Relaxing Real-time Order in Opacity and Linearizability

Tadeusz Kobus, Maciej Kokociński, Paweł T. Wojciechowski

Institute of Computing Science,
Poznan University of Technology,
Piotrowo 2, 90-965, Poland
+(48) 61 665 3021

Abstract

In this paper we introduce two families of safety properties: ◊-opacity and ◊-linearizability. The new properties relax to various degree the real-time order requirement on transaction execution in opacity and, analogically, the real-time order requirement on operation execution in linearizability. This way we can formalize the guarantees provided by a wide class of strongly consistent replicated systems for which opacity and linearizability are too strong. As an example, we analyze Deferred Update Replication, a well known optimistic concurrency control scheme, and prove it satisfies update-real-time opacity, which allows read-only and aborted transactions to operate on stale (but still consistent) data. We also show the relationship between ◊-opacity and ◊-linearizability.

Keywords: correctness, opacity, linearizability, deferred update replication

The project was funded from National Science Centre funds granted by decision No. DEC-2012/07/B/ST6/01230.
1. Introduction

Replication is an established technique used for building dependable and highly available services. In a replicated system, a service is deployed on multiple machines whose actions are coordinated, so that a consistent state is maintained across all the service replicas (processes). This way the clients, who can issue requests to any of the replicas, are served despite of (partial) system failures.

It is often desirable that a replicated service provides a real-time guarantee on the execution of client requests. It means that if the execution of one request ends before another request starts its execution (according to, e.g., a wall-clock), the effects of the former are always visible to the latter. Providing this guarantee in a distributed environment is not easy and tends to be very costly. It is because respecting real-time order at all times requires inter-process synchronization on execution of every single request. Therefore, replicated services often relax the real-time requirement for at least some types of requests. E.g., read-only or aborted requests do not change the state of the system, so they do not need to be ordered wrt. concurrent update requests. This way the system performance can be greatly improved, especially when requests of those special types constitute the vast majority of all requests processed by the system.

To better understand the consequences of loosening real-time order for some types of requests, let us consider an example of Deferred Update Replication (DUR) [1], a popular replication scheme for databases and distributed transactional memory (DTM) systems.

In DUR, every request sent by a client to any of the replicas, is executed by the replica as an atomic transaction. The resulting updates are broadcast to all processes using a Total Order Broadcast protocol [2], so that they can update their state accordingly. However, upon receipt of a message with a transaction’s updates, the processes do not update their state right away. In order to ensure that consistency is preserved across the system, all processes (independently) execute a certification procedure that checks if the transaction read any stale data. If so, the transaction has to be rolled back and restarted.
Since all updates are delivered in the same order (guaranteed by Total Order Broadcast), all processes change their state in the same way. However, read-only transactions do not modify the system’s state in any way, so they do not require a distributed certification. Instead, only the process that executed a read-only transaction, certifies it to ensure that it has not read any inconsistent data.

Now consider a DUR system which consists of a few replicas (servers), where one of them is lagging behind. Client $c_1$ interacts with an up-to-date replica, while client $c_2$ interacts with the lagging one. Suppose that $c_1$ issues an update request (an updating transaction) and receives feedback from its replica. This update does not reach $c_2$’s replica because of the lag. If the two clients communicate with each other directly, $c_2$ may notice it is missing an update, or even worse, it may not notice, but still take actions which depend on the operations issued by $c_1$. Indeed, if $c_2$ starts a new transaction on the lagging replica, it will not be able to observe the update it was already informed about. If the transaction undergoes certification, it will be aborted, but still up to some point in time (possibly until the commit attempt) the transaction will be executed on a stale snapshot. Moreover, if the transaction executed by $c_2$ is a query (a read-only transaction), the transaction may even commit (since no inter-process synchronization is required for read-only transactions). This clearly stands in contrast with the real-time order requirement.

The real-time order requirement on transaction execution is relatively easy to provide in a local environment. That is why the most prevalent safety properties for transactional systems such as opacity and TMS1, originally proposed for local transactional memory systems, require that the real-time order on transaction execution is always respected. On the other hand, the existing properties that do not require real-time order are too weak: i.e., serializability provides no guarantees on live or aborted transactions, and update serializability or extended update serializability do not require all processes to witness

---

1To prevent live transactions from reading stale data, the system would have to run a distributed consensus round before the start of each transaction.
all updates in the system in the same order (for further discussion see Section 2).

Since, as explained above, none of these properties fit the realm of distributed transactional systems, new properties are needed.

In this paper we propose $\diamond$-opacity, a family of closely related, opacity-based safety properties that relax to various degree the real-time order requirement on transaction execution. At the same time, they preserve other requirements of opacity that are essential for transactional memory systems, e.g., that transactions never read an inconsistent state. We prove that DUR satisfies update-real-time opacity (a member of the $\diamond$-opacity family), which allows read-only and aborted transactions to operate on stale but consistent data.

Alongside $\diamond$-opacity, we introduce a family of safety properties called $\diamond$-linearizability, which formalize the guarantees offered by strongly consistent systems that do not feature transactional semantics. We show that $\diamond$-linearizability preserves two important properties of linearizability: locality and nonblocking.

We also give a formal result on the relationship between $\diamond$-opacity and $\diamond$-linearizability. In order to establish a link between the two families we introduce a so-called gateway object. We show that, roughly speaking, when transactions are hidden from clients, a $\diamond$-opaque replicated system is $\diamond$-linearizable. In particular, this result establishes a formal relationship between opacity and linearizability (in their original definitions). Our approach to comparing properties using a gateway object is general enough to be applied to other transactional and non-transactional properties.

