_2\}$ of distinct concurrent transactions must satisfy one of the disjointness conditions identified in 3.6.

This may be expressed in another way; let $T_1, T_2 \in \mathbf{T}$. There is a *gw-dependency* from T_1 to T_2 for $\leq_{\mathbf{T}}$ with initial state $M \in \mathsf{ELDB}(\mathbf{D})$, written $T_1 \xrightarrow{\mathsf{gw}} T_2$, if T_2 writes the guard of T_1; that is, if $\mathsf{WSet}\langle \mathsf{Trim}_{\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T_2\rangle}\langle\langle \mathbf{D}, \mathcal{U}_{T_2}\rangle\rangle\rangle \cap \mathcal{G}_{T_1}(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T_1\rangle) \neq \emptyset$. $\{T_1, T_2\}$ is a *guard-write pair*, or *gw-pair*, in $\mathsf{GDSG}\langle \leq_{\mathbf{T}} : M\rangle$ if it forms a concurrent pair for which both $T_1 \xrightarrow{\mathsf{gw}} T_2$ and $T_2 \xrightarrow{\mathsf{gw}} T_1$ hold. The result of 3.7 may be restated to say that if $\leq_{\mathbf{T}}$ does not contain any guard-write pairs, then it is constraint preserving. For extensive examples surrounding CPSI, including in particular ones for which SSI flags conflict but CPSI does not, see [10, Examples 5.18 and 5.22]. For implementation issues, see [10, Disc. 5.25].

3.9 Examples of guard pairs and gw-dependency Consider again the transactions T_{01} and T_{02} on the schema $\mathbf{E}_0$, introduced in Sec. 1. Let $n_1, n_2 \in \mathbb{Z}$ with $n_1 + n_2 > 0$, and let $M_{0\langle n_1,n_2\rangle} \in \mathsf{ELDB}(\mathbf{E}_0)$ be the database with $x_1 = n_1$ and $x_2 = n_2$. It is easy to see that, for $i \in \{1, 2\}$, $\mathcal{G}_{T_i}(M_{0\langle n_1,n_2\rangle}) = \{x_{3-i}\}$ if the update $x_i \leftarrow x_i - 1$ is allowed; *i.e.*, if $n_1 + n_2 > 1$. If $n_1 + n_2 \leq 1$, the update would violate the integrity constraint $x_1 + x_2 > 0$, so the transaction executes the identify update instead, and $\mathcal{G}_{T_i}(M) = \emptyset$. Thus, under ordinary CPSI, T_{01} and T_{02} may not execute concurrently on the same initial snapshot $M_{0\langle n_1,n_2\rangle}$ with $n_1 + n_2 > 1$, since the gw-dependencies $T_{01} \xrightarrow{\mathsf{gw}} T_{02}$ and $T_{02} \xrightarrow{\mathsf{gw}} T_{01}$ both hold, identifying a gw-dependency (see

3.8). Nevertheless, it is clear that if $n_1+n_2>2$, the two may execute concurrently, on the same initial snapshot, with no integrity violation. In order to permit such concurrent execution, the guards need to be made *tolerant*, as developed in the next section.

4 Tolerant CPSI

In this section, the main ideas of tolerant CPSI are developed, as an extension of the ideas of CPSI outlined in Sec. 3.

4.1 Tolerant guard pairs, functions, and transactions Under CPSI, given two concurrent transactions, at least one is not allowed to write the guard of the other. Under tolerant CPSI, on the other hand, for each transaction, a set of allowable writes to its guard is specified. A concurrent transaction is allowed to write the guard, provided those writes lie within the specification. Formally, a *tolerant guard function* h for $T \in \mathsf{BBTrans}_{\mathbf{D}}$ assigns to each $N \in \mathsf{ELDB}(\mathbf{D})$ a pair $\langle \mathsf{GObj}_h(N), \mathsf{GTol}_h(N)\rangle$, with the following properties.

(tgp-i) GObj_h is a guard function for T; *i.e.*, $\mathsf{GObj}_h(N)$ is a guard object for $\mathsf{WTrim}_N\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle$.

(tgp-ii) $\mathsf{GTol}_h(N) \subseteq \mathsf{DB}(\mathsf{GObj}_h(N))$ with $N_{|\mathsf{GObj}_h(N)} \in \mathsf{GTol}_h(N)$.

(tgp-iii) $((N_{|\mathsf{WSet}\langle\mathsf{WTrim}_N\langle\langle\mathbf{D},\mathcal{U}_T\rangle\rangle\rangle} = \mathsf{WTrim}_N\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle^{(1)})$
$\wedge\ ((\mathsf{WTrim}_N\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle^{(2)})_{|\mathsf{GObj}_h(N)} \in \mathsf{GTol}_h(N))$
$\Rightarrow (\mathsf{FLift}_{\mathsf{GObj}_h(N)\cup\mathsf{WSet}\langle\langle\mathbf{D},\mathcal{U}_T\rangle\rangle}\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle(N)\downarrow \in \mathsf{ELDB}([\![\mathbf{D}|(\mathsf{GObj}_h(N)\cup\mathsf{WSet}\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle)]\!])).$

(tgp-iv) $(\forall N_1, N_2 \in \mathsf{ELDB}(\mathbf{D}))((\mathsf{WTrim}_{N_1}\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle = \mathsf{WTrim}_{N_2}\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle)$
$\Rightarrow (h(N_1) = h(N_2))).$

GObj_h is called the *guard-object assignment*, while GTol_h is called the *tolerance assignment*. For a fixed $N \in \mathsf{ELDB}(\mathbf{D})$, $\mathsf{GObj}_h(N)$ is called the *guard object* and $\mathsf{GTol}_h(N)$ is called the *tolerance* (of h) for N. As a slight abuse of notation, h may be written as $\langle \mathsf{GObj}_h, \mathsf{GTol}_h\rangle$.

Condition (tgp-i) identifies GObj_h as the associated guard function. Condition (tgp-ii) asserts that GTol_h assigns to each $N \in \mathsf{ELDB}(\mathbf{D})$ a set of databases of the guard object which includes in particular the projection of N onto the guard object. Condition (tgp-iii) liberalizes (go-ii) of 3.4. Instead of requiring that a concurrent transaction T' not write the guard object at all, the the tolerance identifies a range of values, within which a write of T' may lie. Since $N_{|\mathsf{GObj}_h(N)} \in \mathsf{GTol}_h(N)$, no change to the guard state is allowed, recapturing the requirement of an ordinary guard function. In view of (go-ii) and the fact that GObj_h is a guard function in the sense of 3.4, (tgp-iii) is equivalent to the following, simpler assertion.

