Transactions are Back—but How Different They Are?
Relating STM and Databases Consistency Conditions (Preliminary Version)

Hagit Attiya
Technion
hagit@cs.technion.ac.il

Sandeep Hans
Technion
sandeep@cs.technion.ac.il

Abstract

We describe several database consistency conditions that restrict ongoing transactions (which might later be aborted), and relate them to known consistency conditions for transactional memory. In particular, we show that rigorousness is strictly stronger than opacity, but strictness is incomparable to opacity. The same relationships also hold for virtual world consistency. We also show that all non-eager STMs are strict.

1. Introduction

The transactional approach to programming concurrent applications is based on designating parts of the program as *transactions*. A transaction is a sequence of operations (typically, reads and writes) which are guaranteed to appear to execute atomically [9].

A *software implementation of transactional memory* (in short, *STM*) executes transactions in an *optimistic* manner, proceeding to read data and perform calculations based on it. The STM may need to abort transactions, when the consistency of the data could be compromised.

Transactions are ordinary pieces of code, and their execution is not expected to be "sandboxed", namely, they execute in an unsupervised manner. Therefore, an ongoing transaction must also see a consistent view of the data. Hence, there is a need to specify consistency conditions that also apply to the views of ongoing transactions, even if they may later abort.

This led to the introduction of several consistency conditions for transactional memory. All these conditions require committed transactions to be serializable [12], preserving the real-time order on non-overlapping transactions; they differ in the way they ensure that views of ongoing transactions that later abort, are consistent with a sequential history. The most widely-known STM consistency condition is *opacity* [6]; other consistency conditions include *virtual worlds consistency* (*VWC*) [11], and *transactional memory specification* (*TMS*) [4].

It has been argued that transactional memory imposes more stringent demands on transactions than database systems [5]. The claim is that, since the execution of database transactions is governed by a *concurrency control monitor*, database consistency conditions are less concerned with the views of aborted transactions.

However, the database literature includes several consistency conditions that enforce conditions on ongoing, and even aborted, transactions. The most notable ones are *rigorousness* [3], *strictness* [2], and *recoverability* [8]. These conditions are known to be comparable to one another, with rigorousness being the strongest condition and recoverability the weakest (Figure 1, based on [14]).

The transactional memory approach is inspired by database research [9], and it is interesting to understand DB conditions and relate them to STM conditions. This is important not only for historical reasons, but also in order to obtain better understanding of consistency of transactional memories, and hopefully, come up with cleaner and more accepted conditions.

This paper takes a step in this direction, and proves that rigorousness is strictly stronger than opacity. Surprisingly, the relationship between opacity and strictness is not straightforward, even though both are weaker than rigorousness. There are histories that are opaque but not strict and histories that are strict but not opaque. VWC follows the same relations as opacity.

We also show that STMs that are not *eager*, i.e., expose the values they write only when they are sure to be committed, satisfy strictness.

Organization of the paper: After presenting the basic model (Section 2), we describe the database consistency conditions in Section 3. STM conditions, in particular, opacity, are presented in Section 4, and the relationships between database and STM conditions is explored in Section 5. Non-eager STMs are discussed in Section 6. We conclude, with a summary of the results and related future research, in Section 7.

2. Basic Model

We outline the basic definitions, following [2, 14].

Let $\Gamma = \{T_1, \ldots, T_N\}$ be a set of *transactions*, where each transaction $T_i \in \Gamma$ has the form $T_i = (op_i, <_i)$, with op_i denoting the set of operations of T_i and $<_i$ denoting their total ordering, $1 \leq i \leq n$.

We consider only read and write operations, letting $r_i(x)$ and $w_i(x)$ denote, respectively, read and write operations on data item x by T_i. Two operations by different transactions *conflict* if they are on the same data item and at least one of them is a write.