1.1. Contributions

The main contributions of our work are the following:

- We formally define $\diamond$-opacity, a family of properties based on opacity which relax the real-time order requirement of opacity to a various degree. In particular, we allow read-only and aborted transactions to execute on stale, but still consistent data. The strongest property of $\diamond$-opacity is equivalent to the original definition of opacity. We prove that all members of $\diamond$-opacity are safety properties (non-empty, prefix-closed and limit-closed).
• We formally define \(\diamond\)-linearizability, a family of properties similar to \(\diamond\)-opacity, but suited for systems modelled as shared objects and which do not feature transactional semantics. The strongest property of \(\diamond\)-linearizability is equivalent to the original definition of linearizability. As with \(\diamond\)-opacity, we show that all members of \(\diamond\)-linearizability are safety properties. We also prove that each member of \(\diamond\)-linearizability is a non-blocking local property, similarly to linearizability in its original definition.

• We establish a formal relationship between \(\diamond\)-opacity and \(\diamond\)-linearizability. We show that when requests are executed as transactions in a \(\diamond\)-opaque system and transactions appear invisible to the processes that issue the requests, the system is \(\diamond\)-linearizable. In particular, we show the relationship between opacity and linearizability in their original definitions (to our knowledge it is the first result of this sort).

• We show that relaxing real-time order for aborted operation executions in linearizability does not result in a weaker safety property. It means that a transactional system that guarantees that every transaction that read stale data is eventually aborted, is still (real-time) linearizable given that transactions are hidden from the processes.

• We demonstrate the usefulness of the new properties by proving that DUR satisfies update-real-time opacity, which allows aborted and read-only transactions to operate on stale, but still consistent data. We also show how DUR can be used to build a shared object that is update-real-time linearizable (i.e., an object which only requires operations that change the state of the object to respect the real-time order).

Preliminary results of this work have been presented in [8] (where we gave the definition of \(\diamond\)-opacity and explained how DUR satisfies update-real-time opacity but without giving a formal proof of correctness).
1.2. Paper structure

The rest of the paper is structured as follows. In Section 2 we discuss the work relevant most closely to ours. Next, in Section 3 we formalize the system model. Then, in Sections 4 and 5 we provide the formal definitions of \(\Diamond\)-opacity and \(\Diamond\)-linearizability family of properties and establish the relationship between them. Next, in Section 6 we describe DUR and show that it guarantees update-real-time opacity. Finally, we conclude in Section 7.

2. Related Work

In this section we present work relevant to our research. We start with various correctness criteria defined for strongly consistent transactional systems. Then we focus on semantics of non-transactional systems. Next, we outline two important categories of correctness conditions and show how our new properties relate to them. Finally, we discuss example optimistic concurrency control schemes for which the properties introduced in this paper are applicable.

2.1. Correctness criteria for transactional systems

Several correctness criteria have been proposed to formalize the guarantees offered by transactional systems. Below we describe the most relevant ones. We show that none of them allows for relaxed time-ordering of transactions and at the same time provides sufficiently strong consistency guarantees that match those of real practical systems (such as those based on DUR). Next, we proceed to describe how our correctness criteria relate to others.

Serializability \([5]\) specifies that all committed transactions are executed as if they were executed sequentially by a single process. Strict serializability \([5]\) additionally requires that the real-time order of transaction execution is respected (i.e. the execution order of non-overlapping committed transactions is preserved). Update serializability \([6]\) is very similar to serializability, but allows read-only transactions to observe different (but still legal) histories of the already committed transactions.
All three properties mentioned above regard only committed transactions and say nothing about live or aborted transactions. Therefore, they are not sufficient to describe behaviour of some transactional systems, especially when transactions may perform arbitrary operations and reading an inconsistent state may lead to erroneous behaviour. Therefore, new correctness criteria emerged that formalize the behaviour of all transactions in the system, including the live ones. Some of the correctness criteria, such as recoverability, avoiding cascading aborts and strictness [9], specify the behaviour of read and write operations for both live and completed transactions. However, they say nothing about global ordering of transactions (unlike serializability and properties similar to it). This, in turn, limits their usefulness in the context of strongly consistent transactional systems. Therefore, our attention focuses on properties that (in most cases) maintain a global serialization for all transactions.

The following properties maintain a global serialization only for a subset of transactions. Extended update serializability [7] ensures update serializability for both committed and live transactions. Therefore, it features a global serialization for all the updating transactions (read-only transactions may observe a different serialization). Virtual world consistency [10] allows an aborted transaction to observe a different (but still legal) history.

Opacity [11] [3], which features a global serialization of all transactions, extends strict serializability in a way that live transactions are always guaranteed to read a consistent state. In a sense, it is thus similar to extended update serializability which adds similar guarantees to update serializability. Rigorousness [12], TMS2 [4] and DU-opacity [13] offer even stronger guarantees. They restrict some particular sets of histories compared to opacity. Rigorousness and TMS2 impose stronger requirements on the ordering of concurrent transactions. DU-opacity explicitly requires that no read operation ever reads from a commit-
pending transaction which later aborts, unless the transaction which executes
the read aborts itself. Moreover, all these three properties are only defined in
a model that assumes read-write registers. TMS1 [4], which was proposed to
slightly relax opacity, allows not only each transaction, but even each opera-
tion, to observe a different view of past transactions. The possible histories
are, however, restricted by a few conditions which enforce quite strong consist-
tency (despite the lack of a global serialization). All of the properties mentioned
above, including opacity, require that the real-time order of transaction execu-
tion is respected, similarly as in case of strict-serializability, thus they are not
applicable to systems considered in this paper.

