Robustness Against Read Committed for Transaction Templates with Functional Constraints

Brecht Vandevoort
UHasselt, Data Science Institute, ACSL, Diepenbeek, Belgium

Bas Ketsman
Vrije Universiteit Brussel, Belgium

Christoph Koch
École Polytechnique Fédérale de Lausanne, Switzerland

Frank Neven

UHasselt, Data Science Institute, ACSL, Diepenbeek, Belgium

Abstract

The popular isolation level Multiversion Read Committed (RC) trades some of the strong guarantees of serializability for increased transaction throughput. Sometimes, transaction workloads can be safely executed under RC obtaining serializability at the lower cost of RC. Such workloads are said to be robust against RC. Previous work has yielded a tractable procedure for deciding robustness against RC for workloads generated by transaction programs modeled as transaction templates. An important insight of that work is that, by more accurately modeling transaction programs, we are able to recognize larger sets of workloads as robust. In this work, we increase the modeling power of transaction templates by extending them with functional constraints, which are useful for capturing data dependencies like foreign keys. We show that the incorporation of functional constraints can identify more workloads as robust that otherwise would not be. Even though we establish that the robustness problem becomes undecidable in its most general form, we show that various restrictions on functional constraints lead to decidable and even tractable fragments that can be used to model and test for robustness against RC for realistic scenarios.

2012 ACM Subject Classification Information systems → Database transaction processing

Keywords and phrases concurrency control, robustness, complexity

Digital Object Identifier 10.4230/LIPIcs.ICDT.2022.16

Related Version *Full Version*: `https://arxiv.org/abs/2201.05021`

Funding This work is funded by FWO-grant G019921N.

1 Introduction

Many database systems implement several isolation levels, allowing users to trade isolation guarantees for improved performance. The highest, serializability, projects the appearance of a complete absence of concurrency, and thus perfect isolation. Executing transactions concurrently under weaker isolation levels can introduce certain anomalies. Sometimes, a transactional workload can be executed at an isolation level lower than serializability without introducing any anomalies. This is a desirable scenario: a lower isolation level, usually implementable with a cheaper concurrency control algorithm, yields the stronger isolation guarantees of serializability for free. This formal property is called robustness [12, 7]: a set of transactions $\mathcal{T}$ is called *robust against a given isolation level* if every possible interleaving of the transactions in $\mathcal{T}$ that is allowed under the specified isolation level is serializable.

Robustness received quite a bit of attention in the literature. Most existing work focuses on Snapshot Isolation (SI) [2, 4, 12, 13] or higher isolation levels [5, 7, 8, 10]. It is particularly interesting to consider robustness against lower level isolation levels like multi-version Read

© Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven;
licensed under Creative Commons License CC-BY 4.0
25th International Conference on Database Theory (ICDT 2022).
Editors: Dan Olteanu and Nils Vortmeier; Article No. 16; pp. 16:1–16:17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Committed (referred to as RC from now on). Indeed, RC is widely available, often the default in database systems (see, e.g., [4]), and is generally expected to have better throughput than stronger isolation levels.

In previous work [17], we provided a tractable decision procedure for robustness against RC for workloads generated by transaction programs modeled as transaction templates. The approach is centered on a novel characterization of robustness against RC in the spirit of [12, 14] that improves over the sufficient condition presented in [3], and on a formalization of transaction programs, called *transaction templates*, facilitating fine-grained reasoning for robustness against RC. Conceptually, transaction templates as introduced in [17] are functions with parameters, and can, for instance, be derived from stored procedures inside a database system (c.f. Figure 1 for an example). The abstraction generalizes transactions as usually studied in concurrency control research – sequences of read and write operations – by making the objects worked on variable, determined by input parameters. Such parameters are *typed* to add additional power to the analysis. They support *atomic updates* (that is, a read followed by a write of the same database object, to make a relative change to its value). Furthermore, database objects read and written are considered at the granularity of fields, rather than just entire tuples, decoupling conflicts further and allowing to recognize additional cases that would not be recognizable as robust on the tuple level.

An important insight obtained from [17] is that more accurate modeling of the workload allows to recognize larger sets of transaction programs as robust. Processing workloads under RC increases the throughput of the transactional database system compared to when executing the workload under SI or serializable SI, so larger robust sets mean better performance of the database system. In this work, we increase the modeling power of transaction templates by extending them with *functional constraints*, which are useful for capturing data dependencies like foreign keys (inclusion dependencies). This appears to be a sweet spot for strengthening modelling power – as we show in this paper, it allows us to remain with abstractions that have been well established within database theory, without having to move to general program analysis, and it pushes the robustness frontier on popular transaction processing benchmarks. Generally speaking, workloads can profit more from richer modelling the larger and more complex they get, so the fact that adding functional constraints yields larger robust sets already on these simple benchmarks suggests that these techniques are practically useful. Our contributions can be summarized as follows:

- We argue in Section 2 through the SmallBank and TPC-C benchmarks that the incorporation of functional constraints can identify more workloads as robust that otherwise would not be, and that they reduce the extent to which changes need to be made to workloads to make them robust against RC.
- In Section 4, we establish that robustness in its most general form becomes undecidable. The proof is a reduction from PCP and relies on cyclic dependencies between functions allowing to connect data values through an unbounded application of functions.
- We consider a fragment in Section 5 that only allows a very limited form of cyclic dependencies between functions and assumes additional constraints on templates that, together, imply that functions behave as bijections. Robustness against RC can be decided in NLOGSPACE and this fragment is general enough to model the SmallBank benchmark.
- In Section 6, we obtain an EXPSPACE decision procedure when the schema graph is acyclic (so, no cyclic dependencies between functions). Even for small input sizes, such a result is not practical. We provide various restrictions that lower the complexity to PSPACE and EXPTIME, and which allow to model the TPC-C benchmark as discussed. Notice that, for robustness testing, an exponential time decision procedure is considered to be practical as the size of the input is small and robustness is a static property that can be tested offline.

GoPremium:

$\texttt{U}[\texttt{X} : \text{Account}\{\text{N, C}\}\{\text{I}\}]$
$\texttt{R}[\texttt{Y} : \text{Savings}\{\text{C, I}\}]$
$\texttt{U}[\texttt{Y} : \text{Savings}\{\text{C}\}\{\text{I}\}]$
$\texttt{Y} = f_{A\to S}(\texttt{X}),\ \texttt{X} = f_{S\to A}(\texttt{Y})$

Figure 1 Transaction template.

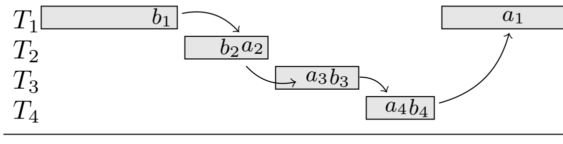

Figure 2 Multiversion split schedule.

These contributions should be contrasted with our earlier work [17], where we focused on a characterization for robustness against RC for basic transaction templates without functional constraints and performed an experimental study to show how the robustness property can improve transaction throughput.

Due to space constraints, proofs as well as a more complete description of the SmallBank and TPC-C benchmarks are moved to the online available full version of this paper [18].

2 Application

We present a small extension of the SmallBank benchmark [2] to exemplify the modeling power of transaction templates and discuss how the addition of functional constraints can detect larger sets of transaction templates to be robust. Finally, we discuss in the context of the TPC-C benchmark how the incorporation of functional constraints requires less changes to templates in making them robust. A full description of these benchmarks can be found in the online available full version [18].

The SmallBank schema consists of three tables: Account(Name, CustomerID, IsPremium), Savings(CustomerID, Balance, InterestRate), and Checking(CustomerID, Balance). Underlined attributes are primary keys. The Account table associates customer names with IDs and keeps track of the premium status (Boolean); CustomerID is a `UNIQUE` attribute. The other tables contain the balance (numeric value) of the savings and checking accounts of customers identified by their ID. Account (CustomerID) is a foreign key referencing both the columns Savings (CustomerID) and Checking (CustomerID). The interest rate on a savings account is based on a number of parameters, including the account status (premium or not). The application code can interact with the database through a fixed number of transaction programs: Balance, TransactSavings, Amalgamate, WriteCheck, DepositChecking, and GoPremium. We only discuss GoPremium(N), given in Figure 1, which converts the account of the customer with name N to a premium account and updates the interest rate of the corresponding savings account.

In short, a transaction template is a sequence of read (`R`), write (`W`) and update (`U`) statements over typed variables (`X`, `Y`, ...) with additional equality and disequality constraints. For instance, $\texttt{R}[\texttt{Y} : \text{Savings}\{\texttt{C}, \texttt{I}\}]$ indicates that a read operation is performed to a tuple in relation Savings on the attributes CustomerID and InterestRate. We abbreviate the names of attributes by their first letter to save space. The set $\{C, I\}$ is the read set. Write operations have an associated write set while update operations contain a read set followed by a write set: e.g., $\texttt{U}[\texttt{X} : \text{Account}\{\texttt{N}, \texttt{C}\}\{\texttt{I}\}]$ first reads the Name and CustomerID of tuple `X` and then writes to the attribute InterestRate. To capture the dependencies between tuples induced by the foreign keys, we use two unary functions: $f_{A\to S}$ maps a tuple of type Account to a tuple of type Savings, while $f_{A\to C}$ maps a tuple of type Account to a tuple of type Checking. As Account(CustomerID) is `UNIQUE`, every savings and checking accounts is associated to

a unique Account tuple. This is modelled through the functions $f_{C\to A}$ and $f_{S\to A}$ with an analogous interpretation. Notice that the equality constraints for GoPremium imply that these functions are bijections and each others inverses.