(tgp-iii′) $((N_{|\mathsf{WSet}\langle\mathsf{WTrim}_N\langle\langle\mathbf{D},\mathcal{U}_T\rangle\rangle\rangle} = \mathsf{WTrim}_N\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle^{(1)})$
$\wedge\ ((\mathsf{WTrim}_N\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle^{(2)})_{|\mathsf{GObj}_h(N)} \in \mathsf{GTol}_h(N))$
$\Rightarrow (\mathsf{FLift}_{\mathbf{D}}\langle\langle \mathbf{D}, \mathcal{U}_T\rangle\rangle(N)\downarrow \in \mathsf{ELDB}(\mathbf{D})).$

Finally, (tgp-iv) corresponds to the similar condition of 3.4; the tolerant guard depends only upon the ground update, not upon how it was obtained.

It might seem that $\mathsf{GTol}_h(N)$ should be limited to databases in $\mathsf{ELDB}(\mathsf{GObj}_h(N))$, but it is harmless to allow those which do not satisfy the constraints, and that flexibility will prove to be of use.

The tolerant guard function h is *zero tolerance* if $\mathsf{GTol}_h(N) = \{N_{|\mathsf{GObj}_h(N)}\}$ for every $N \in \mathsf{ELDB}(\mathbf{D})$. In that case, h is effectively just an ordinary guard function, as defined in 3.4, since GTol_h has no additional effect.

The set of all tolerant guard functions for T is denoted $\mathsf{TolGuards}_{\mathbf{D}}\langle T\rangle$. A *tolerantly guarded black-box transaction* T is a pair $\langle\langle \mathbf{D}, \mathcal{U}_T\rangle, \mathcal{H}_T\rangle$ in which $\langle \mathbf{D}, \mathcal{U}_T\rangle \in \mathsf{FUpdObj}(\mathbf{D})$ and $\mathcal{H}_T \in \mathsf{TolGuards}_{\mathbf{D}}\langle T\rangle$. As a notational convenience, let $\mathcal{H}_T = \langle \mathcal{H}_T^{\mathsf{GFn}}, \mathcal{H}_T^{\mathsf{Tol}}\rangle$. In other words, $\mathcal{H}_T^{\mathsf{GFn}}$ is the function which assigns guard objects, while $\mathcal{H}_T^{\mathsf{Tol}}$ assigns tolerances. The set of all tolerantly guarded black-box transactions over $\mathbf{D}$ is denoted $\mathsf{TolGBBTrans}_{\mathbf{D}}$.

4.2 Examples of tolerant guard pairs and functions Return to the context of $\mathbf{E}_2$, particularly as developed in 3.9. For $i \in \{1, 2\}$, the ordinary guard function $\mathcal{G}_{T_i}$ is renamed $\mathcal{H}_{T_i}^{\mathsf{GFn}}$ in the tolerant setting. To obtain a tolerant guard which permits as much concurrency as possible, for any $n_1, n_2 \in \mathbb{Z}$, define $\mathcal{H}_{T_{2i}}^{\mathsf{Tol}}(M_{0\langle n_1,n_2\rangle}) = \{N \in \mathsf{DB}(x_{3-i}) \mid x_{3-i} > 1 - n_i\}$ if $n_1 + n_2 > 1$, with $\mathcal{H}_{T_{2i}}^{\mathsf{Tol}}(M_{0\langle n_1,n_2\rangle}) = \{\phi_{\mathsf{DB}}\}$ otherwise. For example, $\mathcal{H}_{T_{21}}^{\mathsf{Tol}}(M_{0\langle 3,2\rangle}) = \{N \in \mathsf{DB}(x_2) \mid x_2 > -2\}$, and $\mathcal{H}_{T_{22}}^{\mathsf{Tol}}(M_{0\langle 3,2\rangle}) = \{N \in \mathsf{DB}(x_1) \mid x_1 > -1\}$. In that case, T_{21} and T_{22} are each tolerant of the other for the state assignment of $M_{0\langle n_1,n_2\rangle}$ to each, and so they may execute concurrently. Indeed, T_{21} sets x_1 to 2, which is within the tolerance range of T_{22}, and T_{22} sets x_2 to 1, which is within the tolerance range of T_{21}.

To recapture an ordinary guard function within the tolerant framework, a tolerant guard of zero tolerance is used. Specifically, for any $n_1, n_2 \in \mathbb{Z}$, if $n_1 + n_2 > 1$, for $i \in \{1, 2\}$ define $\mathcal{H}_{T_{2i}}^{\mathsf{Tol}}(M_{0\langle n_1,n_2\rangle}) = \{(M_{0\langle n_1,n_2\rangle})_{|x_{3-i}}\}$. If $n_1 + n_2 \leq 1$, $\mathcal{H}_{T_{2i}}^{\mathsf{Tol}}(M_{0\langle n_1,n_2\rangle}) = \{\phi_{\mathsf{DB}}\}$ The effect of this tolerant guard is exactly the same as that of the ordinary guard, since no change of value of x_{3-i} is tolerated by T_{2i}.

4.3 Tolerance among transactions Let $\mathbf{T}' \subseteq \mathsf{TolGBBTrans}_{\mathbf{D}}$ and let $\iota : \mathbf{T}' \to \mathsf{LDB}(\mathbf{D})$ be a nonoverlapping state assignment for $\mathbf{T}'$. Given an ordered pair $\langle T_1, T_2\rangle \in \mathbf{T}' \times \mathbf{T}'$, ι is *tolerant* for $\langle T_1, T_2\rangle$ if $((\mathsf{FLift}_{\mathbf{D}}\langle \mathsf{WTrim}_{\iota(T_1)}\langle\langle \mathbf{D}, \mathcal{U}_{T_1}\rangle\rangle\rangle)^{(2)}(\iota(T_1))_{|\mathcal{H}_{T_2}^{\mathsf{GFn}}(\iota(T_2))} \in \mathcal{H}_{T_2}^{\mathsf{Tol}}(\iota(T_2))$.
In words, the result of the update of T_1, when restricted to the guard object of T_2, lies within the guard tolerance of T_2. For an unordered pair $\{T_1, T_2\} \subseteq \mathbf{T}'$ of distinct transactions, ι is *tolerant* for $\{T_1, T_2\}$ if it is tolerant for both $\langle T_1, T_2\rangle$ and $\langle T_2, T_1\rangle$.

4.4 The inadequacy of pairwise tolerance While it would be desirable to be able to formulate a constraint-preservation result based upon the pairwise property of 4.3, in a manner similar to the way in which 3.6 is used to establish 3.7, this is unfortunately not possible. To illustrate by example, let $k > 1$ be a positive integer, and let $\mathbf{E}_{3_k}$ be the database schema with k integer-valued data objects $\{x_i \mid i \in [1, k]\}$, constrained by $(\sum_{j=1}^{k} x_j) > 0$. Extending the notation of 3.9, for $n_1, n_2, \ldots, n_k \in \mathbb{Z}$, define $M_{3_k\langle n_1,n_2,\ldots,n_k\rangle} \in \mathsf{DB}(\mathbf{E}_{3_k})$ to have $x_i = n_i$ for $i \in [1, k]$. For $i \in [1, k]$, define $M^{\overline{x_i}}_{3_k\langle n_1,n_2,\ldots,n_k\rangle} \in \mathsf{DB}(\{x_j \mid j \in [1, k] \setminus \{i\}\})$ to be $M_{3_k\langle n_1,n_2,\ldots,n_k\rangle|\{x_j \mid j\in[1,k]\setminus\{i\}\}}$; *i.e.*, $M_{3_k\langle n_1,n_2,\ldots,n_k\rangle}$ with the component for x_i removed. Since n_i in the subscript is irrelevant, and may be written -.