The ♦-opacity family of properties, which we introduce in this paper, relaxes
real-time order requirements on transaction execution in opacity to various de-
gree. The global serialization of all transactions as well as other characteristics
of opacity (such as a guarantee for a transaction to always read a consistent
state) are still maintained. Currently, ♦-opacity features six members, ordered
by the strength of the offered guarantees. The strongest property of them is
real-time opacity which is equivalent to opacity defined in [3]. Commit-real-time
opacity allows live and aborted transactions to read stale (but still consistent)
data. Write-real-time opacity further relaxes the real-time order guarantees on
transactions that are known a priori to be read-only (i.e., before they commence
execution). Update-real-time differs from write-real-time opacity by allowing all
read-only transactions to not respect real-time order. Program order opacity
requires real-time order only for transactions executed by the same process. In
this sense, this property is similar to virtual time opacity (VTO) [11]. How-
ever, unlike VTO, program order opacity does not require that the serialization
respects the causal order of transaction execution across all processes. Finally,
arbitrary order opacity makes no assumptions on the relative ordering of trans-
actions. In this respect, it is similar to serializability. However, contrary to
serializability, arbitrary order opacity also ensures that live transactions always
observe a consistent view of the system’s state.
2.2. Correctness criteria for non-transactional systems

For strongly consistent replicated systems that do not feature transactional semantics, we define $\diamond$-linearizability, a family of properties similar to $\diamond$-opacity, but based on linearizability [14]. Unlike many other correctness criteria for (systems modeled as) shared objects, such as PRAM consistency, cache consistency, slow consistency, etc. [15], linearizability maintains a global serialization of all operation executions, similarly as opacity or strict serializability do so for transactions. Linearizability also requires that the real-time order on operation execution is respected at all times. This trait makes it unsuitable for distributed environment. Analogically to $\diamond$-opacity, members of $\diamond$-linearizability differ in the ordering guarantees imposed on the serialization of operation executions. Real-time linearizability, the strongest property of the family, is equivalent to the original definition of linearizability. Interestingly, final-state program order linearizability is equivalent to sequential consistency [16].

In part, we use $\diamond$-linearizability to formalize the behaviour of a replicated transactional system where transactions are invisible to the clients. This way, every client request is considered as a separate operation that needs to be linearized. A recent study [17] used linearizability as a safety property for a Software Transactional Memory system, in which every invocation of a transaction is executed exactly once and transaction aborts are hidden from the programmer. Conversely, in our approach, aborts are exposed to the processes. In particular, we model transactional systems as abortable objects (defined below).

Objects which in case of contention may return special responses *fail* or *pause* for any operation invocation when there is contention, appeared first in [18] under the name *obstruction-free objects*. The special response *fail* indicates that the operation was not applied and the process is free to invoke any other operation. On the other hand, *pause* means that the implementation is uncertain whether the operation had any effect on the object. The authors show that the objects that may return *fail* but not *pause*, cannot be implemented from only read/write registers. The work in [19] introduced the notions of abortable and query-abortable objects. In this, approach a special response *abort* always
implies uncertainty whether an operation was applied or not. On the other hand, query-abortable objects provide a special query operation which allows processes to determine their last operation that caused a state transition of the object. Deterministic abortable objects [20] feature only one special response abort, which always indicates that the operation did not take an effect. As noted before, such objects cannot be implemented from only read/write registers. The authors in [20] investigate their computational power and the implicit hierarchy they form. The notion of deterministic abortable objects is now well-accepted by the community. Therefore, unless noted otherwise, we mean deterministic abortable objects when we say abortable objects.

The original definitions of obstruction-free, abortable and query-abortable objects required that the objects return special responses only when contention is encountered. This way trivial implementations which never return regular responses are prohibited. Among the most popular measures of contention are step-contention [18] and interval-contention [19]. In this paper, we place no restrictions on the conditions when objects may return the response abort. We do so for two main reasons. First, we believe that this is an orthogonal problem, which is related to progress rather than safety. Secondly, we use an abortable object as a facade for a transactional system. Since transactional systems usually provide their own progress guarantees (e.g., progressiveness, global progress or obstruction-freedom [3]), they would translate into the properties of the facade object (in particular, an obstruction-free transactional memory would impose requirement on special responses of the facade object to occur only upon encountering step-contention).

2.3. Safety and liveness properties

Some argue that it is useful to distinguish two classes of correctness properties: liveness and safety properties (see, e.g., [21] [22]). Although this distinction does not exhaust the whole spectrum of correctness properties (e.g., serializability is neither a safety nor a liveness property), it is useful, as it captures important and radically different facets of the system’s correctness. Moreover,
showing safety and liveness requires different proving techniques.

Informally, a liveness property ensures that something good eventually happens during an execution. On the other hand, a safety property guarantees that nothing bad ever does. This trait can be formalized by requiring that the property is non-empty, prefix-closed and limit-closed. We formally prove that all members of $\diamond$-opacity and $\diamond$-linearizability families are indeed safety properties.

2.4. Optimistic replication schemes

We demonstrate the usefulness of the new properties, which we define in the paper, by using Deferred Update Replication (DUR) [1] (see Section 3). DUR is the most basic optimistic replication protocol for multi-primary-backup replication. Various flavors of DUR are implemented in several commercial database systems, including Ingress, MySQL Cluster and Oracle. These implementations use 2PC [9] as the atomic commitment protocol. In this paper, we consider DUR based on Total Order Broadcast (TOB) [2]. This approach is advocated by several authors because of its nonblocking nature and predictable behaviour (see [23, 24, 25] among others). Most recently, it has been implemented in D2STM [26] and Paxos STM [27, 28]. It has also been used as part of the coherence protocols of S-DUR [29] and RAM-DUR [30].

There are a number of optimistic replication protocols that have their roots in DUR, e.g., Postgres-R [31] and Executive DUR [32]. The differences between these systems lie not in the general approach to processes synchronization, but in the way the transaction certification is handled. Both protocols, similarly to DUR, certify read-only transactions without inter-process synchronization. It means that these protocols are not opaque. However, one can show that they guarantee update-real-time opacity.