A transaction T over a database $\mathbf{D}$ is an *instantiation* of a transaction template τ if there is a variable mapping μ from the variables in τ to tuples in $\mathbf{D}$ that satisfies all the constraints in τ such that with $\mu(\tau) = T$. For instance, consider a database $\mathbf{D}$ with tuples $\mathtt{a}_1, \mathtt{a}_2, \ldots$ of type Account, $\mathtt{s}_1, \mathtt{s}_2, \ldots$ of type Savings, and $\mathtt{c}_1, \mathtt{c}_2, \ldots$ of type Checking with $f^{\mathbf{D}}_{A\to S}(\mathtt{a}_i) = \mathtt{s}_i$, $f^{\mathbf{D}}_{A\to C}(\mathtt{a}_i) = \mathtt{c}_i$, $f^{\mathbf{D}}_{S\to A}(\mathtt{s}_i) = \mathtt{a}_i$, $f^{\mathbf{D}}_{C\to A}(\mathtt{c}_i) = \mathtt{a}_i$ for each i. Then, for $\mu_1 = \{\mathtt{X} \to \mathtt{a}_1, \mathtt{Y} \to \mathtt{s}_1\}$, $\mu_1(\text{GoPremium}) = \mathtt{U}[\mathtt{a}_1]\mathtt{R}[\mathtt{s}_1]\mathtt{U}[\mathtt{s}_1]$ is an instantiation of GoPremium whereas $\mu_2(\text{GoPremium})$ with $\mu_2 = \{\mathtt{X} \to \mathtt{a}_1, \mathtt{Y} \to \mathtt{s}_2\}$ is not as the functional constraint $\mathtt{Y} = f_{A\to S}(\mathtt{X})$ is not satisfied. Indeed, $\mu_2(\mathtt{Y}) = \mathtt{s}_2 \neq \mathtt{s}_1 = f^{\mathbf{D}}_{A\to S}(\mathtt{a}_1) = f^{\mathbf{D}}_{A\to S}(\mu_2(\mathtt{X}))$. We then say that a set of transactions is *consistent* with a set of templates if every transaction is an instantiation of a transaction template.

Functional constraints do not replace the more usual data consistency constraints like key constraints, functional dependencies or denial constraints, ... The latter are intended to verify data consistency, whereas the former are intended to verify whether a set of transactions instantiated from templates are indeed consistent with these templates. The abstraction of functional constraints provides a straightforward mechanism to capture dependencies between tuples implied by e.g. foreign key constraints. Consider for example variables X and Y in GoPremium. Rather than specifying that the value of the attribute CustomerID in the tuple assigned to X should agree with the value of the attribute CustomerID in the tuple assigned to Y and combining this information with the defined foreign key from Account to Savings to conclude that two instantiations of GoPremium that agree on the tuple assigned to X should also agree on the tuple assigned to Y, the functional constraint $\mathtt{Y} = f_{A\to S}(\mathtt{X})$ expresses this dependency more directly. An additional benefit of our abstraction is that this approach is not limited to dependencies implied by foreign keys. For the SmallBank benchmark, for example, we can infer from the fact that Account(CustomerID) is `UNIQUE` that each checking and savings account is associated to exactly one Account tuple, even though no foreign key from respectively Checking and Savings to Account is defined in the schema.

Our previous work [17], which did not consider functional constraints, has shown that {Am,DC,TS}, {Bal,DC}, and {Bal,TS} are maximal robust sets of transaction templates. This means that for any database, for any set of transactions $\mathcal{T}$ that is consistent with one of the three mentioned sets, any possible interleaving of the transactions in $\mathcal{T}$ that is allowed under RC is *always* serializable! Using the results from Section 5, it follows that when functional constraints are taken into account GoPremium can be added to each of these sets as well: {Am,DC,GP,TS}, {Bal,DC,GP}, {Bal,TS,GP} are maximal robust sets.

We argue that incorporating functional constraints is crucial. Indeed, without functional constraints it's easy to show that even the set {GoPremium} is not robust. Consider the schedule over two instantiations T_1 and T_2 of GoPremium, where we use the mappings μ_1 and μ_2 as defined above for respectively T_1 and T_2 (we show the read and write sets to facilitate the discussion):

$$
\begin{array}{llll}
T_1: & \mathtt{U}_1[\mathtt{a}_1\{\text{N,C}\}\{\text{I}\}]\,\mathtt{R}_1[\mathtt{s}_1\{\text{C,I}\}] & & \mathtt{U}_1[\mathtt{s}_1\{\text{C}\}\{\text{I}\}]\,\mathtt{C}_1 \\
T_2: & & \mathtt{U}_2[\mathtt{a}_2\{\text{N,C}\}\{\text{I}\}]\,\mathtt{R}_2[\mathtt{s}_1\{\text{C,I}\}]\,\mathtt{U}_2[\mathtt{s}_1\{\text{C}\}\{\text{I}\}]\,\mathtt{C}_2 &
\end{array}
$$

The above schedule is allowed under RC as there is no dirty write, but it is not conflict serializable. Indeed, there is a rw-conflict between $\mathtt{R}_1[\mathtt{s}_1\{\text{C,I}\}]$ and $\mathtt{U}_2[\mathtt{s}_1\{\text{C}\}\{\text{I}\}]$ as the former reads the attribute I that is written to by the latter, which implies that T_1 should occur before

Delivery:

$\mathtt{U}[\mathtt{S}: \text{Order}\{\text{W, D, O}\}\{\text{Sta}\}]$
$\mathtt{U}[\mathtt{V}_1: \text{OrderLine}\{\text{W, D, O, OL, Del}\}\{\text{Del}\}]$
$\mathtt{U}[\mathtt{V}_2: \text{OrderLine}\{\text{W, D, O, OL, Del}\}\{\text{Del}\}]$
$\mathtt{U}[\mathtt{Z}: \text{Customer}\{\text{W, D, C, Bal}\}\{\text{Bal}\}]$
$\mathtt{Z} = f_{O\to C}(S),\ S = f_{L\to O}(\mathtt{V}_1),\ S = f_{L\to O}(\mathtt{V}_2)$

OrderStatus:

$\mathtt{R}[\mathtt{Z}: \text{Customer}\{\text{W, D, C, Inf, Bal}\}]$
$\mathtt{R}[\mathtt{S}: \text{Order}\{\text{W, D, O, C, Sta}\}]$
$\mathtt{R}[\mathtt{V}_1: \text{OrderLine}\{\text{W, D, O, OL, I, Del, Qua}\}]$
$\mathtt{R}[\mathtt{V}_2: \text{OrderLine}\{\text{W, D, O, OL, I, Del, Qua}\}]$
$\mathtt{Z} = f_{O\to C}(S),\ S = f_{L\to O}(\mathtt{V}_1),\ S = f_{L\to O}(\mathtt{V}_2)$

Figure 3 Transaction templates Delivery and OrderStatus of the TPC-C benchmark.

T_2 in an equivalent serial schedule. But, there is a ww-conflict between $\mathtt{U}_2[\mathtt{s}_1\{C\}\{I\}]$ and $\mathtt{U}_1[\mathtt{s}_1\{C\}\{I\}]$ as both write to the common attribute I implying that T_2 should occur before T_1 in an equivalent serial schedule. Consequently, the schedule is not serializable. However, taking functional constraints into account, $\{T_1, T_2\}$ is not consistent with {GoPremium} as $\mu_2(\mathtt{Y}) = \mathtt{s}_1 \neq \mathtt{s}_2 = f_{A\to S}(a_2) = f_{A\to S}(\mu_2(\mathtt{X}))$ implying that the above schedule is *not* a counter example for robustness.

Incorporating functional constraints for TPC-C can not identify larger sets of templates to be robust. However, when a set of transaction templates $\mathcal{P}$ is not robust against RC, an equivalent set of templates $\mathcal{P}'$ can be constructed from $\mathcal{P}$ by *promoting* certain R-operations to U-operations [17]. By incorporating functional constraints it can be shown that fewer R-operations need to be promoted leading to an increase in throughput as R-operations do not take locks whereas U-operations do. Consider for example the subset $\mathcal{P} = \{\text{Delivery}, \text{OrderStatus}\}$ of the TPC-C benchmark, given in Figure 3, where functional constraints are added to express the fact that a tuple of type OrderLine implies the tuple of type Order, which in turn implies the tuple of type Customer. This set $\mathcal{P}$ is not robust against RC, but robustness can be achieved by promoting the R-operation over Customer in OrderStatus to a U-operation. However, without functional constraints, this single promoted operation no longer guarantees robustness, as witnessed by the following schedule:

$$T_1(\text{Orderstatus}): \mathtt{U}_1[\mathtt{c}]\,\mathtt{R}_1[\mathtt{a}] \qquad\qquad\qquad\qquad \mathtt{R}_1[\mathtt{b}_1]\,\mathtt{R}_1[\mathtt{b}_2]\,\mathtt{C}_1$$
$$T_2(\text{Delivery}): \qquad\qquad \mathtt{U}_2[\mathtt{a}]\,\mathtt{U}_2[\mathtt{b}_1]\,\mathtt{U}_2[\mathtt{b}_2]\,\mathtt{U}_2[\mathtt{c}']\,\mathtt{C}_2$$

Notice in particular how this schedule implicitly assumes in T_2 that Order a belongs to Customer c′ instead of Customer c to avoid a dirty write on c. Without functional constraints, $\mathcal{P}$ is only robust against RC if *all* R-operations in OrderStatus are promoted to U-operations.