Definition 1. *A* history *for a set of transactions* Γ *is a pair* $s = (op(s), <_s)$ *such that:*

1. $\bigcup_{i=1}^{n} <_i \subseteq <_s$, *the partial order of s extends all transaction orders;*
2. $op(s) \subseteq \bigcup_{i=1}^{n} op_i \cup \bigcup_{i=1}^{n} \{a_i, c_i\}$ *and* $\bigcup_{i=1}^{n} op_i \subseteq op(s)$, *i.e., s consists of the union of the operations from the given transactions plus a termination operation, which is either* c_i *(commit) or* a_i *(abort), for each* $T_i \in \Gamma$*;*
3. $(\forall i, 1 \leq i \leq n)$ $c_i \in op(s) \leftrightarrow a_i \notin op(s)$, *i.e., for each transaction, there is either a commit or an abort in s, but not both;*
4. $(\forall i, 1 \leq i \leq n)$ $(\forall o_i \in op_i)$ $o_i <_s a_i$ *or* $o_i <_s c_i$, *i.e., a commit or abort operation always appears as the last step of a transaction;*
5. *every pair of conflicting operations* $o_i, o_j \in op(s)$ *is ordered in s, i.e., either* $o_i <_s o_j$ *or* $o_j <_s o_i$.

A *schedule* is a prefix of a history.

The set of transactions occurring partially or completely in a schedule s is

$$trans(s) \coloneqq \{T_i \mid s \text{ contains steps from } T_i\} .$$

Definition 2. *The* committed projection *of a history s, denoted* $C(s)$, *is the history obtained by deleting all operations that do not belong to transactions committed in s.*

Definition 3. *Two histories s and s′ are* (conflict) equivalent *if*

1. *they are defined over the same set of transactions and have the same operations.*
2. *they order conflicting operations of non-aborted transactions in the same way; that is, for any conflicting operations* o_i *and* o_j *belonging to transactions* T_i *and* T_j *(respectively), where* $a_i, a_j \notin s$, *if* $o_i <_s o_j$ *then* $o_i <_{s'} o_j$.

Definition 4. *A history s is* (conflict) serializable *if its committed projection is (conflict) equivalent to a serial*[1] *history.*

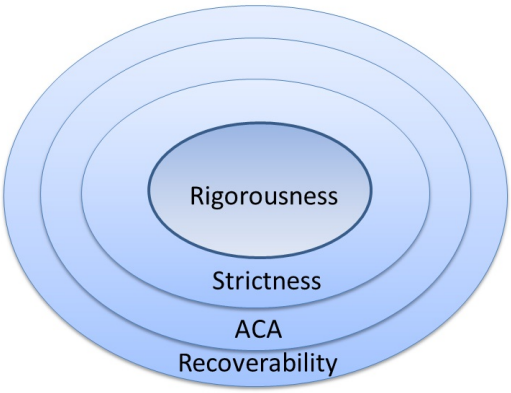

Figure 1. Database consistency conditions

3. Database Consistency Conditions

We will discuss four DB consistency conditions, their shortcomings and the need for stricter conditions; our description follows [2, 14]. In the rest of the paper, we always assume that the histories are also serializable. Figure 1 summarizes the relation between these conditions (cf. [14]).

3.1 Recoverability

Consider two transactions T_1 and T_2 as shown in Figure 2. T_1 writes a value 1 to data item x. T_2 reads x and commits. Then, T_1 aborts. T_2 has read a value written by an aborted transaction and has committed.

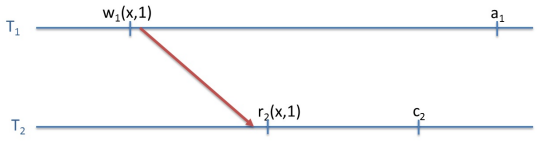

Figure 2. Dirty read problem

This problem is known as the *dirty read* problem, and the first consistency condition that we will describe, *recoverability*, takes care of it.

Recoverability states that all the transactions that have written the shared values read by transaction T, should commit before T.

[1] Roughly speaking, a history s is *serial* if all operations of one transaction precede (in $<_s$) all operations of another transaction.

A transaction T_i reads from transaction T_j if T_i reads a value written by T_j.