3. System model

In this section, we introduce some basic notions used in the following sections to define $\diamond$-opacity and $\diamond$-linearizability. Our formal framework closely follows the one of opacity from [3]. We also borrow some definitions from [33].
We model the system as a set of *shared objects* \( \mathcal{X} = \{X_1, X_2, \ldots \} \) accessible by a finite set \( \mathcal{P} = \{p_1, p_2, \ldots, p_n \} \) of \( n \) processes. Each shared object (or simply an *object*) has a unique identity and a type. Each type is defined by a *sequential specification* that consists of:

- a set \( Q \) of possible states for an object,
- an initial state \( q_0 \in Q \),
- a set \( \text{INV} \) of operations that can be applied to an object,
- a set \( \text{RES} \) of possible responses an object can return, and
- a transition relation \( \delta \subseteq Q \times \text{INV} \times \text{RES} \times Q \).

This specification describes how an object of a given type behaves, if the object is accessed one operation at a time. If \( (q, op, res, q') \in \delta \), it means that applying operation \( op \) to an object in state \( q \) may move the object to state \( q' \) and the response \( res \) is returned to the process that invoked \( op \). For simplicity, we assume that operation arguments are encoded in the operation itself.

We say that an operation \( op \) is *total*, if it is defined for every possible state of an object. More formally, \( op \) is total, if and only if for every \( q \in Q \), there exists \( (q, op, res, q') \in \delta \).

We say that an operation \( op \) is *updating* for a given state \( q \in Q \), if and only if there exists \( (q, op, res, q') \in \delta \), such that \( q \neq q' \). We say that \( op \) is *read-only*, if and only if there does not exist a state \( q \), for which \( op \) is updating. Otherwise, we say that \( op \) is *potentially updating*.

Let us consider an example. *Read/write registers* constitute an important class of shared objects. A read/write register features two simple operations: \( \text{read} \), which returns the current integer value \( v \in \mathbb{Z} \) of the register, and \( \text{write}(i) \) for some \( i \in \mathbb{Z} \), which sets the current value of the register to \( i \) and returns \( \text{ok} \) afterwards. Then, we define the sequential specification of read/write registers as \( T_R = (\mathbb{Z}, 0, \text{INV}_R, \text{RES}_R, \delta_R) \), where \( \text{INV}_R = \{\text{read}\} \cup \{\text{write}(i) : i \in \mathbb{Z}\} \), \( \text{RES}_R = \{\text{ok}\} \cup \mathbb{Z} \), and \( \delta_R = \{(i, \text{read}, i, i) : i \in \mathbb{Z}\} \cup \{(i, \text{write}(j), \text{ok}, j) : i, j \in \mathbb{Z}\} \).
Both the read and write operations are total. Obviously, read is a read-only operation as it never modifies the state of the register, and write is not (write is potentially updating). However, write is not always updating. When the register is in state \( i \) (its value is \( i \)), then write\((i)\) does not change its state. Therefore, write\((i)\) is not updating for state \( i \).

Shared objects form a hierarchy, at the bottom of which are base objects. Base objects represent individual memory locations which can be read and written. Each higher level of the hierarchy contains shared objects of more abstract types. Implementations of objects at a given level may only use base objects and other lower-level shared objects.

Implementations of shared objects are defined by algorithms which describe the steps required to complete each operation. When a process invokes an operation on a shared object, it follows an appropriate algorithm. In each step, a process may either perform local computation or engage other object.

We assume that every shared object encapsulates its state, i.e., each object or local variable is used only by a single shared object implementation.

When a process \( p_i \) executes an operation \( op \) on object \( X \in X \), it invokes an event \( X.inv_i(op) \) and expects a response event \( X.resp_i(v) \). A pair of such events is called a (completed) operation execution and is denoted by \( X.op \rightarrow_i v \). An invocation event, that is not followed by a response event, is called a pending operation execution. \( X.op \rightarrow_i v \) is, respectively, read-only, potentially updating or updating, if \( op \) is read-only, potentially updating or updating.

We model the system execution as a (totally ordered) sequence of events called a history. Naturally, histories respect program order (events executed by the same process are ordered according to their execution order), and also causality between events across processes (if two events executed in the system are causally related, one will precede the other in the history). Events that happen in parallel (in separate processes), and that are not causally dependent, can appear in a history in an arbitrary order.\(^3\) For any history \( H \), we denote by

\(^3\)An alternative approach to model the system execution would be to employ partially
the restriction of $H$ to events issued or received by process $p_i$. We denote by $H|X$ (where $X \in \mathcal{X}$) the restriction of $H$ to operations executed on $X$ and their appropriate responses, and by $H|S$ (where $S \subseteq \mathcal{X}$) the restriction of $H$ to operations executed on objects from the set $S$ and their appropriate responses.

A history, which is not restricted to any particular object or a set of objects, is called an implementation history. On the other hand, a high-level history is a history $H|S$ restricted to a set of objects $S = \{X_1, X_2, \ldots\}$, such that no object $X_i$ is used by the implementation of some other object $X_j$. Unless stated otherwise, when we say history we mean a high-level history.

An implementation history $H$ is said to be well-formed, if for every process $p_i$ and every shared object $X$, $(H|X)|p_i$ is a (finite or infinite) sequence of operation executions, possibly ending with a pending operation execution. A high-level history $H$ is well-formed, if for every process $p_i$, $H|p_i$ is a (finite or infinite) sequence of operation executions, possibly ending with a pending operation execution. We consider only well-formed histories.

**4. The $\Diamond$-opacity family of properties**

In this section, we define the $\Diamond$-opacity family of safety properties. First we introduce some auxiliary definitions and then specify the properties. Finally, we provide a discussion on the properties' characteristics and show some examples.