3 Definitions

We recall the necessary definitions from [17] and extend them with functional constraints.

3.1 Databases

A *relational schema* is a pair (Rels, Funcs) where Rels is a set of relation names and Funcs is a set of function names. A finite set of attribute names $\text{Attr}(R)$ is associated to every relation $R \in \mathsf{Rels}$. Relations will be instantiated by abstract objects that serve as an abstraction of relational tuples. To this end, for every relation $R \in \mathsf{Rels}$, we fix an infinite set of tuples $\mathbf{Tuples}_R$. Furthermore, we assume that $\mathbf{Tuples}_R \cap \mathbf{Tuples}_S = \emptyset$ for all $R, S \in \mathsf{Rels}$ with $R \neq S$. We then denote by $\mathbf{Tuples}$ the set $\bigcup_{R\in\mathsf{Rels}} \mathbf{Tuples}_R$ of all possible tuples. Notice that, by definition, for every $\mathtt{t} \in \mathbf{Tuples}$ there is a unique relation $R \in \mathsf{Rels}$ such that $\mathtt{t} \in \mathbf{Tuples}_R$. In that case, we say that t is of *type* R and denote the latter by $\text{type}(\mathtt{t}) = R$. Each function name $f \in \mathsf{Funcs}$ has a domain $dom(f) \in \mathsf{Rels}$ and a range $range(f) \in \mathsf{Rels}$.

Functions are used to encode relationships between tuples like for instance those implied by foreign-keys constraints. For instance, in the SmallBank example $\mathsf{Funcs} = \{f_{A\to S}, f_{A\to C}\}$, $dom(f_{A\to S}) = dom(f_{A\to C}) = A$, $range(f_{A\to S}) = S$, and $range(f_{A\to C}) = C$. A *database* $\mathbf{D}$ over schema ($\mathsf{Rels}, \mathsf{Funcs}$) assigns to every relation name $R \in \mathsf{Rels}$ a finite set $R^{\mathbf{D}} \subset \mathbf{Tuples}_R$ and to every function name $f \in \mathsf{Funcs}$ a function $f^{\mathbf{D}}$ from $dom(f)^{\mathbf{D}}$ to $range(f)^{\mathbf{D}}$.

3.2 Transactions and Schedules

For a tuple $\mathtt{t} \in \mathbf{Tuples}$, we distinguish three operations $\mathtt{R}[\mathtt{t}]$, $\mathtt{W}[\mathtt{t}]$, and $\mathtt{U}[\mathtt{t}]$ on $\mathtt{t}$, denoting that tuple $\mathtt{t}$ is read, written, or updated, respectively. We say that the operation is on the tuple $\mathtt{t}$. The operation $\mathtt{U}[\mathtt{t}]$ is an atomic update and should be viewed as an atomic sequence of a read of $\mathtt{t}$ followed by a write to $\mathtt{t}$. We will use the following terminology: a *read operation* is an $\mathtt{R}[\mathtt{t}]$ or a $\mathtt{U}[\mathtt{t}]$, and a *write operation* is a $\mathtt{W}[\mathtt{t}]$ or a $\mathtt{U}[\mathtt{t}]$. Furthermore, an $\mathtt{R}$-operation is an $\mathtt{R}[\mathtt{t}]$, a $\mathtt{W}$-operation is a $\mathtt{W}[\mathtt{t}]$, and a $\mathtt{U}$-operation is a $\mathtt{U}[\mathtt{t}]$. We also assume a special *commit* operation denoted $\mathtt{C}$. To every operation o on a tuple of type R, we associate the set of attributes $\text{ReadSet}(o) \subseteq \text{Attr}(R)$ and $\text{WriteSet}(o) \subseteq \text{Attr}(R)$ containing, respectively, the set of attributes that o reads from and writes to. When o is a $\mathtt{R}$-operation then $\text{WriteSet}(o) = \emptyset$. Similarly, when o is a $\mathtt{W}$-operation then $\text{ReadSet}(o) = \emptyset$.

A *transaction* T is a sequence of read and write operations followed by a commit. We assume that a transactions starts when its first operation is executed, but no earlier. Formally, we model a transaction as a linear order $(T, \leq_T)$, where T is the set of (read, write and commit) operations occurring in the transaction and $\leq_T$ encodes the ordering of the operations. As usual, we use $<_T$ to denote the strict ordering.

When considering a set $\mathcal{T}$ of transactions, we assume that every transaction in the set has a unique id i and write T_i to make this id explicit. Similarly, to distinguish the operations of different transactions, we add this id as a subscript to the operation. That is, we write $\mathtt{W}_i[\mathtt{t}]$, $\mathtt{R}_i[\mathtt{t}]$, and $\mathtt{U}_i[\mathtt{t}]$ to denote a $\mathtt{W}[\mathtt{t}]$, $\mathtt{R}[\mathtt{t}]$, and $\mathtt{U}[\mathtt{t}]$ occurring in transaction T_i; similarly $\mathtt{C}_i$ denotes the commit operation in transaction T_i. This convention is consistent with the literature (see, *e.g.* [6, 12]). To avoid ambiguity of notation, we assume that a transaction performs at most one write, one read, and one update per tuple. The latter is a common assumption (see, *e.g.* [12]). All our results carry over to the more general setting in which multiple writes and reads per tuple are allowed.

A *(multiversion) schedule* s over a set $\mathcal{T}$ of transactions is a tuple $(O_s, \leq_s, \ll_s, v_s)$ where O_s is the set containing all operations of transactions in $\mathcal{T}$ as well as a special operation op_0 conceptually writing the initial versions of all existing tuples, $\leq_s$ encodes the ordering of these operations, $\ll_s$ is a *version order* providing for each tuple $\mathtt{t}$ a total order over all write operations on $\mathtt{t}$ occurring in s, and v_s is a *version function* mapping each read operation a in s to either op_0 or to a write[1] operation different from a in s. We require that $op_0 \leq_s a$ for every operation $a \in O_s$, $op_0 \ll_s a$ for every write operation $a \in O_s$, and that $a <_T b$ implies $a <_s b$ for every $T \in \mathcal{T}$ and every $a, b \in T$.[2] We furthermore require that for every read operation a, $v_s(a) <_s a$ and, if $v_s(a) \neq op_0$, then the operation $v_s(a)$ is on the same tuple as a. Intuitively, op_0 indicates the start of the schedule, the order of operations in s is consistent with the order of operations in every transaction $T \in \mathcal{T}$, and the version function maps each read operation a to the operation that wrote the version observed by a. If $v_s(a)$ is op_0, then a observes the initial version of this tuple. The version order $\ll_s$ represents the

[1] Recall that a write operation is either a $\mathtt{W}[\mathtt{x}]$ or a $\mathtt{U}[\mathtt{x}]$.
[2] Recall that $<_T$ denotes the order of operations in transaction T.

order in which different versions of a tuple are installed in the database. For a pair of write operations on the same tuple, this version order does not necessarily coincide with $\leq_s$. For example, under RC the version order is based on the commit order instead.

We say that a schedule s is a *single version schedule* if $\ll_s$ coincides with $\leq_s$ and every read operation always reads the last written version of the tuple. Formally, for each pair of write operations a and b on the same tuple, $a \ll_s b$ iff $a <_s b$, and for every read operation a there is no write operation c on the same tuple as a with $v_s(a) <_s c <_s a$. A single version schedule over a set of transactions $\mathcal{T}$ is *single version serial* if its transactions are not interleaved with operations from other transactions. That is, for every $a, b, c \in O_s$ with $a <_s b <_s c$ and $a, c \in T$ implies $b \in T$ for every $T \in \mathcal{T}$.

The absence of aborts in our definition of schedule is consistent with the common assumption [12, 7] that an underlying recovery mechanism will rollback aborted transactions. We only consider isolation levels that only read committed versions. Therefore there will never be cascading aborts.

3.3 Conflict Serializability

Let a_j and b_i be two operations on the same tuple from different transactions T_j and T_i in a set of transactions $\mathcal{T}$. We then say that a_j is *conflicting* with b_i if:

- *(ww-conflict)* $\text{WriteSet}(a_j) \cap \text{WriteSet}(b_i) \neq \emptyset$; or,
- *(wr-conflict)* $\text{WriteSet}(a_j) \cap \text{ReadSet}(b_i) \neq \emptyset$; or,
- *(rw-conflict)* $\text{ReadSet}(a_j) \cap \text{WriteSet}(b_i) \neq \emptyset$.