For $i \in [1, k]$, define the transaction $T_{3_k i}$ by the conditional $\mathsf{if}\ (\sum_{j=1}^{k} x_j) > 1\ \mathsf{then}\ x_i \leftarrow x_i - 1$. If $(\sum_{j=1}^{k} n_j) > 1$, the guard object $\mathcal{H}_{T_{3_k i}}^{\mathsf{GFn}}(M_{3_k\langle n_1,n_2,\ldots,n_k\rangle}) = \{x_j \mid (j \in [1, k] \setminus \{i\})\}$, with the tolerance set $\mathcal{H}_{T_{3_k i}}^{\mathsf{Tol}}(M_{3_k\langle n_1,n_2,\ldots,n_k\rangle}) = \{M^{\overline{x_i}}_{3_k\langle m_1,m_2,\ldots,m_k\rangle} \mid (\sum_{j\in[1,k]\setminus\{i\}} m_j) > (2 - n_i)\}$.

If $(\sum_{j=1}^{k} n_j) \leq 1$, the identity update is performed, so $\mathcal{H}^{\mathsf{GFn}}_{T_{3_k i}}(M_{3_k\langle n_1,n_2,\ldots,n_k\rangle}) = \emptyset$ with $\mathcal{H}^{\mathsf{Tol}}_{T_{3_k i}}(M_{3_k\langle n_1,n_2,\ldots,n_k\rangle}) = \{\phi_{\mathsf{DB}}\}$.

For any integer p with $1 < p < k+1$, and integers $\langle n_1, n_2, \ldots, n_k\rangle$ satisfying $(\sum_{j=1}^{k} n_i) = p$, at most $p-1$ of the transactions in $\{T_{3i} \mid 1 \leq i \leq k\}$ may be run concurrently, using the same initial snapshot $M_{3_k\langle n_1,n_2,\ldots,n_k\rangle}$, without a constraint violation. If p or more are run concurrently, with that same initial snapshot, a constraint violation will result. This is true despite the fact that, if $p > 2$, then for any distinct pair $\{T_{j_1}, T_{j_2}\} \subseteq \{T_{3i} \mid 1 \leq i \leq k\}$ the state assignment which assigns $M_{3_k\langle n_1,n_2,\ldots,n_k\rangle}$ to each, is tolerant for $\{T_{j_1}, T_{j_2}\}$. Thus, the pairwise definition of tolerance of 4.3 is not adequate, in the general case, to characterize constraint preservation.

It is, however, possible to obtain a pairwise characterization of tolerance, provided that special conditions are imposed upon the structure of the guard object and guard tolerance. The main idea is to partition the schema $\mathbf{D}$ into complex data objects, called Π-objects, to require the each guard object be a union of Π-objects, and, most importantly, to require that each tolerance set be defined by a product of database states, one factor for each Π-object within the guard object. Although this limits somewhat how liberal the tolerance may be, far simpler tests suffice to determine whether an update by a transaction lies within the tolerances specified by the other concurrent transactions. The details constitute the remainder of this section.

4.5 Schema partitions and partition-compatible subobjects A *schema partition* for $\mathbf{D}$ is a partition Π on $\mathsf{DObj}\langle\mathbf{D}\rangle$. Each $\mathbf{x} \in \Pi$ is called a *block* of Π, with the set of all blocks of Π denoted $\mathsf{Blocks}\langle\Pi\rangle$. A subobject $\mathbf{x} \subseteq \mathsf{DObj}\langle\mathbf{D}\rangle$ is a Π-*object* if it is the union of some of the blocks of Π. In that case, $\mathsf{Blocks}_\Pi\langle\mathbf{x}\rangle$ denotes the set of blocks of which it is the union; *i.e.*, $\mathbf{x} = \bigcup \mathsf{Blocks}_\Pi\langle\mathbf{x}\rangle$. The set of all Π-objects of $\mathbf{D}$ is denoted $\mathsf{DObj}_\Pi\langle\mathbf{D}\rangle$.

The Π-*closure* of a data object $\mathbf{z} \subseteq \mathsf{DObj}\langle\mathbf{D}\rangle$ is the smallest Π-object containing $\mathbf{z}$. Formally, $\mathsf{Closure}^\Pi\langle\mathbf{z}\rangle = \bigcup\{\mathbf{x} \in \mathsf{Blocks}_\Pi\langle\Pi\rangle \mid \mathbf{x} \cap \mathbf{z} \neq \emptyset\}$. Given an updateable object $\langle\mathbf{c}, \mathbf{u}\rangle$, $\mathsf{WSet}^\Pi\langle\langle\mathbf{c}, \mathbf{u}\rangle\rangle$ denotes $\mathsf{Closure}^\Pi\langle\mathsf{WSet}\langle\langle\mathbf{c}, \mathbf{u}\rangle\rangle\rangle$. For $N \in \mathsf{LDB}(\mathbf{D})$, the Π-*closed write trim* of $\langle\mathbf{c}, \mathbf{u}\rangle$ to N, denoted $\mathsf{WTrim}^\Pi_N\langle\langle\mathbf{c}, \mathbf{u}\rangle\rangle$, is $\langle\mathbf{c}, \mathbf{u}\rangle_{|\mathsf{WSet}^\Pi\langle\mathsf{Trim}_N\langle\langle\mathbf{c},\mathbf{u}\rangle\rangle\rangle}$. It is the smallest Π-object containing $\mathsf{WSet}^\Pi\langle\mathsf{Trim}_N\langle\langle\mathbf{c}, \mathbf{u}\rangle\rangle\rangle$; that is, the smallest Π-object which contains the write set of $\langle\mathbf{c}, \mathbf{u}\rangle$ when trimmed to N.

4.6 Product sets of states Let Π be a schema partition for $\mathbf{D}$, and let $\mathbf{x} \in \mathsf{DObj}_\Pi\langle\mathbf{D}\rangle$. A subset $\mathbf{M} \subseteq \mathsf{DB}(\mathbf{x})$ is a Π-*product set* for $\mathbf{x}$ if there is a $\mathsf{Blocks}\langle\Pi\rangle$-indexed family $\{\mathbf{M_z} \subseteq \mathsf{DObj}\langle\mathbf{x}\rangle \mid \mathbf{z} \in \mathsf{Blocks}\langle\Pi\rangle\}$ such that for any $N \in \mathsf{DB}(\mathbf{x})$, $N \in \mathbf{M}$ iff $N_{|\mathbf{z}} \in \mathbf{M_z}$ for every $\mathbf{z} \in \mathsf{Blocks}\langle\Pi\rangle$. In other words, a Π-product set for $\mathbf{x}$ is a product of sets, one for each $\mathbf{z} \in \mathsf{Blocks}_\Pi\langle\mathbf{x}\rangle$.