**4.1. The formal definition**

Let us start by distinguishing a subset of shared objects $Q \subset \mathcal{X}$ called $t$-objects and defining a set $T = \{T_1, T_2, \ldots\}$ of transactions.

**ordered sets** (or simply posets). In fact, this approach is equivalent to ours, because a poset can be represented by a set of totally ordered histories (and we always analyze all possible histories a given system can produce). We argue that an approach based on totally ordered histories is more elegant, because it better matches the sequential specifications used for $t$-objects. We also want to stay close to, and remain compatible with, the original formal framework of opacity presented in [3].
The set $Q = \{x_1, x_2, \ldots\}$ of t-objects consists of a special class of shared objects that cannot be accessed directly by processes. Instead, they have to be accessed within a context called a transaction, an abstract notion fully controlled by some process. Also, an operation on a t-object cannot return a value that belongs to a set $A = \{A_1, A_2, \ldots\}$ of special return values used by the system.

Interaction between processes and t-objects is managed by a single transactional memory shared object (TM object) $M$ of the following interface:

- $M.texec(T_k, x, op) \rightarrow_i \{v, A_k\}$ which executes an operation $op$ on a t-object $x$, of some type $T = (Q, q_0, INV, RES, \delta)$, within a transaction $T_k$ and as a result produces a return value $v \in RES$ or a special value $A_k \in A$;

- $M.tryC(T_k) \rightarrow_i \{A_k, C_k\}$ which attempts to commit a transaction $T_k$, and returns the special values $A_k \in A$ or $C_k$;

- $M.tryA(T_k) \rightarrow_i A_k$ which aborts a transaction $T_k$ and returns $A_k \in A$.

Each operation on a TM object (executed by some process $p_i$) can return a special value $A_k$, which indicates that the transaction $T_k$ that executed this operation has aborted. The value $C_k$, returned by the operation $tryC(T_k)$, means that $T_k$ has committed. For any t-object of type $T = (Q, q_0, INV, RES, \delta)$, $A_k \notin RES$ and $C_k \notin RES$. A response event with a return value $A_k$ or $C_k$ is called, respectively, an abort event or commit event (of transaction $T_k$). The commit or abort events of transaction $T_k$ are always the last events for $T_k$.

We say that a transaction $T_k$ performs some action, when a given process executes this action as part of $T_k$. A transaction that only executes read-only operations in a given history is called a read-only transaction (in that history). Otherwise, we say that it is an updating transaction. In general, during an execution of a transaction, it is impossible to tell whether it will not perform any updating operations before it finishes its execution. However, we distinguish a special class of transactions called declared read-only (DRO), which are known \textit{a priori} to be always read-only in any history (they are allowed to execute only read-only operations on t-objects). Then, for any such transaction $T_k$, we write
Every transaction $T_k$ for which $\text{DRO}(T_k) = \text{true}$ is read-only, but the opposite is not necessarily true. A transaction that is not declared read-only is called a potentially updating transaction. Every updating transaction is potentially updating, and every read-only transaction that is not declared read-only is also potentially updating (no matter whether the read-only transaction is still live or already completed).\footnote{The distinction between read-only and declared read-only (and between updating and potentially updating) transactions is relevant for some systems and is reflected in different ordering relations which we define later. Sometimes, it is useful to treat a potentially updating transaction as an updating one, even though it does not produce any updates. Such a transaction resembles a write operation which is not always updating.}

Let $H_f$ be an implementation history and $M$ be a TM object. Then, we use the term $t$-history for a high-level history $H_f|M$. Let $H$ be a $t$-history of some TM object $M$. We denote by $H|T_k$ the restriction of $H$ to events concerning $T_k$, i.e. invocation events of operations $M.\text{texec}(T_k, x.\text{op})$, $M.\text{tryC}(T_k)$, $M.\text{tryA}(T_k)$ and their corresponding response events (for any t-object $x$ and operation $\text{op}$).

We say that a transaction $T_k$ is in $H$, if $H|T_k$ is not empty. Let $x$ be any t-object. We denote by $H|x$ the restriction of $H$ to events concerning $x$, i.e., the invocation events of any operation $M.\text{texec}(T_k, x.\text{op})$ and their corresponding response events (for any transaction $T_k$ and operation $\text{op}$ on $x$).

We say that a transaction $T_k$ is committed in $H$, if $H|T_k$ contains operation execution $M.\text{tryC}(T_k) \rightarrow C_k$ (for some process $p_i$). We say that the transaction $T_k$ is aborted in $H$, if $H|T_k$ contains response event $\text{resp}_i(A_k)$ from any operation of $T_k$ (for some process $p_i$). If an aborted transaction $T_k$ contains an invocation of the operation $M.\text{tryA}(T_k)$, it is said to be aborted on demand. Otherwise, we say that $T_k$ is forcibly aborted in $H$. A transaction $T_k$ in $H$ that is committed or aborted is called completed. A transaction that is not completed is called live. A transaction $T_k$ is said to be commit-pending in $H$, if $H|T_k$ has a pending operation $M.\text{tryC}(T_k)$ ($T_k$ invoked the operation $M.\text{tryC}(T_k)$, but has not received any response from this operation).
We say that a t-history $H$ is $t$-completed, if every transaction $T_k$ in $H$ is completed. A $t$-completion of a t-history $H$ is any (well-formed) t-completed t-history $ar{H}$ such that:

1. $H$ is a prefix of $ar{H}$, and
2. for every transaction $T_k$ in $H$, sub-history $ar{H}|T_k$ is equal to one of the following histories:

- $H|T_k$, when $T_k$ is completed, or
- $H|T_k \cdot \langle \text{tryA}(T_k) \rightarrow A_k \rangle$, for some process $p_i$, when $T_k$ is live and there is no pending operation in $H|T_k$, or
- $H|T_k \cdot \langle \text{resp}_i(A_k) \rangle$, when $T_k$ is live and there is a pending operation in $H|T_k$ invoked by some process $p_i$, or
- $H|T_k \cdot \langle \text{resp}_i(C_k) \rangle$, when $T_k$ is commit-pending for some process $p_i$.