In this case, we also say that a_j and b_i are conflicting operations. Furthermore, commit operations and the special operation op_0 never conflict with any other operation. When a_j and b_i are conflicting operations in $\mathcal{T}$, we say that a_j *depends on* b_i in a schedule s over $\mathcal{T}$, denoted $b_i \rightarrow_s a_j$ if:[3]

- *(ww-dependency)* b_i is ww-conflicting with a_j and $b_i \ll_s a_j$; or,
- *(wr-dependency)* b_i is wr-conflicting with a_j and $b_i = v_s(a_j)$ or $b_i \ll_s v_s(a_j)$; or,
- *(rw-antidependency)* b_i is rw-conflicting with a_j and $v_s(b_i) \ll_s a_j$.

Intuitively, a ww-dependency from b_i to a_j implies that a_j writes a version of a tuple that is installed after the version written by b_i. A wr-dependency from b_i to a_j implies that b_i either writes the version observed by a_j, or it writes a version that is installed before the version observed by a_j. A rw-antidependency from b_i to a_j implies that b_i observes a version installed before the version written by a_j.

Two schedules s and s' are *conflict equivalent* if they are over the same set $\mathcal{T}$ of transactions and for every pair of conflicting operations a_j and b_i, $b_i \rightarrow_s a_j$ iff $b_i \rightarrow_{s'} a_j$.

▶ **Definition 1.** *A schedule s is* conflict serializable *if it is conflict equivalent to a single version serial schedule.*

A *conflict graph* $CG(s)$ for schedule s over a set of transactions $\mathcal{T}$ is the graph whose nodes are the transactions in $\mathcal{T}$ and where there is an edge from T_i to T_j if T_i has an operation b_i that conflicts with an operation a_j in T_j and $b_i \rightarrow_s a_j$.

▶ **Theorem 2** ([15]). *A schedule s is conflict serializable iff the conflict graph for s is acyclic.*

[3] Throughout the paper, we adopt the following convention: a b operation can be understood as a "before" while an a can be interpreted as an "after".

3.4 Multiversion Read Committed

Let s be a schedule for a set $\mathcal{T}$ of transactions. Then, *s exhibits a dirty write* iff there are two ww-conflicting operations a_j and b_i in s on the same tuple $\mathtt{t}$ with $a_j \in T_j$, $b_i \in T_i$ and $T_j \neq T_i$ such that $b_i <_s a_j <_s \mathtt{C}_i$. That is, transaction T_j writes to an attribute of a tuple that has been modified earlier by T_i, but T_i has not yet issued a commit.

For a schedule s, the version order $\ll_s$ corresponds to the commit order in s if for every pair of write operations $a_j \in T_j$ and $b_i \in T_i$, $b_i \ll_s a_j$ iff $\mathtt{C}_i <_s a_j$. We say that a schedule s is *read-last-committed (RLC)* if $\ll_s$ corresponds to the commit order and for every read operation a_j in s on some tuple $\mathtt{t}$ the following holds:

- $v_s(a_j) = op_0$ or $\mathtt{C}_i <_s a_j$ with $v_s(a_j) \in T_i$; and
- there is no write[4] operation $c_k \in T_k$ on $\mathtt{t}$ with $\mathtt{C}_k <_s a_j$ and $v_s(a_j) \ll_s c_k$.

So, a_j observes the most recent version of $\mathtt{t}$ (according to the order of commits) that is committed before a_j. Note in particular that a schedule cannot exhibit dirty reads, defined in the traditional way [6], if it is read-last-committed.

▶ **Definition 3.** *A schedule is* allowed under isolation level *read committed (RC) if it is read-last-committed and does not exhibit dirty writes.*

3.5 Transaction Templates

Transaction templates are transactions where operations are defined over typed variables together with functional constraints on these variables. Types of variables are relation names in Rels and indicate that variables can only be instantiated by tuples from the respective type. We fix an infinite set of variables **Var** that is disjoint from **Tuples**. Every variable $\mathtt{X} \in \mathbf{Var}$ has an associated relation name in Rels as type that we denote by $\mathrm{type}(\mathtt{X})$. For an operation o_i in a template, $var(o_i)$ denotes the variable in o_i. An *equality constraint* is an expression of the form $\mathtt{X} = f(\mathtt{Y})$ where $\mathtt{X}, \mathtt{Y} \in \mathbf{Var}$, $dom(f) = \mathrm{type}(\mathtt{Y})$ and $range(f) = \mathrm{type}(\mathtt{X})$. A *disequality constraint* is an expression of the form $\mathtt{X} \neq \mathtt{Y}$ where $\mathrm{type}(\mathtt{X}) = \mathrm{type}(\mathtt{Y})$.

▶ **Definition 4.** *A* transaction template *is a transaction τ over* **Var** *together with a set $\Gamma(\tau)$ of equality and disequality constraints. In addition, for every operation o in τ over a variable $\mathtt{X}$, $ReadSet(o) \subseteq Attr(type(\mathtt{X}))$ and $WriteSet(o) \subseteq Attr(type(\mathtt{X}))$.*

Recall that we denote variables by capital letters $\mathtt{X}, \mathtt{Y}, \mathtt{Z}$ and tuples by small letters $\mathtt{t}, \mathtt{v}$. A variable assignment μ is a mapping from **Var** to **Tuples** such that $\mu(\mathtt{X}) \in \mathbf{Tuples}_{\mathrm{type}(\mathtt{X})}$. Furthermore, μ *satisfies* a constraint $\mathtt{X} = f(\mathtt{Y})$ (resp., $\mathtt{X} \neq \mathtt{Y}$) over a database $\mathbf{D}$ when $\mu(\mathtt{X}) = f^{\mathbf{D}}(\mu(\mathtt{Y}))$ (resp., $\mu(\mathtt{X}) \neq \mu(\mathtt{Y})$). A variable assignment μ for a transaction template τ is *admissible* for $\mathbf{D}$ if it satisfies all constraints in $\Gamma(\tau)$ over $\mathbf{D}$. By $\mu(\tau)$, we denote the transaction obtained by replacing each variable $\mathtt{X}$ in τ with $\mu(\mathtt{X})$.

A set of transactions $\mathcal{T}$ is *consistent* with a set of transaction templates $\mathcal{P}$ and database $\mathbf{D}$, if for every transaction T in $\mathcal{T}$ there is a transaction template $\tau \in \mathcal{P}$ and a variable mapping μ_T that is admissible for $\mathbf{D}$ such that $\mu_T(\tau) = T$.

3.6 Robustness

We define the robustness property [7] (also called *acceptability* in [12, 13]), which guarantees serializability for all schedules of a given set of transactions for a given isolation level.

[4] Recall that a write operation is either a $\mathtt{W}$ or a $\mathtt{U}$-operation.

▶ **Definition 5** (Transaction Robustness)**.** *A set $\mathcal{T}$ of transactions is* robust *against RC if every schedule for $\mathcal{T}$ that is allowed under RC is conflict serializable.*

In the next definition, we represent conflicting operations from transactions in a set $\mathcal{T}$ as quadruples (T_i, b_i, a_j, T_j) with b_i and a_j conflicting operations, and T_i and T_j their respective transactions in $\mathcal{T}$. We call these quadruples *conflicting quadruples* for $\mathcal{T}$. Further, for an operation $b \in T$, we denote by $\mathsf{prefix}_b(T)$ the restriction of T to all operations that are before or equal to b according to $\leq_T$. Similarly, we denote by $\mathsf{postfix}_b(T)$ the restriction of T to all operations that are strictly after b according to $\leq_T$. Throughout the paper, we interchangeably consider transactions both as linear orders as well as sequences. Therefore, T is then equal to the sequence $\mathsf{prefix}_b(T)$ followed by $\mathsf{postfix}_b(T)$ which we denote by $\mathsf{prefix}_b(T) \cdot \mathsf{postfix}_b(T)$ for every $b \in T$.

▶ **Definition 6** (Multiversion split schedule)**.** *Let $\mathcal{T}$ be a set of transactions and $C = (T_1, b_1, a_2, T_2), (T_2, b_2, a_3, T_3), \ldots, (T_m, b_m, a_1, T_1)$ a sequence of conflicting quadruples for $\mathcal{T}$ such that each transaction in $\mathcal{T}$ occurs in at most two different quadruples. A* multiversion split schedule *for $\mathcal{T}$ based on C is a multiversion schedule that has the following form:*

$$\mathsf{prefix}_{b_1}(T_1) \cdot T_2 \cdot \ldots \cdot T_m \cdot \mathsf{postfix}_{b_1}(T_1) \cdot T_{m+1} \cdot \ldots \cdot T_n,$$

where

1. *there is no write operation in $\mathsf{prefix}_{b_1}(T_1)$ ww-conflicting with a write operation in any of the transactions $T_2, \ldots, T_m$;*
2. *$b_1 <_{T_1} a_1$ or b_m is rw-conflicting with a_1; and,*
3. *b_1 is rw-conflicting with a_2.*

Furthermore, $T_{m+1}, \ldots, T_n$ are the remaining transactions in $\mathcal{T}$ (those not mentioned in C) in an arbitrary order.

Figure 2 depicts a schematic multiversion split schedule. The name stems from the fact that the schedule is obtained by splitting one transaction in two (T_1 at operation b_1 in Figure 2) and placing all other transactions in C in between. The figure does not display the trailing transactions $T_{m+1}, T_{m+2}, \ldots$ and assumes $b_1 <_{T_1} a_1$.

The following theorem characterizes non-robustness in terms of the existence of a multiversion split schedule.