4.7 Π-compatible guard objects and functions Using the concepts surrounding schema partitions, a definition of tolerant guard function which will admit a pairwise characterization of independence may be made. Formally, given a schema partition Π for $\mathbf{D}$, a tolerant guard function $h = \langle\mathsf{GObj}_h, \mathsf{GTol}_h\rangle$ for T is Π-*compatible* if the following three conditions are satisfied.

(Π-tg-i) For each $N \in \mathsf{ELDB}(\mathbf{D})$,

$$\mathsf{Blocks}_\Pi\langle\mathsf{WSet}^\Pi\langle\mathsf{Trim}_{\iota(T_1)}\langle\langle\mathbf{D}, \mathcal{U}_{T_1}\rangle\rangle\rangle\rangle \cap \mathsf{Blocks}_\Pi\langle\mathsf{GObj}_h(N)\rangle = \emptyset.$$

(Π-tg-ii) For each $N \in \mathsf{ELDB}(\mathbf{D})$, $\mathsf{GObj}_h(N) \in \mathsf{DObj}_\Pi\langle\mathbf{D}\rangle$.

(Π-tg-iii) $\mathsf{GTol}_h(N)$ is a Π-product set for $\mathsf{GObj}_h(N)$.

Condition (Π-tg-i) extends (go-i) of 3.4 by requiring that the Π-closure of the write set not intersect the guard. (Π-tg-ii) and (Π-tg-iii) mandate that the guard object be Π-compatible and the tolerance be a Π-product set, respectively.

The tolerant guard function h is Π-minimal just in case the guard object cannot be reduced in size, and the tolerance cannot be increased in size, while still retaining the property of a guard. Formally, call h Π-*minimal* if for every $N \in \mathsf{ELDB}(\mathbf{D})$, no proper Π-compatible subset $H \subsetneq \mathsf{GObj}_h(N)$ is a guard object for T, and for no proper Π-product superset $\mathsf{GTol}_h(N) \subsetneq \mathbf{P}$ is condition (tgp-iii) of 4.1 satisfied when, in that formula, $\mathsf{GTol}_h(N)$ is replaced by $\mathbf{P}$.

A tolerantly guarded transaction $T \in \mathsf{TolGBBTrans}_\mathbf{D}$ is called Π-*compatible* if $\mathcal{H}_T$ has that property. The set of all Π-compatible tolerantly guarded transactions is denoted $\Pi\text{-}\mathsf{TolGBBTrans}_\mathbf{D}$.

Extending the definition of 3.3 to the Π-compatible setting, for $\mathbf{T}' \subseteq \Pi\text{-}\mathsf{TolGBBTrans}_\mathbf{D}$, call a state assignment $\iota : \mathbf{T}' \to \mathsf{ELDB}(\mathbf{D})$ Π-*nonoverlapping* for if for every $T_1, T_2 \in \mathbf{T}'$ with $T_1 \neq T_2$, $\mathsf{WSet}^\Pi\langle\mathsf{Trim}_{\iota(T_1)}\langle\langle\mathbf{D}, \mathcal{U}_{T_1}\rangle\rangle\rangle \cap \mathsf{WSet}^\Pi\langle\mathsf{Trim}_{\iota(T_2)}\langle\langle\mathbf{D}, \mathcal{U}_{T_2}\rangle\rangle\rangle = \emptyset$.

Extending the definition of 4.3 to the Π-compatible setting, given an ordered pair $\langle T_1, T_2\rangle \in \mathbf{T}' \times \mathbf{T}'$ and ι Π-nonoverlapping as above, ι is Π-*tolerant for* $\langle T_1, T_2\rangle$ if, for each

$$\mathbf{x} \in \mathsf{Blocks}_\Pi\langle\mathsf{WSet}^\Pi\langle\mathsf{Trim}_{\iota(T_1)}\langle\langle\mathbf{D}, \mathcal{U}_{T_1}\rangle\rangle\rangle\rangle \cap \mathsf{Blocks}_\Pi\langle\mathcal{H}^{\mathsf{GFn}}_{T_2}(\iota(T_2))\rangle,$$

it is the case that $((\mathsf{FLift}_\mathbf{D}\langle\mathsf{WTrim}^\Pi_{\iota(T_1)}\langle\langle\mathbf{D}, \mathcal{U}_{T_1}\rangle\rangle\rangle))^{(2)}(\iota(T_1))_{|\mathbf{x}} \in \mathcal{H}^{\mathsf{Tol}}_{T_2}(\iota(T_2))_{|\mathbf{x}}$. Finally, ι is Π-tolerant for $\{T_1, T_2\}$ if is Π-tolerant for both (T_1, T_2) and (T_2, T_1).

4.8 Notational convention Extending 3.5, from now on, unless stated explicitly to the contrary, assume that a partition Π of $\mathsf{DObj}\langle\mathbf{D}\rangle$ is fixed, and that $\mathbf{T}$ is a finite subset of $\Pi\text{-}\mathsf{TolGBBTrans}_\mathbf{D}$. Assume further that the guard for transaction T is denoted $\mathcal{H}_T = \langle\mathcal{H}^{\mathsf{GFn}}_T, \mathcal{H}^{\mathsf{Tol}}_T\rangle$.

4.9 Theorem — constraint preservation under tolerance *Let $\leq_\mathbf{T}$ be an SI-schedule for $\mathbf{T}$ and let $M \in \mathsf{ELDB}(\mathbf{D})$. If $\mathsf{StAssign}_{\langle\leq_\mathbf{T}:\, M\rangle}$ is Π-nonoverlapping and $\{T_1, T_2\}$-tolerant for Π for every pair of distinct concurrent transactions $\{T_1, T_2\} \subseteq \mathbf{T}$, then $\leq_\mathbf{T}$ is constraint preserving for initial state M.*

Proof: The proof is by induction on the number of transactions in $\mathbf{T}$. Let T_i represent the i^{th} transaction which commits; for n transactions, the commit order is therefore $T_1, T_2, \ldots, T_{n-1}, T_n$. The basis step of the induction, for $n \in \{0, 1\}$, is trivial. For the inductive step, let $n > 1$ and assume that the result is true whenever the number of transactions in $\mathbf{T}$ is no more than n, and consider the case that $\mathbf{T}$ consists of $n+1$ transactions. Upon removing T_{n+1}, the schedule consisting of the transactions in $\mathbf{T} \setminus \{T_{n+1}\}$ is constraint preserving for M by the inductive hypothesis. To verify constraint preservation for the entire set $\mathbf{T}$, it suffices to verify that committing the writes of T_{n+1} to $\mathsf{AfterCmt}_{\langle\leq_\mathbf{T}:\, M\rangle}\langle T_n\rangle$ does not violate any integrity constraints. This is guaranteed by the requirement that, for every T_i with the property that T_i and T_{n+1} run concurrently, $\{T_i, T_{n+1}\}$ form Π-tolerant pair for $\mathsf{StAssign}_{\langle\leq_\mathbf{T}:\, M\rangle}$. If the Π-closure of the update set of T_i overlaps any part of the guard tolerance T_{n+1}, then for each $\mathbf{x} \in \mathsf{Blocks}\langle\Pi\rangle$ which lies in both, the update by T_{n+1}, restricted to $\mathbf{x}$ is guaranteed to lie within the tolerance

set $\mathcal{H}^{\mathsf{Tol}}_{T_{n+1}}(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}}: M\rangle}\langle T_{n+1}\rangle)$ of T_{n+1}, as stipulated in 4.3. The key point is that (in contrast to the general case, as illustrated in 4.4), the actions of other transactions, even concurrent ones, cannot change this. Thus, the pairwise check for tolerance suffices. □