Let $T_i$ and $T_j$ be any two transactions in some t-history $H$, where $T_i$ is completed. We define the following order relations on transactions in $H$:

- **real-time order** $\prec^r_H$ — we say that $T_i \prec^r_H T_j$ (read as $T_i$ precedes $T_j$), if the last event of $T_i$ precedes the first event of $T_j$;

- **commit-real-time order** $\prec^c_H$ — we say that $T_i \prec^c_H T_j$, if (1) $T_i \prec^r_H T_j$, and (2) both $T_i$ and $T_j$ are committed, or both $T_i$ and $T_j$ are executed by the same process $p_k$;

- **write-real-time order** $\prec^w_H$ — we say that $T_i \prec^w_H T_j$, if (1) $T_i \prec^r_H T_j$, and (2) both $T_i$ and $T_j$ are potentially updating and are committed, or both $T_i$ and $T_j$ are executed by the same process $p_k$;

- **update-real-time order** $\prec^u_H$ — we say that $T_i \prec^u_H T_j$, if (1) $T_i \prec^r_H T_j$, and (2) both $T_i$ and $T_j$ are updating and are committed, or both $T_i$ and $T_j$ are executed by the same process $p_k$;

- **program order** $\prec^p_H$ — we say that $T_i \prec^p_H T_j$, if $T_i \prec^r_H T_j$ and both $T_i$ and $T_j$ are issued by the same process $p_k$;
Let $H$, $H'$ be two t-histories. We say that $H'$ respects the $\diamond$-order of $H$ (where $\diamond$ is any of the order relations specified above), if and only if $\prec_H^\diamond \subseteq \prec_H'^\diamond$.

For any t-history $H$ the following holds: $\emptyset = \prec_H^\diamond \subseteq \prec_H^a \subseteq \prec_H^p \subseteq \prec_H^u \subseteq \prec_H^w \subseteq \prec_H^c \subseteq \prec_H^r$.

We say that $T_i$ and $T_j$ are concurrent if neither $T_i \prec_H^r T_j$ nor $T_j \prec_H^r T_i$. We say that any t-history $H$ is t-sequential, if $H$ has no concurrent transactions.

Let $S$ be any t-completed t-sequential t-history, such that every transaction in $S$, possibly except the last one, is committed. We say that $S$ is t-legal, if for every t-object $x$, the subhistory $S[x] = \langle \text{exec}(T_i, x, op_1) \rightarrow_k \text{res}_1, \text{exec}(T_j, x, op_2) \rightarrow_l \text{res}_2, \ldots \rangle$ (for any processes $p_k, p_l, \ldots$, and for any transactions $T_i, T_j, \ldots$) satisfies the sequential specification $(Q, q_0, INV, RES, \delta)$ of $x$, i.e., there exists a sequence of states $q_1, q_2, \ldots$ in $Q$, such that $(q_{n-1}, op_n, \text{res}_n, q_n) \in \delta$ for any $n$.

Let $S$ be any t-completed t-sequential t-history. We denote by $\text{visible}_S(T_k)$ the longest subsequence $S'$ of $S$, such that for every transaction $T_i$ in $S'$, either

1. $i = k$, or
2. $T_i$ is committed and $T_i$ precedes $T_k$. We say that a transaction $T_k$ in $S$ is legal in $S$, if the t-history $\text{visible}_S(T_k)$ is t-legal.

We say that t-histories $H$ and $H'$ are equivalent, denoted $H \equiv H'$, if for every transaction $T_k$ in $\mathcal{T}$, $H[T_k] = H'[T_k]$.

**Definition 1.** A finite t-history $H$ is final-state $\diamond$-opaque, if there exists a t-sequential t-history $S$ equivalent to some t-completion of $H$, such that:

1. every transaction $T_k$ in $S$ is legal in $S$, and
2. $S$ respects the $\diamond$-order of $H$.

**Definition 2.** A t-history $H$ is $\diamond$-opaque, if every finite prefix of $H$ is final-state $\diamond$-opaque.

**Definition 3.** A system (modelled as a TM object) is $\diamond$-opaque, if every history it produces is $\diamond$-opaque.

In the above three definitions $\diamond$ can be either real-time, commit-real-time, write-real-time, update-real-time, program order, or arbitrary order. Therefore,
Figure 1: Example t-histories for two processes and up to three transactions. $H_a$ is commit-real-time opaque, but not real-time opaque. $H_b$ is update-real-time opaque, but not commit-real-time opaque. Additionally, $H_b$ is write-real-time opaque, if and only if $DRO(T_3)$ holds true in $H_b$. $H_c$ is program order opaque, but not update-real-time opaque. $H_d$ is arbitrary order opaque, but not program order opaque. We use simplified notation: explicit calls to the TM object $M$ as well as process numbers are omitted, lower indices indicate the transaction number, $rd$ is a read operation and $wr$ is a write operation.

we obtain a whole family of $\diamond$-opacity properties. Real-time opacity is equivalent to opacity. By substituting real-time with weaker ordering guarantees we obtain gradually weaker properties with arbitrary order opacity being the weakest one.