▶ **Theorem 7** ([17])**.** *For a set of transactions $\mathcal{T}$, the following are equivalent:*

1. *$\mathcal{T}$ is not robust against RC;*
2. *there is a multiversion split schedule s for $\mathcal{T}$ based on some C.*

Let $\mathcal{P}$ be a set of transaction templates and $\mathbf{D}$ be a database. Then, $\mathcal{P}$ is *robust against RC over* $\boldsymbol{D}$ if for every set of transactions $\mathcal{T}$ that is consistent with $\mathcal{P}$ and $\mathbf{D}$, it holds that $\mathcal{T}$ is robust against RC.

▶ **Definition 8** (Template Robustness)**.** *A set of transaction templates $\mathcal{P}$ is* robust *against RC if $\mathcal{P}$ is robust against RC for every database $\boldsymbol{D}$.*

We say that a transaction template (τ, Γ) is a *variable transaction template* when $\Gamma = \emptyset$ and an *equality transaction template* when all constraints in Γ are equalities. We denote these sets by **VarTemp** and **EqTemp**, respectively. For an isolation level $\mathcal{I}$ and a class of transaction templates $\mathcal{C}$, T-ROBUSTNESS($\mathcal{C}$,$\mathcal{I}$) is the problem to decide if a given set of transaction templates $\mathcal{P} \in \mathcal{C}$ is robust against $\mathcal{I}$. When $\mathcal{C}$ is the class of all transaction templates, we simply write T-ROBUSTNESS($\mathcal{I}$).

▶ **Theorem 9** ([17])**.** T-ROBUSTNESS*(**VarTemp**,RC) is decidable in* PTIME.

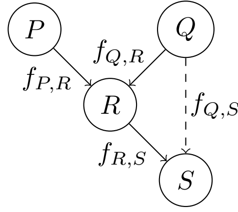

Figure 4 Acyclic schema graph for schema $(\{P,Q,R,S\},\{f_{P,R},f_{Q,R},f_{R,S},f_{Q,S}\})$. If we remove function name $f_{Q,S}$ (dashed edge), the resulting schema graph is a multi-tree.

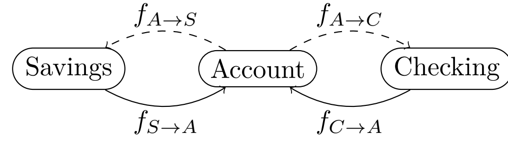

Figure 5 Schema graph for the SmallBank benchmark. The dashed edges correspond to the multi-tree schema graph for the schema restricted to $f_{A\to S}$ and $f_{A\to C}$.

4 Robustness for Templates

We start out with a negative result and show that the robustness problem in its most general form is undecidable (even when disequalities are not allowed). The proof is a reduction from *Post's Correspondence Problem (PCP)* [16] and relies on cyclic dependencies between functional constraints. The proof can be found in the full version of this paper [18] and is quite elaborate but the basic intuition is simple: the counterexample split schedule will build up the two strings that need to be generated by the PCP instance by repeated application of functional constraints.

▶ **Theorem 10.** T-ROBUSTNESS*(**EqTemp**,RC) is undecidable.*

It might be tempting to relate the above result to the undecidability of the implication problem for functional and inclusion dependencies [11]. Functional constraints indeed allow to define inclusion dependencies (as in the SmallBank example) but they always relate complete tuples and are not suited to define functional dependencies. Furthermore, the proof of Theorem 10 makes use of only unary relations, for which the implication problem for functional dependencies and inclusion dependencies is known to be decidable.

To obtain decidable fragments, we introduce restrictions on the structure of functional constraints. The *schema graph* $SG(\mathsf{Rels},\mathsf{Funcs})$ of a schema $(\mathsf{Rels},\mathsf{Funcs})$ is a directed multigraph having the relations in Rels as nodes, and in which there are as many edges from a node $R \in \mathsf{Rels}$ to node $S \in \mathsf{Rels}$ as there are functions $f \in \mathsf{Funcs}$ with $dom(f) = R$ and $range(f) = S$. We say that a schema $(\mathsf{Rels},\mathsf{Funcs})$ is *acyclic* if the multigraph $SG(\mathsf{Rels},\mathsf{Funcs})$ is acyclic and that it is a *multi-tree* if there is at most one directed path between any two nodes in $SG(\mathsf{Rels},\mathsf{Funcs})$.

▶ **Example 11.** Consider the schema $(\{P,Q,R,S\},\{f_{P,R},f_{Q,R},f_{R,S}\})$ with $dom(f_{i,j}) = i$ and $range(f_{i,j}) = j$ for each function $f_{i,j}$. The corresponding schema graph with solid lines is given in Figure 4. This schema is a multi-tree, as there is at most one path between any pair of nodes. Notice that the definition of a multi-tree is more general than a forest, as a node can still have multiple parents (e.g., node R in our example). Adding the function name $f_{Q,S}$ with $dom(f_{Q,S}) = Q$ and $range(f_{Q,S}) = S$ results in the schema graph given in Figure 4 that is still acyclic, but no longer a multi-tree as there are now two paths from Q to S. ◻

The schema graph constructed in the proof of Theorem 10 contains several cycles (we refer to [18] for a visualization of the constructed schema graph). We consider in Section 5 robustness for a fragment where a restricted form of cycles in the schema graph is allowed but where additional constraints on the templates are assumed. We consider robustness for acyclic schema graphs in Section 6.

5 Robustness for Templates admitting Multi-Tree Bijectivity

We say that a set of transaction templates $\mathcal{P}$ over a schema $(\mathsf{Rels}, \mathsf{Funcs})$ admits *multi-tree bijectivity* if a disjoint partitioning of Funcs in pairs $(f_1, g_1), (f_2, g_2), \ldots, (f_n, g_n)$ exists such that $dom(f_i) = range(g_i)$ and $dom(g_i) = range(f_i)$ for every pair of function names (f_i, g_i); every schema graph $SG(\mathsf{Rels}, \{h_1, h_2, \ldots, h_n\})$ over the schema restricted to function names $\{h_1, h_2, \ldots, h_n\}$ (with $h_i = f_i$ or $h_i = g_i$) is a multi-tree; and, for every pair of function names (f_i, g_i) and for every pair of variables $\mathtt{X}, \mathtt{Y}$ occurring in a template $\tau_j \in \mathcal{P}$, we have $f_i(\mathtt{X}) = \mathtt{Y} \in \Gamma_j$ iff $g_i(\mathtt{Y}) = \mathtt{X} \in \Gamma_j$. Intuitively, we can think of f_i as a bijective function, with g_i its inverse. We denote the class of all sets of templates admitting multi-tree bijectivity by **MTBTemp**. The SmallBank benchmark discussed in Section 2 is in **MTBTemp**, witnessed by the partitioning $\{(f_{A\to C}, f_{C\to A}), (f_{A\to S}, f_{S\to A})\}$. For example, the schema graph restricted to $f_{A\to C}$ and $f_{A\to S}$ is a tree and therefore also a multi-tree, as illustrated in Figure 5.

The next theorem allows disequalities whereas Theorem 10 does not require them.

▶ **Theorem 12.** T-ROBUSTNESS*(**MTBTemp**,RC) is decidable in* NLOGSPACE*.*

The approach followed in the proof of Theorem 12 is to repeatedly pick a transaction template while maintaining an overall consistent variable mapping in search for a counter-example multiversion split schedule that by Theorem 7 suffices to show that robustness does not hold. The main challenge is to show that a variable mapping consistent with all functional constraints can be maintained in logarithmic space and that all requirements for a multiversion split schedule can be verified in NLOGSPACE.

Central to our approach is a generalization of conflicting operations. Let $\mathcal{P}$ be a set of transaction templates. For τ_i and τ_j in $\mathcal{P}$, we say that an operation $o_i \in \tau_i$ is *potentially conflicting* with an operation $o_j \in \tau_j$ if o_i and o_j are operations over a variable of the same type, and at least one of the following holds:

- $\mathrm{WriteSet}(o_i) \cap \mathrm{WriteSet}(o_j) \neq \emptyset$ (potentially ww-conflicting);
- $\mathrm{WriteSet}(o_i) \cap \mathrm{ReadSet}(o_j) \neq \emptyset$ (potentially wr-conflicting); or
- $\mathrm{ReadSet}(o_i) \cap \mathrm{WriteSet}(o_j) \neq \emptyset$ (potentially rw-conflicting).