Definition 5 (Hadzilacos [8]). *A schedule s is* recoverable *if the following holds for all transactions $T_i, T_j \in trans(s)$, $i \neq j$: if T_i reads from T_j in s and $c_i \in op(s)$, then $c_j <_s c_i$.*

Consider the scenario in Figure 3. Transaction T_2 reads the value of x from T_1. By recoverability, T_2 should commit only if T_1 has committed.

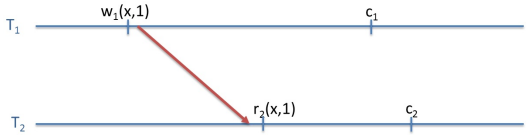

Figure 3. Recoverability

3.2 Avoiding Cascading Aborts

Consider the scenario in Figure 4. T_2 reads the value written by T_1 and T_1 aborts after that. When T_2 tries to commit, it will have to abort because of T_1.

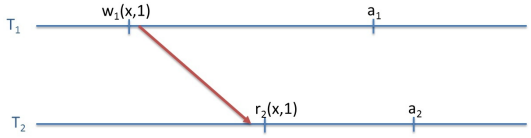

Figure 4. Cascading abort problem

This is known as the *cascading aborts* problem. If a transaction aborts, then all the subsequent transactions reading the values written by this transaction will have to abort. Recoverability states that if a transaction, say T_1, writes a value read by another transaction T_2, then T_2 can only commit after T_1 is committed or T_1 should have committed before T_2 is committed. In order to avoid cascading aborts, T_1 should have committed before T_2 reads the value, i.e., a transaction can read only from committed transactions, as stated in the next definition.

Definition 6 (Bernstein et al. [2]). *A schedule s* avoids cascading aborts *(ACA) if the following holds for all transactions $T_i, T_j \in trans(s)$, $i \neq j$: if T_i reads the value of x from T_j in s in the operation r_i, then $c_j <_s r_i(x)$.*

In Figure 5, T_2 reads the value written by T_1, which is committed.

3.3 Strictness

There is a problem with the ACA condition, which applies only for databases or for STMs with in-place updates. If a transaction aborts, all its effects are rolled back to the state before the transaction began. Consider

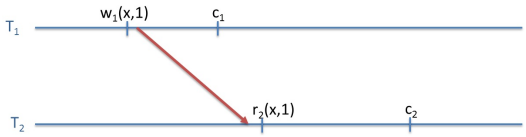

Figure 5. Avoiding cascading aborts

the scenario depicted in Figure 6. Let the initial value of data item x be 1. T_1 writes 2 to x, then T_2 writes 3 and commits. Then T_1 aborts and the system is rolled-back to the state where it was before T_1 started. The write by T_2, though committed, is lost.

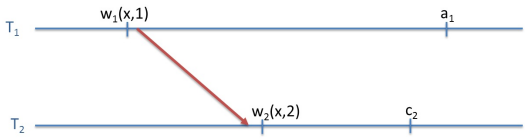

Figure 6. Problem with ACA

Recoverability and ACA are based on causality, where a transaction is concerned only with transactions writing values that it reads. In *strictness*, however, a transaction is also concerned with transactions that write to a data item it reads. A transaction writing to a data item should complete (commit or abort) before another transaction reads from the data item or overwrites it.

Definition 7 (Bernstein et al. [2]). *A schedule s is* strict *if the following holds for all transactions $T_i \in trans(s)$ and for all $o_i(x) \in op(T_i)$, $o \in \{r, w\}$: if $w_j(x) <_s o_i(x)$, $i \neq j$, then $a_j <_s o_i(x) \vee c_j <_s o_i(x)$.*

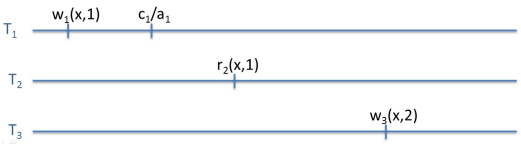

Figure 7. Strictness

3.4 Rigorousness

In strictness, if a transaction reads from a data item, it can read only if all the transactions that write to this data item have committed or aborted. *Rigorousness* states that in addition to strictness, a transaction can write to a data item only if all the transactions reading it have committed or aborted.