4.2. Discussion

Real-time opacity, which is equivalent to the original definition of opacity \cite{3}, requires that all transactions, regardless of their state of execution (live, aborted, commit-pending or committed) always observe a consistent and most recent view of the system. Commit-real-time opacity relaxes opacity, by restricting the real-time order to only committed transactions (aborted transactions may observe stale, but still consistent data). Write-real-time opacity and update-real-time opacity additionally relax the real-time order requirement on transactions that, respectively, are known a priori to be read-only (are declared read-only), or that have not performed any potentially updating operations (are read-only). Program order opacity ensures that transactions respect program
order (i.e. the order of execution of all local transactions has to be respected across all processes). Finally, arbitrary order opacity imposes no requirements on the order of transactions’ execution, as long as all transactions are legal.

Write-real time opacity is suitable only for systems that can distinguish between transactions that have not performed any potentially updating operations and transactions known \textit{a priori} to be read only (only for the latter ones the DRO predicate holds). In such systems, the additional information about transactions can be either provided by the programmer or can be deduced prior to a transaction execution from the transaction code. By manually marking some transactions as declared read-only, a programmer can decide whether a read-only transaction $T_k$ may read-stale data ($DRO(T_k)$ holds) or has to respect the real-time order ($DRO(T_k)$ does not hold). We can make the following two simple observations which follow directly from the definitions. Firstly, in a write real-time opaque history $H$, if there are no transactions for which the DRO predicate holds, $H$ is also commit-real-time opaque. Secondly, in an update real-time opaque t-history $H$, if for all read-only transactions the predicate $DRO$ holds, then $H$ is also write real-time opaque.

Note that not every updating transaction modifies the state of any t-object. Consider a transaction $T_k$, which first reads value $i$ from a register $x$ and then writes the same value $i$ to $x$, or just happens to write $i$ to $x$ without reading $x$ beforehand. Clearly, $T_k$ is not a read-only transaction as it executes a write operation (which is a potentially updating operation). However, in this particular scenario, the write is not updating, because the state of $x$ remains unchanged.

A similar example can be formulated for transactions that execute operations on any objects featuring operations that are not always updating (see Section 3).

Our definition of update-real-time does not differentiate between transactions that performed potentially updating operations, which modified state of some t-objects, and those, which just happened not to change the value of the t-objects, as in the example above. Doing so would result in a correctness property that has no real practical application. None of the real-world transactional systems we are aware of treat differently these two kinds of transactions.
and neither does DUR (see Section 6). On the contrary, making such a distinction is necessary when considering sequences of operation executions, as in ♦-linearizability (see Section 5).

Figure 1 illustrates the relations between the members of the ♦-opacity family by example. It depicts four t-histories (two variants of t-history $H_b$ can be deduced depending on the value of $DRO(T_3)$). Each t-history represents a case when one property is satisfied while another, a stronger one, is not.

T-histories $H_a$ and $H_b$ represent our main motivation: enabling aborted and read-only transactions to read from a stale (but consistent) snapshot. Let us first consider $H_a$. Transactions $T_1$ and $T_2$ access the same t-object $x$. $T_1$ precedes $T_2$, however $T_2$ reads a stale value of $x$, and subsequently aborts. The only possible serialization of $H_a$ in which all transactions are legal is $\langle H_a | T_2 \cdot H_a | T_1 \rangle$. This serialization does not respect the real-time order, as clearly $T_1 \prec_r H_a T_2$. Therefore, $H_a$ breaks (real-time) opacity. It satisfies, however, commit-real-time opacity, because $T_2$ is aborted and it may observe stale data.

In t-history $H_b$, transaction $T_3$, which does not perform any updating operations, is preceded by transaction $T_2$. However, $T_3$ does not observe the operation $x.wr_2(2)$ of $T_2$, as its operation $x.rd_3$ returns the value written by $T_1$. Therefore, $H_b$ breaks real-time opacity. Moreover, it breaks commit-real-time opacity. On the other hand, $H_b$ satisfies write-real-time opacity when $DRO(T_3)$ holds, and update-real-time opacity when $DRO(T_3)$ does not hold.

In t-history $H_c$, similarly as in the previous example, $T_3$ does not obey the real-time order. This time, however, $T_3$ is an updating transaction. Therefore, the history only satisfies program order opacity and not update-real-time opacity, nor any stronger property. Finally, in t-history $H_d$, even the program order is not preserved, as $p_2$’s transaction $T_3$ does not observe the effects of another transaction ($T_2$) executed by $p_2$ earlier. This t-history, however, satisfies arbitrary order opacity, as transactions $T_2$ and $T_3$ can be reordered, yielding an equivalent legal execution ($\langle T_3 \cdot T_1 \cdot T_2 \rangle$). This trait makes arbitrary order opacity similar to serializability.

Now we show that ♦-opacity is a safety property \[21\] \[22\], similarly as opac-
ity. The proofs of all formal results can be found in the Appendix.

**Theorem 1.** $\Diamond$-opacity is a safety property.

**Corollary 1.** A system (modelled as a TM object) is $\Diamond$-opaque if, and only if every finite history it produces is final-state $\Diamond$-opaque.

5. The $\Diamond$-linearizability family of properties

In this section we introduce the $\Diamond$-linearizability family of safety properties. We begin by giving a formal definition, followed by a discussion on characteristics of $\Diamond$-linearizability. Finally, we show the relationship between $\Diamond$-linearizability and $\Diamond$-opacity.

5.1. The formal definition

We provide the definition of $\Diamond$-linearizability for systems where shared objects may be abortable. It means, that any operation $op$ invoked on an abortable shared object $X \in \mathcal{X}$ (for some operation execution $o$), may return a special value $\bot$, thus indicating that the execution of $op$ failed and did not change $X$’s state. More precisely, in the sequential specification of an abortable object, the transition $(q, op, \bot, q)$ is possible for any operation $op$ and state $q$. In such case, we say that $op$ aborted. Otherwise, i.e., when $op$ returns some value $v \neq \bot$, we say that $op$ committed. An operation execution $X. op \rightarrow_t v$ is, respectively, aborted or committed, if $op$ aborted or committed.