Intuitively, potentially conflicting operations lead to conflicting operations when the variables of these operations are mapped to the same tuple by a variable assignment. In analogy to conflicting quadruples over a set of transactions as in Definition 6, we consider *potentially conflicting quadruples* $(\tau_i, o_i, p_j, \tau_j)$ over $\mathcal{P}$ with $\tau_i, \tau_j \in \mathcal{P}$, and $o_i \in \tau_i$ an operation that is potentially conflicting with an operation $p_j \in \tau_j$. For a sequence of potentially conflicting quadruples $D = (\tau_1, o_1, p_2, \tau_2), \ldots, (\tau_m, o_m, p_1, \tau_1)$ over $\mathcal{P}$, we write $\mathrm{Trans}(D)$ to denote the set $\{\tau_1, \ldots, \tau_m\}$ of transaction templates mentioned in D. For ease of exposition, we assume a variable renaming such that any pair of templates in $\mathrm{Trans}(D)$ uses a disjoint set of variables.[5] The sequence D induces a sequence of conflicting quadruples $C = (T_1, b_1, a_2, T_2), \ldots, (T_m, b_m, a_1, T_1)$ by applying a variable assignment μ_i to each τ_i in $\mathrm{Trans}(D)$. We call such a set of variable assignments simply a *variable mapping* for D, denoted $\bar{\mu}$, and write $\bar{\mu}(D) = C$. For a variable $\mathtt{X}$ occurring in a template τ_i, we write $\bar{\mu}(\mathtt{X})$ as a shorthand notation for $\mu_i(\mathtt{X})$, with μ_i the variable assignment over τ_i in $\bar{\mu}$. This is well-defined as all templates in $\mathrm{Trans}(D)$ are variable-disjoint. Furthermore, $\bar{\mu}(var(o_i)) = \bar{\mu}(var(p_j))$ for each potentially

[5] To be formally correct, the latter would require to add every such variable-renamed template to $\mathcal{P}$ creating a larger set $\mathcal{P}'$. This does not influence the complexity of Theorem 12 as $\mathrm{Trans}(D)$ nor $\mathcal{P}'$ are used in the algorithm. Their only purpose is to reason about properties of $\bar{\mu}$.

conflicting quadruple $(\tau_i, o_i, p_j, \tau_j)$ in D as otherwise the induced quadruple (T_i, b_i, a_j, T_j) is not a valid conflicting quadruple in C. We say that a variable mapping $\bar{\mu}$ is admissible for a database $\mathbf{D}$ if every variable assignment μ_i in $\bar{\mu}$ is admissible for $\mathbf{D}$.

A basic insight is that if there is a multiversion split schedule s for some C over a set of transactions $\mathcal{T}$ consistent with $\mathcal{P}$ and a database $\mathbf{D}$, then there is a sequence of potentially conflicting quadruples D such that $\bar{\mu}(D) = C$ for some $\bar{\mu}$. We will verify the existence of such a C, satisfying the properties of Definition 6, by nondeterministically constructing D on-the-fly together with a mapping $\bar{\mu}$. We show in Lemma 14 that when $\mathcal{P} \in$ **MTBTemp**, $\bar{\mu}$ is a collection of disjoint type mappings (that map variables of the same type to the same tuple) such that variables that are "connected" in D (in a way that we will make precise next) are mapped using the same type mapping. Lemma 15 then shows that already a constant number of those type mappings suffice.

We introduce the necessary notions to capture when two variables are connected in D. We can think of equality constraints $\mathtt{Y} = f(\mathtt{X})$ in a template τ as constraints on the possible variable assignments μ for τ when a database $\mathbf{D}$ is given. Indeed, if we fix $\mu(\mathtt{X})$ to a tuple in $\mathbf{D}$, then $\mu(\mathtt{Y}) = f^{\mathbf{D}}(\mu(\mathtt{X}))$ is immediately implied. These constraints can cause a chain reaction of implications. If for example $\mathtt{Z} = g(\mathtt{Y})$ is a constraint in τ as well, then $\mu(\mathtt{X})$ immediately implies $\mu(\mathtt{Z}) = g^{\mathbf{D}}(f^{\mathbf{D}}(\mu(\mathtt{X})))$. We formalize this notion of implication next. We use sequences of function names $F = f_1 \cdots f_n$, denoting the empty sequence as ε and the concatenation of two sequences F and G by $F \cdot G$. For two variables $\mathtt{X}, \mathtt{Y}$ occurring in a template τ and a (possibly empty) sequence of function names F, we say that $\mathtt{X}$ *implies* $\mathtt{Y}$ *by* F *in* τ, denoted $\mathtt{X} \overset{F}{\rightsquigarrow}_\tau \mathtt{Y}$, if $\mathtt{X} = \mathtt{Y}$ and $F = \varepsilon$ or if there is a variable $\mathtt{Z}$ such that $\mathtt{Y} = f(\mathtt{Z})$ is a constraint in τ, $\mathtt{X} \overset{F'}{\rightsquigarrow}_\tau \mathtt{Z}$ and $F = F' \cdot f$. We next extend the notions of implication to sequences of potentially conflicting quadruples. Let $D = (\tau_1, o_1, p_2, \tau_2), \ldots, (\tau_m, o_m, p_1, \tau_1)$ be a sequence of potentially conflicting quadruples, and let $\mathtt{X}$ and $\mathtt{Y}$ be two variables occurring in templates τ_i and τ_j in $\text{Trans}(D)$, respectively. Then $\mathtt{X}$ *implies* $\mathtt{Y}$ *by a sequence of function names* F *in* D, denoted $\mathtt{X} \overset{F}{\rightsquigarrow}_D \mathtt{Y}$ if

- $i = j$ and $\mathtt{X} \overset{F}{\rightsquigarrow}_{\tau_i} \mathtt{Y}$ (implication within the same template);
- $F = \varepsilon$ and $(\tau_i, o_i, p_j, \tau_j)$ or $(\tau_j, o_j, p_i, \tau_i)$ is a potentially conflicting quadruple in D with o_i (respectively p_i) an operation over $\mathtt{X}$ and p_j (respectively o_j) an operation over $\mathtt{Y}$ (implication between templates, notice that $\mathtt{X} \overset{\varepsilon}{\rightsquigarrow}_D \mathtt{Y}$ iff $\mathtt{Y} \overset{\varepsilon}{\rightsquigarrow}_D \mathtt{X}$); or
- there exists a variable $\mathtt{Z}$ such that $\mathtt{X} \overset{F_1}{\rightsquigarrow}_D \mathtt{Z}$ and $\mathtt{Z} \overset{F_2}{\rightsquigarrow}_D \mathtt{Y}$ with $F = F_1 \cdot F_2$.

Two variables $\mathtt{X}$ and $\mathtt{Y}$ occurring in $\text{Trans}(D)$ are *connected in* D, denoted $\mathtt{X} \approx_D \mathtt{Y}$, if $\mathtt{X} \overset{F}{\rightsquigarrow}_D \mathtt{Y}$ or $\mathtt{Y} \overset{F}{\rightsquigarrow}_D \mathtt{X}$, or if there is a variable $\mathtt{Z}$ with $\mathtt{X} \approx_D \mathtt{Z}$ and either $\mathtt{Z} \overset{F}{\rightsquigarrow}_D \mathtt{Y}$ or $\mathtt{Y} \overset{F}{\rightsquigarrow}_D \mathtt{Z}$ for some sequence F. Furthermore, two variables $\mathtt{X}$ and $\mathtt{Y}$ occurring in a template τ are *connected in* τ, denoted $\mathtt{X} \approx_\tau \mathtt{Y}$, if $\mathtt{X} \overset{F}{\rightsquigarrow}_\tau \mathtt{Y}$ or $\mathtt{Y} \overset{F}{\rightsquigarrow}_\tau \mathtt{X}$, or if there is a variable $\mathtt{Z}$ with $\mathtt{X} \approx_\tau \mathtt{Z}$ and either $\mathtt{Z} \overset{F}{\rightsquigarrow}_\tau \mathtt{Y}$ or $\mathtt{Y} \overset{F}{\rightsquigarrow}_\tau \mathtt{Z}$ for some sequence F. These definitions of connectedness can be trivially extended to operations over variables: two operations in D (respectively τ) are connected in D (respectively τ) if they are over variables that are connected in D (respectively τ). When F is not important we drop it from the notation. For instance, we denote by $\mathtt{X} \rightsquigarrow_D \mathtt{Y}$ that there is an F with $\mathtt{X} \overset{F}{\rightsquigarrow}_D \mathtt{Y}$.

▶ **Lemma 13.** *Let D be a sequence of potentially conflicting quadruples over $\mathcal{P} \in$ **MTBTemp**. Then $\mathtt{X} \approx_D \mathtt{Y}$ implies $\mathtt{X} \rightsquigarrow_D \mathtt{Y}$ and $\mathtt{Y} \rightsquigarrow_D \mathtt{X}$. Furthermore, if type$(\mathtt{X})$ = type$(\mathtt{Y})$ then $\bar{\mu}(\mathtt{X}) = \bar{\mu}(\mathtt{Y})$ for every variable mapping $\bar{\mu}$ for D that is admissible for some database $\mathbf{D}$.*

It follows from Lemma 13 that, if we group connected variables, then the same tuple is assigned to all variables of the same type in this group. We encode this choice of tuples for variables through (total) functions $c : \mathsf{Rels} \to \mathbf{Tuples}$ that we call *type mappings* and which

map a relation onto a particular tuple of that relation's type. For instance, in SmallBank, a type mapping c is determined by an Account tuple $\mathtt{a}$, a Savings tuple $\mathtt{s}$, and a Checking tuple $\mathtt{c}$. The following lemma makes explicit how $\bar{\mu}$ can be decomposed into type mappings such that connected variables use the same type mapping and disequalities enforce the use of different type mappings.