Definition 8 (Breitbart et al.[3]). *A schedule s is* rigorous *if it is strict and additionally satisfies the following condition: for all transactions $T_i, T_j \in trans(s)$ if $r_j(x) <_s w_i(x)$, $i \neq j$, then $a_j <_s w_i(x) \vee c_j <_s w_i(x)$.*

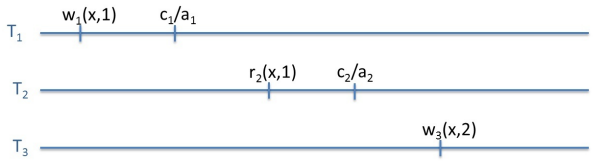

Figure 8. Rigorousness

4. STM Consistency Conditions

4.1 Opacity

Opacity, introduced in [6], is the strictest consistency condition known for STM. Informally, a history satisfies opacity if all the committed and aborted transactions (projected on the read operations) appear as if they execute in a sequential order and this order respects real-time occurrences of all transactions.

Definition 9 (Guerraoui and Kapalka [7]). *A sequential STM history S is* legal *if, for every shared variable x, the restriction of the history S to operations on x is in the sequential specification of x.*

Definition 10 (Guerraoui and Kapalka [7]). *A transaction T_i is* legal *in a complete sequential history S, if STM history $visible_S(T_i)$ is legal, where $visible_S(T_i)$ is the longest subsequence S' of S such that, for every transaction T_k in S', either (1) $k = i$, or (2) T_k is committed and $T_k <_S T_i$.*

Definition 11 (Guerraoui and Kapalka [7]). *A finite STM history H is* final-state opaque *if there exists a sequential STM history S equivalent to some completion*[2] *of H, such that*

1. *S preserves the real-time order of H, and*
2. *every transaction T_i in H is legal in S.*

Definition 12 (Guerraoui and Kapalka [7]). *A STM history H is* opaque *if every finite prefix of H is final-state opaque.*

There is an equivalent characterization of opacity, based on a graph representation of a history [7]. This characterization assumes that operations are atomic, and that writes are unique, i.e., the values written to each data item are distinct.

A directed graph $OPG(H)$ is defined for a STM history H as follows.

Every transaction T_i is a vertex in the graph. It is labeled *vis* if T_i is committed or if some transaction reads from T_i, and *loc*, otherwise.

There are four types of edges:

1. Real Time (rt).
   If a transaction T_i finishes before T_j starts, there is an edge from T_i to T_j; it is labeled as *rt*.
2. Read From (rf).
   If T_j reads a value written by T_i, then there is an edge from T_i to T_j; it is labeled *rf*.
3. Write Before (ww).
   If T_j overwrites a value written by T_i, and both T_i and T_j are committed or commit-pending[3], then there is an edge from T_i to T_j; it is labeled *ww*.
4. Read Before Write (rw).
   If T_j is labeled *vis*, T_j writes to data item x, and T_i reads from another transaction that writes to x before T_j (this is well-defined since writes are unique and atomic), then there is an edge from T_i to T_j; it is labeled *rw*.

The opacity graph for the history of Figure 9 is shown in Figure 10. T_0 is a "virtual" transaction, "writing" the initial values of all data items. There is an *rt* edge from T_1 to T_2 since T_1 finishes before T_2 starts. The edges labeled as *rf* show that T_3 is reading from T_0, i.e. the initial value of x and T_2 is reading from T_1. The *ww* edge from T_0 to T_1 shows that the transaction T_1 overwrites an initial value. The *rw* edge from T_3 to T_1 shows that T_3 is reading a value overwritten by T_1. Note that there is no *ww* edge from T_1 to T_3, since T_3 aborts.