4.10 Examples of constraint preservation under tolerance Continue with the setting and examples of 4.4, specifically the schema $\mathbf{E}_{3_k}$ for some $k > 3$. Let the partition $\Pi_{\mathbf{E}_{3_k}}$ have each data object of $\{x_j \mid j \in [1,k]\}$ in its own block. It is useful to extend the notation of 4.4 for states to $\Pi_{\mathbf{E}_{3_k}}$-product sets. To this end, define

$$\mathbf{M}_{3_k\langle n_1,n_2,\ldots,n_k\rangle} = \{M_{3_k\langle m_1,m_2,\ldots,m_k\rangle} \in \mathsf{DB}(\mathbf{E}_{3_k}) \mid (\forall j \in [1,k])(m_i \geq n_i)\},$$

the product set in which, for each database, the value of x_i is at least as large as n_i. Similarly, define

$$\mathbf{M}^{\overline{x_i}}_{3_k\langle n_1,n_2,\ldots,n_k\rangle} = \{M^{\overline{x_i}}_{3_k\langle m_1,m_2,\ldots,m_k\rangle} \in \mathsf{DB}(\mathbf{E}_{3_k}) \mid (\forall j \in [1,k])(m_i \geq n_i)\}.$$

Take the initial state to be $M_{3_k\langle p_1,p_2,\ldots,p_k\rangle}$ with $(\sum_{j\in[1,k]} p_i) = p$, for some integer p with $0 < p < k+1$. As argued in 4.4, exactly $p-1$ of the transactions in $\{T_{3i} \mid 1 \leq i \leq k\}$ may be run concurrently with initial snapshot $M_{3\langle p_1,p_2,\ldots,p_k\rangle}$. However, in contrast to the general setting, in the setting of partition compatibility, which $p+1$ transactions may run depends upon how tolerance sets are chosen. Consider the transaction $T_{3_k i}$. In the $\Pi_{\mathbf{E}_{3_k}}$-compatible context, while $\mathcal{H}^{\mathsf{GFn}}_{T_{3_k i}}(M_{3_k\langle p_1,p_2,\ldots,p_k\rangle}) = \{x_j \mid j \in [1,k] \setminus \{i\}\}$, as in 4.4, it is no longer permissible to define its guard tolerance via $(\sum_{j=2}^{k} x_j) > 1$ alone, since a $\Pi_{\mathbf{E}_{3_k}}$-product set must be chosen for the guard. A set of the form $\mathbf{M}^{\overline{x_1}}_{3_k\langle \text{-},n_2,\ldots,n_k\rangle}$ must have $(\sum_{i\in[2,k]} n_i) > 2 - p_1$ (in order to allow the update $x_1 \leftarrow x_1 - 1$ to occur without violating the integrity constraint), as well as $n_i \leq p_i$ for $i \in [2,k]$ (since the current value of x_i must always be present in the range).

To keep things concrete, choose $k = 3$, with initial snapshot $M_{3_3\langle 1,1,1\rangle}$. For transaction $T_{3_3 1}$, a minimal Π_{3_4}-compatible guard is of the form $\mathbf{M}^{\overline{x_1}}_{3_3\langle \text{-},q_2,q_3\rangle}$ with $q_2 + q_3 > 1$ and both $q_2 \leq 1$ and $q_3 \leq 1$. This means that the only two possibilities are $\mathbf{M}^{\overline{x_1}}_{3_3\langle \text{-},0,1\rangle}$ and $\mathbf{M}^{\overline{x_1}}_{3_3\langle \text{-},1,0\rangle}$. If $\mathbf{M}^{\overline{x_1}}_{3_3\langle \text{-},0,1\rangle}$ is chosen, then $T_{3_3 3}$ may not run concurrently, since $x_2 \geq 1$ is required by $T_{3_3 1}$. Similarly, if $\mathbf{M}^{\overline{x_1}}_{3_3\langle \text{-},1,0\rangle}$ is chosen, then $T_{3_3 2}$ may not run concurrently. Thus, $T_{3_3 1}$ must in effect choose a "victim" which cannot run concurrently. Suppose that victim is $T_{3_2 3}$, and $T_{3_3 2}$ attempts to run concurrently with $T_{3_2 1}$. It will succeed only if it chooses its tolerance correctly. Specifically, it must choose $\mathbf{M}^{\overline{x_1}}_{3_3\langle 0,\text{-},1\rangle}$. If it chooses $\mathbf{M}^{\overline{x_1}}_{3_3\langle 1,\text{-},0\rangle}$, its update will not lie in the tolerance specified by $T_{3_2 1}$. This illustrates that transactions should be given the opportunity to know the guard functions of other concurrent transactions, and to choose their own guard functions to ensure success. This idea is explored more thoroughly in Sec. 5.

Although Π-compatibility may seem limiting, two points should be kept in mind. First, constraints which tie many data objects together are relatively uncommon. For the most part, constraints relate just two data objects, The examples shown here are specifically designed to show the theoretical limitations. Second, these problems only occur in border cases. For example, if the initial state is $M_{3_3\langle 2,2,2\rangle}$, then all three transactions may execute concurrently, without any problem, provided the guards are chosen reasonably.

4.11 Tolerant CPSI — TPCSI By *tolerant CPSI*, or *TCPSI* for short, is meant the strategy described in this section, with Π-compatibility.

5 An Operational Description of TCPSI

As noted in 4.10, under TCPSI, there is an advantage in allowing transactions to be aware of the actions of each other, in order to choose guards dynamically. To this end, an operational version of TCPSI is described, in the spirit of FUW (first-updater wins) of SI [10, Sum. 2.3], which allows such dynamic choices.