▶ **Lemma 14.** *For a multiversion split schedule s based on a sequence of conflicting quadruples C over a set of transactions $\mathcal{T}$ consistent with a $\mathcal{P} \in \textbf{MTBTemp}$ and a database $\mathbf{D}$, let $\bar{\mu}$ be the variable mapping for a sequence of potentially conflicting quadruples D over $\mathcal{P}$ with $\bar{\mu}(D) = C$. Then, a set $\mathcal{S}$ of type mappings over disjoint ranges and a function $\varphi_{\mathcal{S}} : \mathbf{Var} \to \mathcal{S}$ exist with:*

- $\bar{\mu}(\mathtt{X}) = c(type(\mathtt{X}))$ *for every variable* $\mathtt{X}$*, with* $c = \varphi_{\mathcal{S}}(\mathtt{X})$*;*
- $\varphi_{\mathcal{S}}(\mathtt{X}) = \varphi_{\mathcal{S}}(\mathtt{Y})$ *whenever* $\mathtt{X} \approx\!\approx_D \mathtt{Y}$*; and,*
- $\varphi_{\mathcal{S}}(\mathtt{X}) \neq \varphi_{\mathcal{S}}(\mathtt{Y})$ *for every constraint* $\mathtt{X} \neq \mathtt{Y}$ *occurring in a template* $\tau \in Trans(D)$*.*

From $D = (\tau_1, o_1, p_2, \tau_2), \ldots, (\tau_m, o_m, p_1, \tau_1)$ and $\varphi_{\mathcal{S}}$ as in Lemma 14 we can derive a sequence of quintuples $E = (\tau_1, o_1, c_{o_1}, p_1, c_{p_1}), \ldots, (\tau_m, o_m, c_{o_m}, p_m, c_{p_m})$ such that $c_{o_i} = \varphi_{\mathcal{S}}(var(o_i))$ and $c_{p_i} = \varphi_{\mathcal{S}}(var(p_i))$ for $i \in [1, m]$. Intuitively, this sequence of quintuples can be used to reconstruct the original multiversion split schedule s. The next lemma shows that we can decide robustness against RC over a set of transaction templates admitting multi-tree bijectivity by searching for a specific sequence of quintuples over at most four type mappings.

▶ **Lemma 15.** *Let $\mathcal{P} \in \textbf{MTBTemp}$ and let $\mathcal{S} = \{c_1, c_2, c_3, c_4\}$ be a set consisting of four type mappings with disjoint ranges. Then, $\mathcal{P}$ is not robust against RC iff there is a sequence of quintuples $E = (\tau_1, o_1, c_{o_1}, p_1, c_{p_1}), \ldots, (\tau_m, o_m, c_{o_m}, p_m, c_{p_m})$ with $m \geq 2$ such that for each quintuple $(\tau_i, o_i, c_{o_i}, p_i, c_{p_i})$ in E:*

1. o_i *and* p_i *are operations in* τ_i*, and* $c_{o_i}, c_{p_i} \in \mathcal{S}$*;*
2. $\mathtt{X}_i \not\approx\!\approx_{\tau_i} \mathtt{Y}_i$ *for each constraint* $\mathtt{X}_i \neq \mathtt{Y}_i$ *in* τ_i*;*
3. $c_{o_i} = c_{p_i}$ *if* $o_i \approx\!\approx_{\tau_i} p_i$*;*
4. $c_{o_i} \neq c_{p_i}$ *if there is a constraint* $\mathtt{X}_i \neq \mathtt{Y}_i$ *in* τ_i *with* $\mathtt{X}_i \approx\!\approx_{\tau_i} var(o_i)$ *and* $\mathtt{Y}_i \approx\!\approx_{\tau_i} var(p_i)$*;*
5. *if* $i \neq 1$ *and* $c_{q_i} = c_{q_1}$ *for some* $q_i \in \{o_i, p_i\}$ *and* $q_1 \in \{o_1, p_1\}$*, then there is no operation* o'_i *in* τ_i *potentially ww-conflicting with an operation* o'_1 *in* $\mathsf{prefix}_{o_1}(\tau_1)$ *with* $var(o'_i) \approx\!\approx_{\tau_i} var(q_i)$ *and* $var(o'_1) \approx\!\approx_{\tau_1} var(q_1)$*.*

Furthermore, for each pair of adjacent quintuples $(\tau_i, o_i, c_{o_i}, p_i, c_{p_i})$ and $(\tau_j, o_j, c_{o_j}, p_j, c_{p_j})$ in E with $j = i + 1$, or $i = m$ and $j = 1$:

6. o_i *is potentially conflicting with* p_j *and* $c_{o_i} = c_{p_j}$*;*
7. *if* $i = 1$ *and* $j = 2$*, then* o_1 *is potentially rw-conflicting with* p_2*; and*
8. *if* $i = m$ *and* $j = 1$*, then* $o_1 <_{\tau_1} p_1$ *or* o_m *is potentially rw-conflicting with* p_1*.*

The items have the following meaning: (2) τ_i is satisfiable; (3) connected operations are assigned the same type mapping; (4) variables connected through an inequality are assigned a different type mapping; (5) $\varphi_{\mathcal{S}}$ only assigns the same type mapping to o_1 or p_1 in τ_1 and o_i or p_i in τ_i if it does not introduce a dirty write in the resulting multiversion split schedule (cf. Condition (1) in Definition 6); (6) each pair of variables in operations used for conflicts are assigned the same type mapping; (7, 8) the operations used for conflicts between τ_1, τ_2 and τ_m are restricted to satisfy respectively Condition (3) and (2) in Definition 6 in the resulting multiversion split schedule.

The characterization for T-ROBUSTNESS(**MTBTemp**,RC) in Lemma 15 implies an NLOGSPACE algorithm guessing the counterexample sequence E, thereby proving Theorem 12. Indeed, the algorithm guesses the sequence of quintuples E, verifying all conditions for each newly guessed quintuple while only requiring logarithmic space. Notice in particular that

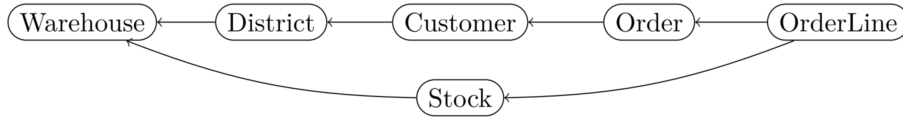

Figure 6 Acyclic schema graph for the TPC-C benchmark.

we only need to keep track of two other quintuples when verifying all conditions for the newly guessed quintuple, namely the first quintuple over τ_1 and the quintuple immediately preceding the newly guessed one. As usual, we can think of the encoding of templates and operations mentioned in each quintuple as pointers referring to the corresponding templates and operations on the input tape. Furthermore, we do not encode the four type mappings explicitly as such a representation of a mapping might require polynomial space. Since we are only interested in (dis)equality between type mappings, an encoding where these four type mappings are represented by four arbitrary strings of constant size suffices. More details can be found in [18].

6 Robustness for Templates over Acyclic Schemas

We denote by **AcycTemp** the class of all sets of transaction templates over acyclic schemas. As a concrete example, the schema graph for the TPC-C benchmark is given in Figure 6. Since this schema graph does not contain any cycles, the TPC-C benchmark is situated within **AcycTemp**. Notice in particular how this acyclic schema graph corresponds to the hierarchical structure of many-to-one relationships inherent to the schema for this benchmark. For example, every orderline belongs to exactly one order, and every order is related to exactly one customer, but the opposite is never true (i.e., a customer can be related to multiple orders, each of which can be related to multiple orderlines). In general, the results presented in this section can be applied to all workloads over schemas with such a hierarchical structure.

▶ **Theorem 16.** t-robustness*(**AcycTemp**,RC) is decidable in* expspace.

We provide some intuition for the proof. For a given acyclic schema graph SG, $R \overset{F}{\rightsquigarrow}_{SG} S$ denotes the directed path from node R to node S in SG with F the sequence of edge labels on the path. The next lemma relates implication between variables to paths in SG.

▶ **Lemma 17.** *Let D be a sequence of potentially conflicting quadruples over a set of transaction templates $\mathcal{P} \in$ **AcycTemp**. For every pair of variables $\mathtt{X}, \mathtt{Y}$ occurring in $Trans(D)$, if $\mathtt{X} \overset{F}{\rightsquigarrow}_D \mathtt{Y}$, then $type(\mathtt{X}) \overset{F}{\rightsquigarrow}_{SG} type(\mathtt{Y})$, with SG the corresponding schema graph.*

Notice that an assignment of a tuple to a variable $\mathtt{X}$ determines the tuples assigned to all variables $\mathtt{Y}$ with $\mathtt{X} \overset{F}{\rightsquigarrow}_D \mathtt{Y}$ for some sequence of function names F. From Lemma 17 it follows that each such implied tuple is witnessed by a path in the corresponding schema graph SG. Therefore, the maximal number of different tuples implied by $\mathtt{X}$ corresponds to the number of paths in SG starting in $\text{type}(\mathtt{X})$, which is finite when SG is acyclic. Because there can be multiple paths between nodes in the schema graph, it is no longer the case as in the previous section that variables of the same type connected in D must be assigned the same value. So, instead of using type mappings, we introduce *tuple-contexts* to represent the sets of all tuples implied by the assignment of a given variable. Formally, a *tuple-context for a type* $R \in \mathsf{Rels}$ is a function from paths with source R in $SG(\mathsf{Rels}, \mathsf{Funcs})$ to tuples in **Tuples** of the appropriate type. That is, for each tuple-context c for type R and for each path $R \overset{F}{\rightsquigarrow}_{SG} S$ in SG, $\text{type}(c(R \overset{F}{\rightsquigarrow}_{SG} S)) = S$.