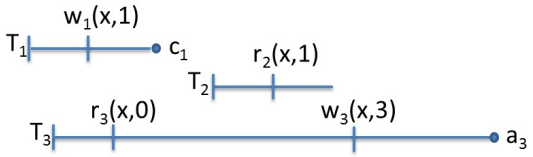

Figure 9. Example

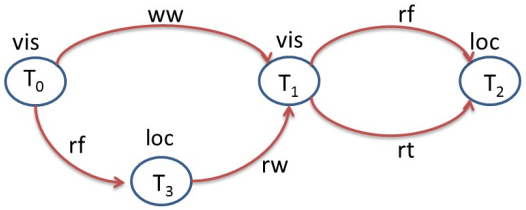

Figure 10. Opacity graph for the history of Figure 9

Theorem 1 (Graph characterization of opacity [7]). *A consistent STM history H is final-state opaque if and only if the graph $OPG(H)$ is acyclic.*

4.2 Virtual World Consistency

Virtual world consistency (VWC) [11] is a weaker condition than opacity. It requires that all the committed

[2] Roughly, inserting abort for all ongoing transactions.

[3] A transaction is *commit-pending* if it has completed all its operations, has invoked the termination operation and is waiting for its response.

transactions appear to execute in a sequential order, which also respects the real-time order of these transactions. For each aborted transaction, VWC only requires that the values it reads are consistent with respect only to its *causal past*, i.e., the committed transactions from which it reads and previously committed transactions by the same thread.

Figure 11 shows example of a history which is virtual world consistent but not opaque. Transaction T_1 reads a value of x written by T_2, and hence considers only T_2 and not T_3.

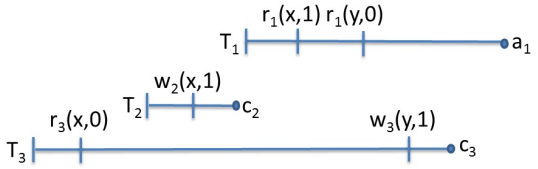

Figure 11. A VWC history that is not opaque

5. Relating Database and STM conditions

5.1 Rigorousness is Contained in Opacity

Lemma 2. *The graph OPG of a rigorous STM history is acyclic.*

Proof. Consider a rigorous *STM* history H. Suppose the graph $OPG(H)$ has a cycle of edges from transaction T_1 to transaction T_2, from transaction T_2 to transaction T_3, and so on, and there is also an edge from transaction T_n to transaction T_1, completing the cycle. Each edge is of one of the four types $\{rt, rf, ww, rw\}$.

We argue for each edge (T_i, T_j) that T_i must complete before T_j. By the definition of rigorousness, if a transaction T_j is either reading from T_i (rf), or overwriting some data item written by T_i (ww) or overwriting a value read by T_i (rw), then T_i must complete before the corresponding operation by T_j. This also means that T_i must complete before T_j, since last step of T_j would be a completing step. If the edge is labeled rt, T_i must complete even before T_j started.

Thus, T_1 must complete before T_2, T_2 must complete before T_3, and so on. Finally, the edge from T_n to T_1 implies that T_n must complete before T_1, which is clearly not possible. This contradiction shows that the opacity graph of a rigorous history is always acyclic. □

Theorem 3. *Rigorousness ⊊ Opacity.*

Proof. Consider a rigorous *STM* history H. By Lemma 2, $OPG(H)$ is acyclic. Hence, by Theorem 1, H is opaque.

Consider the history in Figure 12. It is opaque since it is equivalent to the sequential history T_1, T_2. This

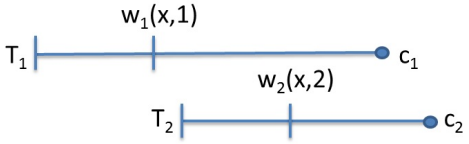

Figure 12. An opaque history that is not rigorous

history is not rigorous, since T_1 does not complete before T_2 writes 2 to x. □

Note that STMs with *invisible reads*, in which read operations do not write to any base object, can be opaque but cannot be rigorous.

5.2 Strictness is Incomparable to Opacity

As mentioned, the history in Figure 12 is opaque; however, is not strict since T_2 writes the value of x before T_1 completes.

On the other hand, the history in Figure 13 is not opaque since there is a cycle T_2T_3 in its opacity graph. There is an rw edge from T_2 to T_3 and an rf edge from T_3 to T_2. However, this history is strict since T_2 is reading from T_1 after T_1 is complete and T_2 is reading from T_3 after T_3 is completed.