5.1 Declaration-augmented SI schedules and transactions To begin, the simple model of SI, as and summarized in 3.1, is extended to a third time point of each transaction, the *declaration time*. It is at this time that a transaction declares its (proposed) update as well as its guard pair to the system (and to the other transactions), and a check is made to determine whether these declarations are consistent with the current global database state, as well as the declarations of the other, concurrent transactions which have already declared. Formally, given $\mathbf{T} \subseteq \mathsf{BBTrans}_{\mathbf{D}}$, define $\mathsf{PSCSet}\langle\mathbf{T}\rangle = \mathsf{SCSet}\langle\mathbf{T}\rangle \cup \{T^d \mid T \in \mathbf{T}\} = \{T^s \mid T \in \mathbf{T}\} \cup \{T^d \mid T \in \mathbf{T}\} \cup \{T^c \mid T \in \mathbf{T}\}$. A *declaration-augmented SI-schedule* on $\mathbf{T}$ is a total order $\leq_{\mathbf{T}}$ on $\mathsf{PSCSet}\langle T\rangle$ with the property that for each $T \in \mathbf{T}$, $T^s <_{\mathbf{T}} T^d <_{\mathbf{T}} T^c$. T^d represents the time at which T *declares*.

A transaction which declares has the same black-box representation, as given in 4.1, as one which does not declare. Thus, the conditions for constraint preservation, as presented in 4.9, apply equally well to transactions which declare. However, the interpretation is a bit different. In a transaction T which declares, the update $\langle\mathbf{D}, \mathcal{U}_T\rangle$ and the guard pair $\mathcal{H}_T = \langle\mathcal{H}_T^{\mathsf{GFn}}, \mathcal{H}_T^{\mathsf{Tol}}\rangle$ need not be specified when the transaction begins; rather, they may be determined at declaration time, allowing T to make real-time decisions about the choice of guard and even the choice of update. To highlight this difference, transactions with an explicit declaration point will be termed *grey box*. The set of all tolerantly guarded Π-compatible grey-box transactions over $\mathbf{D}$ is denoted $\Pi\text{-}\mathsf{TolGBBTrans}_{\mathbf{D}}$.

5.2 Notational convention In addition to the conventions of 3.5 and 4.8, from now on, also assume that $\leq_{\mathbf{T}}$ is a declaration-augmented SI schedule, and that $\mathbf{T}$ is a finite subset of $\Pi\text{-}\mathsf{TolGBBTrans}_{\mathbf{D}}$.

5.3 Global values for declarations The steps which are taken during the execution of a declaration-augmented SI schedule are based upon the values of certain objects which evolve during the execution. Each object listed below has a value at each point in time during the lifetime of the schedule $\leq_{\mathbf{T}}$.

$\mathsf{ActiveTrans}_{\langle\leq_{\mathbf{T}}:M\rangle}$ is the set of all transactions which are *active*; that is, which have started but have not yet committed.

$\mathsf{DeclTrans}_{\langle\leq_{\mathbf{T}}:M\rangle}$ is the set of all active transactions which have declared.

The remaining three object families have values which are defined at each point in time for each $\mathbf{z} \in \mathsf{Blocks}\langle\Pi\rangle$.

$\mathsf{CmtVal}^{\mathbf{z}}_{\langle\leq_{\mathbf{T}}:M\rangle}$ is the current committed value of data object $\mathbf{z}$. It is already present in every system as a record or set of records in the global database. However, it is convenient to have this explicit notation for it.

$\mathsf{PndVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}$ is the value, if any, which a transaction $T \in \mathsf{DeclTrans}_{\langle \leq_{\mathbf{T}}:M\rangle}$ has proposed as an update via a declaration (see 5.5 below). It may be realized as a link to a record or records in the snapshot of T. If no running transaction has proposed an update to $\mathbf{x}$, the value is that of $\mathsf{CmtVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}$.

$\mathsf{CurrTol}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}$ is defined to be

$$\bigcap\{(\mathcal{H}^{\mathsf{Tol}}_{T'}(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}}:M\rangle}\langle T'\rangle))_{|\mathbf{z}} \mid (\mathbf{z} \in \mathcal{H}^{\mathsf{GFn}}_{T}(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}}:M\rangle}\langle T'\rangle)) \wedge (T' \in \mathsf{DeclTrans}_{\langle \leq_{\mathbf{T}}:M\rangle})\}.$$

It expresses the combined tolerance on $\mathbf{z}$ of all transactions which have declared updates. If there is no active transaction T with $\mathbf{z} \in \mathcal{H}^{\mathsf{GFn}}_{T}(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}}:M\rangle}\langle T\rangle)$, the value of $\mathsf{CurrTol}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}$ is $\mathsf{DB}(\mathbf{z})$.

It should be noted that an explicit value need not be stored for each such $\mathbf{z}$, any more than an initial snapshot under SI contains an explicit record for each data object. Rather, in the above, explicit values for $\mathbf{z} \in \mathsf{Blocks}\langle\Pi\rangle$ are only required in the case that some active transaction which has declared uses that data object, either as a writer or else in a guard.

5.4 Transaction-specific instances of global values For each transaction, the values of the last three objects listed in 5.3, at the point of declaration, are central to the process. A transaction cannot change the value of $\mathsf{CmtVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}$ until it commits, so a single value at the point of declaration suffices. Since the transaction may alter $\mathsf{PndVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}$ and $\mathsf{CurrTol}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}$ when it declares, separate before and after values are necessary. The formal definitions follow. Each applies for each $T \in \mathbf{T}$ and each $\mathbf{z} \in \mathsf{Blocks}\langle\Pi\rangle$.

$\mathsf{PrCmtVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}\langle T\rangle$: the value of $\mathsf{CmtVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}$ at T^d.

$\mathsf{BeforePrPndVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}\langle T\rangle$: the value of $\mathsf{PndVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}$ immediately before T^d.

$\mathsf{AfterPrPndVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}\langle T\rangle$: the value of $\mathsf{PndVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}$ immediately after T^d.

$\mathsf{BeforePrTolVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}\langle T\rangle$: the value of $\mathsf{CurrTol}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}$ immediately before T^d.

$\mathsf{AfterPrTolVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}\langle T\rangle$: the value of $\mathsf{CurrTol}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle}$ immediately after T^d.

5.5 Pending-update maintenance under SI The steps (pum-i)-(pum-ix), identified below, when applied to the schedule $\leq_{\mathbf{T}}$ with initial database $M \in \mathsf{ELDB}(\mathbf{D})$, are called collectively *pending-update maintenance*.

Schedule initialization: At the beginning of the execution of $\leq_{\mathbf{T}}$ with initial database M, for each $\mathbf{z} \in \mathsf{Blocks}_{\Pi}\langle\Pi\rangle$, the following three assignments are executed, in order to give the objects the appropriate initial values.

(pum-i) $\mathsf{CmtVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle} \leftarrow M_{|\mathbf{z}}$.
(pum-ii) $\mathsf{PndVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle} \leftarrow M_{|\mathbf{z}}$.
(pum-iii) $\mathsf{CurrTol}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}}:M\rangle} \leftarrow \mathsf{DB}(\mathbf{z})$.

Transaction declaration: When a transaction T declares (at T^d), the following assignments are executed, provided the conditions (decl-i) to (decl-iv) of 5.6 below are satisfied. If

those conditions are not satisfied, the transaction may either wait until the conditions are satisfied, or else abort.