Similar to Lemma 14, we show that we can represent a counterexample schedule based on D by assigning a tuple-context to each variable in Trans(D), taking special care when assigning contexts to variables connected in D to make sure that they are properly related to each other. For this, we introduce a (partial) function $\varphi_{\mathcal{A}} : \mathbf{Var} \to \mathcal{A}$ mapping (a subset of) variables in Trans(D) to tuple-contexts in $\mathcal{A}$ (for $\mathcal{A}$ a set of tuple-contexts)and refer to it as a *(partial) context assignment for D over $\mathcal{A}$*. In a sequence of lemma's, we show that $\varphi_{\mathcal{A}}$ can always be expanded into a total function and an approach based on enumeration of quintuples analogous to Lemma 15 suffices to decide robustness. A major difference with the previous section is that there is no longer a constant bound on the number of tuple-contexts that are needed and consistency between tuple-contexts in connected variables needs to be maintained. A full proof can be found in [18].

Next, we consider restrictions that lower the complexity. To this end, we say that two variables $\mathtt{X}$ and $\mathtt{Y}$ occurring in a transaction template τ are *equivalent in τ*, denoted $\mathtt{X} \equiv_\tau \mathtt{Y}$ if

- $\mathtt{X} = \mathtt{Y}$;
- there exists a pair of variables $\mathtt{Z}$ and $\mathtt{W}$ in τ and a sequence of function names F with $\mathtt{Z} \equiv_\tau \mathtt{W}$, $\mathtt{Z} \overset{F}{\rightsquigarrow}_\tau \mathtt{X}$ and $\mathtt{W} \overset{F}{\rightsquigarrow}_\tau \mathtt{Y}$; or
- there exists a variable $\mathtt{Z}$ with $\mathtt{X} \equiv_\tau \mathtt{Z}$ and $\mathtt{Y} \equiv_\tau \mathtt{Z}$.

Then, a transaction template τ is *restricted* if for every combination of variables $\mathtt{X}, \mathtt{Y}, \mathtt{W}, \mathtt{Z}$ in τ with $\mathtt{X} \rightsquigarrow_\tau \mathtt{W}$ and $\mathtt{Y} \rightsquigarrow_\tau \mathtt{Z}$, either $\mathtt{W} \equiv_\tau \mathtt{Z}$, $\mathtt{W} \rightsquigarrow_\tau \mathtt{Z}$ or $\mathtt{Z} \rightsquigarrow_\tau \mathtt{W}$. We denote by **AcycResTemp** the class of all sets of restricted transaction templates over acyclic schemas.

▶ **Theorem 18.** **1.** T-ROBUSTNESS*(**AcycResTemp**,RC) is decidable in* EXPTIME.
2. T-ROBUSTNESS*(**AcycTemp**,RC) is decidable in* PSPACE *when the number of paths between any two nodes in the schema graph is bounded by a constant k.*

Regarding (1), all templates in TPC-C with the exception of NewOrder are restricted. Regarding (2), when the schema graph is a multi-tree then $k = 1$ and for TPC-C $k = 2$ (recall that in general there can be an exponential number of paths), leading to a more practical algorithm for robustness in those cases.

7 Related Work

Transaction Programs

Previous work on static robustness testing [13, 3] for transaction programs is based on the following key insight: when a *schedule* is not serializable, then the dependency graph constructed from that schedule contains a cycle satisfying a condition specific to the isolation level at hand (*dangerous structure* for SNAPSHOT ISOLATION and the presence of a *counterflow edge* for RC). That insight is extended to a workload of *transaction programs* through the construction of a so-called static dependency graph where each program is represented by a node, and there is a conflict edge from one program to another if there can be a schedule that gives rise to that conflict. The absence of a cycle satisfying the condition specific to that isolation level then guarantees robustness while the presence of a cycle does not necessarily imply non-robustness.

Other work studies robustness within a framework for uniformly specifying different isolation levels in a declarative way [8, 7, 9]. A key assumption here is *atomic visibility* requiring that either all or none of the updates of each transaction are visible to other transactions. These approaches aim at higher isolation levels and cannot be used for RC, as RC does not admit *atomic visibility*.

Transaction Templates

The static robustness approach based on transaction templates [17] differs in two ways. First, it makes more underlying assumptions explicit within the formalism of transaction templates (whereas previous work departs from the static dependency graph that should be constructed in some way by the dba). Second, it allows for a decision procedure that is sound and complete for robustness testing against RC, allowing to detect larger subsets of transactions to be robust [17].

The formalisation of transactions and conflict serializability in [17] and this paper is based on [12], generalized to operations over attributes of tuples and extended with U-operations that combine R- and W-operations into one atomic operation. These definitions are closely related to the formalization presented by Adya et al. [1], but we assume a total rather than a partial order over the operations in a schedule. There are also a few restrictions to the model: there needs to be a fixed set of read-only attributes that cannot be updated and which are used to select tuples for update. The most typical example of this are primary key values passed to transaction templates as parameters. The inability to update primary keys is not an important restriction in many workloads, where keys, once assigned, never get changed, for regulatory or data integrity reasons.

In [17], a PTIME decision procedure is obtained for robustness against RC for templates without functional constraints and the present paper improves that result to NLOGSPACE. In addition, an experimental study was performed showing how an approach based on robustness and making transactions robust through promotion can improve transaction throughput.

Transactions

Fekete [12] is the first work that provides a necessary and sufficient condition for deciding robustness against SNAPSHOT ISOLATION for a workload of concrete transactions (not transaction programs). That work provides a characterization for acceptable allocations when every transaction runs under either SNAPSHOT ISOLATION or strict two-phase locking (S2PL). The allocation then is acceptable when every possible execution respecting the allocated isolation levels is serializable. As a side result, this work indirectly provides a necessary and sufficient condition for robustness against SNAPSHOT ISOLATION, since robustness against SNAPSHOT ISOLATION holds iff the allocation where each transaction is allocated to SNAPSHOT ISOLATION is acceptable. Ketsman et al. [14] provide full characterisations for robustness against READ COMMITTED and READ UNCOMMITTED under lock-based semantics. In addition, it is shown that the corresponding decision problems are complete for coNP and LOGSPACE, respectively, which should be contrasted with the polynomial time characterization obtained in [17] for robustness against *multiversion* read committed.

8 Conclusion

This paper falls within a more general research line investigating how transaction throughput can be improved through an approach based on robustness testing that can be readily applied without making any changes to the underlying database system. As argued in Section 2, incorporating functional constraints can detect larger sets of templates to be robust and requires less R-operations to be promoted to U-operations. In future work, we plan to look at lower bounds, restrictions that lower complexity, and consider other referential integrity constraints to further enlarge the modelling power of transaction templates.

References

1. Atul Adya, Barbara Liskov, and Patrick E. O'Neil. Generalized isolation level definitions. In *ICDE*, pages 67–78, 2000.
2. Mohammad Alomari, Michael Cahill, Alan Fekete, and Uwe Rohm. The cost of serializability on platforms that use snapshot isolation. In *ICDE*, pages 576–585, 2008.
3. Mohammad Alomari and Alan Fekete. Serializable use of read committed isolation level. In *AICCSA*, pages 1–8, 2015.
4. Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. Checking robustness against snapshot isolation. In *CAV*, pages 286–304, 2019.
5. Sidi Mohamed Beillahi, Ahmed Bouajjani, and Constantin Enea. Robustness against transactional causal consistency. In *CONCUR*, pages 1–18, 2019.
6. Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, pages 1–10, 1995.
7. Giovanni Bernardi and Alexey Gotsman. Robustness against consistency models with atomic visibility. In *CONCUR*, pages 7:1–7:15, 2016.
8. Andrea Cerone, Giovanni Bernardi, and Alexey Gotsman. A framework for transactional consistency models with atomic visibility. In *CONCUR*, pages 58–71, 2015.
9. Andrea Cerone and Alexey Gotsman. Analysing snapshot isolation. *J.ACM*, 65(2):1–41, 2018.
10. Andrea Cerone, Alexey Gotsman, and Hongseok Yang. Algebraic Laws for Weak Consistency. In *CONCUR*, pages 26:1–26:18, 2017.
11. Ashok K. Chandra and Moshe Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM J. Comput.*, 14(3):671–677, 1985.
12. Alan Fekete. Allocating isolation levels to transactions. In *PODS*, pages 206–215, 2005.
13. Alan Fekete, Dimitrios Liarokapis, Elizabeth J. O'Neil, Patrick E. O'Neil, and Dennis E. Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.
14. Bas Ketsman, Christoph Koch, Frank Neven, and Brecht Vandevoort. Deciding robustness for lower SQL isolation levels. In *PODS*, pages 315–330, 2020.
15. Christos H. Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.
16. Emil L. Post. A variant of a recursively unsolvable problem. *Bull. Amer. Math. Soc.*, pages 264–268, 1946.
17. Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. Robustness against read committed for transaction templates. *PVLDB*, 14(11):2141–2153, 2021.
18. Brecht Vandevoort, Bas Ketsman, Christoph Koch, and Frank Neven. Robustness against read committed for transaction templates with functional constraints (full version), 2022. URL: `https://arxiv.org/abs/2201.05021`.