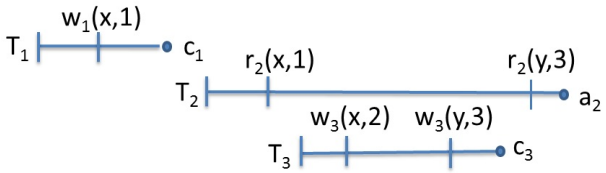

Figure 13. A strict history that is not opaque

5.3 ACA and Recoverability

We have seen that there is a strict history that is not opaque (Figure 13); this history is also ACA. We believe that any opaque history is ACA (and hence recoverable), since opacity does not allow inconsistent reads, making it impossible for a transaction to read from a uncommitted transaction, making the history ACA.

5.4 Virtual World Consistency

Inspecting our results easily show that the same relationships hold also for VWC. For example, the strict history of Figure 13, which is not opaque, does not satisfy virtual world consistency either.

6. Non Eager STMs

Many STMs write their shared data only when it is clear they are going to commit. This is in contrast to STMs that write their new values with each write

operation. We say that STMs of the first kind are *non eager*, and remark that they may write to some base objects before getting committed, e.g., acquiring a lock or incrementing a shared counter.

All the database conditions that we have discussed so far, assume commit is an atomic operation (like read and write). This assumption is not valid when commit has a duration and is not executed in exclusion. During a commit operation, there is a point at which the transaction is sure to commit successfully; we call this the *logical commit-point* and it may be well before the end of the commit operation. Non-eager STMs write only after their logical commit-point. All the above-stated relationships between STM and DB consistency conditions hold even if the definitions of database conditions are changed to refer to the logical commit-point instead the commit.

6.1 Non-eagerness Implies Strictness

In non-eager STMs, a read operation always reads a committed value, which enables to prove that they are always strict.

Theorem 4. *A non-eager STM is strict.*

Proof. Since a transaction T_w writes its values only at the commit point, any other transaction reads or overwrites the committed values only after the logical commit-point of T_w. □

Figure 14 shows the difference between strictness and non-eagerness. This history is strict since the read of shared variable x and the write of shared variable y, by transaction T_2, are done only after the transaction T_1 is completed (committed in this case). This history is, however, not non-eager since all the writes are done well before commit is invoked.

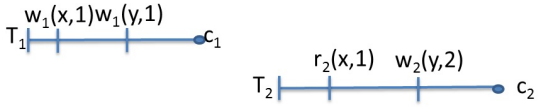

Figure 14. A strict history that is not non-eager

6.2 Non-eagerness is Incomparable to Rigorousness and Opacity

The history in Figure 15 is *not* non-eager, since the write of shared variable x ($w_1(x, 2)$) is done before the commit point. However, this history is rigorous (and hence opaque) since the write of x by T_2 is done only after T_1, which is reading a previous value of x, completes.

Consider the history in Figure 16 where writes of T_2 happen at the commit-point. This history is non-eager since all the writes are done only at the commit

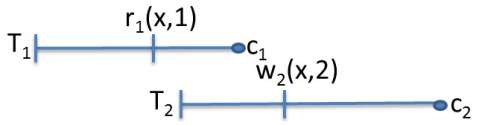

Figure 15. A rigorous history that is not non-eager

point only. This history is not opaque (and hence not rigorous) since there is a cycle in the opacity graph, between transaction T_1 and transaction T_2, because T_2 overwrites the value of shared variable x read by T_1 and T_1 reads the value of shared variable y written by T_2.

Figure 16. A non-eager history that is not opaque

7. Discussion

Figure 17 summarizes the relationships we have proved between STM and DB consistency conditions. It shows that rigorousness is the strongest condition of all STM and DB conditions, but the relationships between other conditions is not clear.

Rigorousness imposes a condition between reads and later writes to the same data item. This seems to require that reads write to the shared memory, a property that is considered undesirable in STM implementations. Proving that this is indeed a necessary condition, and understanding the ramifications is an interesting question.