(pum-iv) $\mathsf{DeclTrans}_{\langle \leq_{\mathbf{T}} : M\rangle} \leftarrow \mathsf{DeclTrans}_{\langle \leq_{\mathbf{T}} : M\rangle} \cup \{T\}$.

(pum-v) For each $\mathbf{z} \in \mathsf{Blocks}\langle \Pi\rangle \cap \mathsf{WSet}\langle \mathsf{WTrim}^{\Pi}_{\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle}\langle\langle \mathbf{D}, \mathcal{U}_T\rangle(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle)\rangle\rangle$, execute

$$\mathsf{PndVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}} : M\rangle} \leftarrow (\mathsf{WTrim}^{\Pi}_{\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle}\langle\langle \mathbf{D}, \mathcal{U}_T\rangle(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle)\rangle^{(2)})_{|\mathbf{z}}.$$

(pum-vi) For each $\mathbf{z} \in \mathsf{Blocks}_{\Pi}\langle \mathcal{H}^{\mathsf{GFn}}(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle)\rangle$, execute

$$\mathsf{CurrTol}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}} : M\rangle} \leftarrow \mathsf{CurrTol}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}} : M\rangle} \cap \mathcal{H}^{\mathsf{Tol}}(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle)_{|\mathbf{z}}.$$

Action (pum-v) sets the pending value $\mathsf{PndVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}} : M\rangle}$ of each data object $\mathbf{z}$ which is updated by T to the new, updated value, while (pum-vi) adds the tolerance for that update to $\mathsf{CurrTol}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}} : M\rangle}$.

Transaction commit: For any $T \in \mathbf{T}$, when T commits, the following three assignments are executed.

(pum-vii) $\mathsf{DeclTrans}_{\langle \leq_{\mathbf{T}} : M\rangle} \leftarrow \mathsf{DeclTrans}_{\langle \leq_{\mathbf{T}} : M\rangle} \setminus \{T\}$.

(pum-viii) For each $\mathbf{z} \in \mathsf{WSet}\langle \mathsf{WTrim}^{\Pi}_{\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle}\langle\langle \mathbf{D}, \mathcal{U}_T\rangle(M)\rangle\rangle$, execute

$$\mathsf{CmtVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}} : M\rangle} \leftarrow \mathsf{PndVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}} : M\rangle}.$$

(pum-ix) For each $\mathbf{z} \in \mathsf{Blocks}\langle \Pi\rangle \cap \mathcal{H}^{\mathsf{GFn}}_{T}(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle)$, update $\mathsf{CurrTol}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}} : M\rangle}$ to reflect that T is no longer active nor declared.

Step (pum-viii) commits the pending update to the global database; (pum-ix) removes the tolerance required by transaction T, since it is now finished.

5.6 Tolerance-compliant and nonoverlapping schedules The following four conditions must be verified for each $T \in \mathbf{T}$, at T^d.

(decl-i) $(\forall \mathbf{z} \in \mathsf{Blocks}\langle \Pi\rangle \cap \mathsf{WSet}\langle \mathsf{WTrim}^{\Pi}_{\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle}\langle\langle \mathbf{D}, \mathcal{U}_T\rangle(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle)\rangle\rangle)$

$$(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle_{|\mathbf{z}} = \mathsf{BeforePrPndVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle = \mathsf{PrCmtVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle).$$

(decl-ii) $(\forall \mathbf{z} \in \mathsf{Blocks}\langle \Pi\rangle \cap \mathcal{H}^{\mathsf{GFn}}(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle))$

$$((\mathsf{CmtVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}} : M\rangle} \neq (\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle)_{|\mathbf{z}}) \Rightarrow (\mathsf{CmtVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}} : M\rangle} \in \mathcal{H}^{\mathsf{Tol}}(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle)_{|\mathbf{z}})).$$

(decl-iii) $(\forall \mathbf{z} \in \mathsf{Blocks}\langle \Pi\rangle \cap \mathcal{H}^{\mathsf{GFn}}(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle))$

$$((\mathsf{PndVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}} : M\rangle} \neq (\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle \mathbf{z}\rangle)_{|\mathbf{z}}) \Rightarrow (\mathsf{PndVal}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}} : M\rangle} \in \mathcal{H}^{\mathsf{Tol}}(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle)_{|\mathbf{z}})),$$

(decl-iv) $(\forall \mathbf{z} \in \mathsf{Blocks}\langle \Pi\rangle \cap \mathsf{WSet}\langle \mathsf{WTrim}^{\Pi}_{\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle}\langle\langle \mathbf{D}, \mathcal{U}_T\rangle(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle)\rangle\rangle)$

$$((\mathsf{WTrim}^{\Pi}_{\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle}\langle\langle \mathbf{D}, \mathcal{U}_T\rangle(\mathsf{InitSnap}_{\langle \leq_{\mathbf{T}} : M\rangle}\langle T\rangle)\rangle^{(2)})_{|\mathbf{z}} \in \mathsf{CurrTol}^{\mathbf{z}}_{\langle \leq_{\mathbf{T}} : M\rangle}).$$

Condition (decl-i) checks for write overlap; this test must be performed for any implementation of SI, with or without constraint preservation. Testing at declaration time is nothing more than FUW (first updater wins) [10, Sum. 2.3] — a transaction T is allowed to continue if its

declared update does not conflict with those updates by other transactions which have been declared since T started.

The remaining three tests concern constraint preservation. When transaction T declares, it must be established that for each concurrent transaction T' which has already declared, the update of T' does not violate the tolerance of T. (decl-ii) verifies this in the case that T' has already committed, while (decl-iii) serves the same purpose in the case that T' has declared but not yet committed. (decl-iv) verifies that no read-write conflict exists in the opposite direction; it checks whether the proposed writes of T violate the tolerance limits of the other declared but not yet committed transactions.

5.7 Example An example of operational TCPSI, as described in this section, is summarized in Table 1. It uses the schema $\mathbf{E}_{3_3}$, as well as the associated transactions and notation, described in 4.4 and 4.10, On transactions, 3_3 is omitted in the subscript; thus, for $1 \in [1,3]$, $T_{3_3 i}$ becomes just T_i. The transaction S_2 is new; it executes the update $x_2 \leftarrow x_2 + 1$ unconditionally. Its guard object is always $\emptyset$, and its tolerance always $\{\phi_{\mathsf{DB}}\}$; it can never induce a constraint violation when run in isolation, since increasing the value of x_2 can never decrease the value of a sum of which it is a summand. The initial state is $M_{3_3\langle 1,1,1\rangle}$. In the first two lines of the table,

	x_1			x_2			x_3			$\geq$
Tr	CmtVal	PndVal	CurrTol	CmtVal	PndVal	CurrTol	CmtVal	PndVal	CurrTol	TrTol
T_1^s	1	1	-	1	1	-	1	1	-	
T_2^s	1	1	-	1	1	-	1	1	-	
T_1^d	1	0	-	1	1	≥ 0	1	1	≥ 1	(-, 0, 1)
T_2^d	1	0	≥ 0	1	0	≥ 0	1	1	≥ 1	(0, -, 1)
T_3^s	1	0	≥ 0	1	0	≥ 0	1	1	≥ 1	
T_3^d	1	0	≥ 0	1	0	≥ 0	1	1	≥ 1	blocked
T_2^c	1	0	-	0	0	≥ 0	1	1	≥ 1	
S_2^s	1	0	-	0	0	≥ 0	1	1	≥ 1	
S_2^d	1	0	-	0	1	≥ 0	1	1	≥ 1	(-, -, -)
S_2^c	1	0	-	1	1	≥ 0	1	1	≥ 1	
$[T_3^d]$	1	0	≥ 0	1	1	≥ 1	1	0	≥ 1	(0, 1, -)
T_1^c	0	0	-	1	1	≥ 1	1	0	-	
T_3^c	0	0	-	1	1	-	0	0	-	