This paper focused on opacity and VWC. *Transactional memory specification* (TMS) [4] is another consistency condition for STMs, defined using *I/O automata*. The original definition of opacity [6] is not prefix-closed, and hence there are histories that satisfy opacity, but not TMS (cf. [4]). These histories do not satisfy the updated definition of opacity [7], which we use. Under the new definition, opacity is contained in TMS and hence, rigorousness is also contained in TMS. It seems that the other incomparability results hold also for TMS, but verifying this precisely is left for future work.

Another approach to STM consistency [13] builds upon *snapshot isolation*. Snapshot isolation decouples the consistency of the reads and the writes. Informally, all read operations in a transaction return the value of the most recent value as of the time the transaction

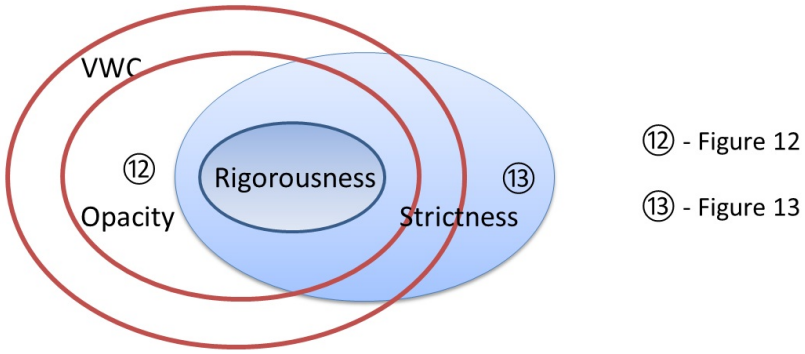

Figure 17. Relating STM and DB conditions; numbers indicate the figures presenting the relevant example

starts. In addition, the write sets of any pair of concurrent transactions must be disjoint.

Both DB and STM aim to provide consistency conditions that admit the maximal level of parallelism. Looking back at the definitions, however, it is clear that different perspectives are manifested in the way the consistency conditions are defined. DB conditions are defined at an operation level, in terms of basic operations (reads and writes). On the other hand, STM conditions are defined at a transaction level, in terms of the sequences of responses that can be observed by a thread (in a manner that follows serializability [12], or linearizability [10]).

Our results relate these different levels, proving, in one case, that a DB condition (rigorousness) implies an STM condition (opacity). It would be interesting to provide more systematic relations between these points of view.

There is another, even higher level to define consistency conditions, namely, from the point-of-view of programs that employ transactions (e.g., [1]). It is very intriguing (and important) to relate definitions at the programming language level to definitions based on allowed responses.

Acknowledgements: The authors would like to thank Eshcar Hillel and Panagiota Fatourou for helpful discussions, and the referees for their comments.

This research is supported by funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 238639, ITN project TRANSFORM.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM SIGPLAN Notices*, 43(1):63–74, 2008.

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

[3] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Trans. Softw. Eng.*, 17:954–960, September 1991.

[4] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Electron. Notes Theor. Comput. Sci.*, 259:245–261, December 2009. updated version available at http://labs.oracle.com/people/moir/pubs/Doherty-et-al-Formal-Aspects-submission.pdf.

[5] P. Felber, C. Fetzer, R. Guerraoui, and T. Harris. Transactions are back—but are they the same? *SIGACT News*, 39:48–58, March 2008.

[6] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175–184, 2008.

[7] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2011.

[8] V. Hadzilacos. A theory of reliability in database systems. *J. ACM*, 35:121–145, January 1988.

[9] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *ACM SIGARCH Computer Architecture News*, 21(2):289–300, 1993.

[10] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[11] D. Imbs, J. R. G. de Mendívil, and M. Raynal. Brief announcement: virtual world consistency: a new condition for stm systems. In *PODC*, pages 280–281, 2009.

[12] C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26:631–653, October 1979.

[13] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *In Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT06)*, 2006.

[14] G. Weikum and G. Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. ISBN 1-55860-508-8.