Table 1: Tabular summary of the example of 5.7

T_1 and T_2 begin, in that order. Note that the CmtVal and the PndVal for each x_i carries the values of the initial state. Next, T_1 declares its intent to execute the ground update $1 \overset{x_1}{\rightsquigarrow} 0$ (see 2.6) using the tolerance $\mathbf{M}^{\overline{x_1}}_{3_3\langle \text{-},0,1\rangle}$. Note that PndVal^{x_1} has been set to zero, but that CmtVal^{x_2} remains unchanged. In the next step, T_2 declares its intent to execute the ground update $1 \overset{x_2}{\rightsquigarrow} 0$. For this not to conflict with T_1, the tolerance $\mathbf{M}^{\overline{x_2}}_{3_3\langle 0,\text{-},1\rangle}$ is chosen. Had $\mathbf{M}^{\overline{x_2}}_{3_3\langle 1,\text{-},0\rangle}$ been chosen instead, T_1 and T_2 could not both commit without a constraint violation. This illustrates the desirability of T_2 choosing its guard tolerance interactively, with knowledge of the pending updates of other, concurrent transactions. In the next two lines, T_3 begins and declares. However, it is blocked by the combination of T_1 and T_2; there is no choice of tolerance on x_1 and x_2 which would allow it to run. At this point, it can either abort or wait; assume that it waits. Next, T_2 commits. This sets CmtVal^{x_2} to 0 and the $\mathsf{CurrTol}^{x_1}$ and $\mathsf{CurrTol}^{x_3}$ constraints

which it applied are removed. There is now no constraint on $\mathsf{CurrTol}^{x_1}$, but the constraint on $\mathsf{CurrTol}^{x_3}$ remains since T_1 requires it also. Next, S_2 starts, declares, and commits, executing the update $0 \overset{x_2}{\rightsquigarrow} 1$. No changes are made to the value of any $\mathsf{CurrTol}^{x_i}$, since the guard object is $\emptyset$ for this transaction. The waiting T_3 may now continue, asserting the tolerance $\mathbf{M}^{\overline{x_3}}_{3_3\langle 0,1,\text{-}\rangle}$, since the value of x_2 has increased to 1. Its second declaration point is shown as $[T_3^d]$. This illustrates the advantage of a transaction which is blocked at declaration to wait; in an interactive setting this is possible. Finally, T_1 and then T_3 commit, to complete the execution of the schedule.

5.8 Extended and multiple declaration As illustrated in 5.7, it is advantageous to allow a transaction to determine its declarations later than at the very beginning of execution, and to wait if conditions are not suitable. It is furthermore advantageous to allow it to make different parts of its declarations at different times, and even to alter its declarations. Indeed, this approach has the potential to blend well with cooperative update [8], [11]. The extension is straightforward, but space limitations preclude a further elaboration here.

6 Conclusions and Further Directions

The ideas of CPSI have been extended to the case in which one transaction may write the guard of another while preserving satisfaction of integrity constraints. By using a value-oriented model, in which transactions announce a tolerance on updates to their read sets by other transactions, significantly greater concurrency is possible. In addition to an abstract model, a more operational model has been presented as well. There are at least two key areas for further work.

Extension to concurrent writes: The extension of CPSI presented here supports updates to read sets, but does not allow concurrent writes. An approach which supports write concurrency under certain conditions is the next step in the theoretical development.

Prototype implementation: A prototype implementation is essential, especially to evaluate data structures and algorithms for the management of the declaration phase. The preferable platform would be to build upon an existing open-source systems, such as PostgreSQL or MariaDB.

References

[1] Bancilhon, F., Kim, W., Korth, H.F.: A model of CAD transactions. In: A. Pirotte, Y. Vassiliou (eds.) VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, August 21-23, 1985, Stockholm, Sweden, pp. 25–33. Morgan Kaufmann (1985)

[2] Berenson, H., Bernstein, P.A., Gray, J., Melton, J., O'Neil, E.J., O'Neil, P.E.: A critique of ANSI SQL isolation levels. In: Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995, pp. 1–10 (1995)

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

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

[5] Fekete, A., Liarokapis, D., O'Neil, E.J., O'Neil, P.E., Shasha, D.: Making snapshot isolation serializable. ACM Trans. Database Syst. **30**(2), 492–528 (2005)

[6] Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)

[7] Härder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. **15**(4), 287–317 (1983)

[8] Hegner, S.J.: A simple model of negotiation for cooperative updates on database schema components. In: Y. Kiyoki, T. Tokuda, A. Heimbürger, H. Jaakkola, N. Yoshida. (eds.) Frontiers in Artificial Intelligence and Applications XX11, pp. 154–173. IOS Press (2011)

[9] Hegner, S.J.: Guard independence and constraint-preserving snapshot isolation. In: C. Bierle, C. Meghini (eds.) Foundations of Information and Knowledge Systems: Eighth International Symposium, FoIKS 2014, Bordeaux, France, March 3-7, 2014, Proceedings, *Lecture Notes in Computer Science*, vol. 8367, pp. 231–250. Springer-Verlag (2014)

[10] Hegner, S.J.: Constraint-preserving snapshot isolation. Ann. Math. Art. Intell. **76**(3), 281–326 (2016)

[11] Hegner, S.J., Schmidt, P.: Update support for database views via cooperation. In: Y. Ioannis, B. Novikov, B. Rachev (eds.) Advances in Databases and Information Systems, 11th East European Conference, ADBIS 2007, Varna, Bulgaria, September 29 - October 3, 2007, Proceedings, *Lecture Notes in Computer Science*, vol. 4690, pp. 98–113. Springer-Verlag (2007)

[12] Korth, H.F., Speegle, G.D.: Formal aspects of concurrency control in long-duration transaction systems using the NT/PV model. ACM Trans. Database Syst. **19**(3), 492–535 (1994)

[13] Papadimitriou, C.: The Theory of Database Concurrency Control. Computer Science Press (1986)

[14] Sippu, S., Soisalon-Soininen, E.: Transaction Processing: Management of the Logical Database and its Underlying Physical Structure. Springer (2014)

[15] Weikum, G., Vossen, G.: Transactional Information Systems. Morgan Kaufmann (2002)