A history which has no pending operation executions is called completed. A completion of a history $H$ is any (well-formed) completed history $\bar{H}$ such that $\bar{H}$ consists of all completed operation executions from $H$ and appropriate response events for a subset of pending operation executions in $H$ (i.e., some pending operations execution in $H$ appear completed in $\bar{H}$ or do not appear at all).

\footnote{Note that the definition of completion and t-completion from Section 4 are quite different. T-completion is always a prefix of the original t-history, whereas a completion may lack some}
Let $o_i$ and $o_j$ be any two operation executions in some history $H$, where $o_i$ is a completed operation execution. We define the following order relations on operations in $H$:

- **real-time order** $<^r_H$ — we say that $o_i <^r_H o_j$ (read as $o_i$ precedes $o_j$), if the response event of $o_i$ precedes the invocation event of $o_j$;

- **commit-real-time order** $<^c_H$ — we say that $o_i <^c_H o_j$, if (1) $o_i <^r_H o_j$, and (2) both $o_i$ and $o_j$ committed, or both $o_i$ and $o_j$ are executed by the same process $p_k$;

- **write-real-time order** $<^w_H$ — we say that $o_i <^w_H o_j$, if (1) $o_i <^r_H o_j$, and (2) both $o_i$ and $o_j$ are potentially updating and committed, or both $o_i$ and $o_j$ are executed by the same process $p_k$;

- **program order** $<^p_H$ — we say that $o_i <^p_H o_j$, if $o_i <^r_H o_j$ and both $o_i$ and $o_j$ are issued by the same process $p_k$;

- **arbitrary order** $<^a_H$ — equivalent to $\emptyset$. Never $o_i <^a_H o_j$ holds true.

Let $H, H'$ be two histories. We say that $H'$ respects the $\Diamond$-order of $H$, if and only if $<^\Diamond_H \subseteq <^\Diamond_H'$. For any history $H$ the following holds: $\emptyset = <^a_H \subseteq <^p_H \subseteq <^w_H \subseteq <^r_H \subseteq <^c_H \subseteq <^\Diamond_H$.

We say that $o_i$ and $o_j$ are concurrent, if neither $o_i <^r_H o_j$ nor $o_j <^r_H o_i$. We say that any history $H$ is sequential, if $H$ has no concurrent operation executions.

Let $S$ be any completed sequential history. We say that $S$ is legal, if for every object $X$, the subhistory $S|X = \langle X.op_1 \rightarrow_k v_1, X.op_2 \rightarrow_l v_2, \ldots \rangle$ (for any processes $p_k, p_l, \ldots$) satisfies the sequential specification $(Q,q_0,INV,RES,\delta)$ of the events of the original history. Such a formulation is necessary to account for non-abortable objects which have some non-total operations. Consider a history of a shared object implementing a blocking queue. Let us assume that the history contains a pending `dequeue` operation and no `enqueue` operations. Clearly, it is impossible to complete this history by adding some (legal) return event for the `dequeue` operation. Yet, it is still possible that a future arrival of an `enqueue` operation will allow the pending `dequeue` operation to complete. Therefore, the definition of completion has to be more admitting.
of $X$ such that there exists a sequence $W = \langle q_0, q_1, \ldots \rangle$ of states in $Q$ and 
$(q_{n-1}, op_n, v_n, q_n) \in \delta$ for any $n$. We call any such a sequence a witness history of $S$. For any operation execution $o_n = X. op_n \rightarrow_k v_n$ in $S$, we say that $o_n$ is 
updating in $S$ according to $W$, if $q_{n-1} \neq q_n$.

We say that histories $H$ and $H'$ are equivalent, denoted $H \equiv H'$, if $H'$
contains all of the events of $H$ and vice versa.

We distinguish yet another order relation whose definition we give below.

Let $H$ be any history and $S$ be a legal sequential history equivalent to some 
completion of $H$. Let $W$ be some witness history of $S$. Let $o_i$ and $o_j$ be any two 
operation executions in $H$, where $o_i$ is a completed operation execution. We 
define the following order relation on operations in $H$ (according to $S$ and $W$):

- update-real-time order $<^u_H(S, W)$ — we say that $o_i <^u_H(S, W) o_j$, if (1) 
  $o_i <^r_H o_j$, and (2) both $o_i$ and $o_j$ are updating in $S$ according to $W$ and are 
  committed, or both $o_i$ and $o_j$ are executed by the same process $p_k$.

Let $H, H'$ be two histories. We say that $H'$ respects the update-real-time 
order of $H$ according to $S$ and $W$, if and only if $<^u_H(S, W) \subseteq <^u_H(S, W)$. When 
$H' = S$, we say that $S$ respects the update-real-time order of $H$, if and only if 
there exists $W$, a witness history of $S$, such that $<^u_H(S, W) \subseteq <^u_H(S, W)$. In 
such case, we simplify the notation and write $<^u_H \subseteq <^u_P$. For any history $H$ and 
any legal history $S$ equivalent to some completion of $H$ (with any witness history 
$W$) the following holds: $\emptyset = <^u_H \subseteq <^P_H \subseteq <^u_H(S, W) \subseteq <^u_H \subseteq <^u_H \subseteq <^r_H$.

Now, let us provide the definition of $\Diamond$-linearizability.

**Definition 4.** A finite history $H$ is final-state $\Diamond$-linearizable, if there exists a 
sequential history $S$ that:

1. $S$ is equivalent to $\bar{H}$, a completion of $H$,
2. $S$ is legal, and
3. $S$ respects the $\Diamond$-order of $\bar{H}$.

**Definition 5.** A history $H$ is $\Diamond$-linearizable, if every finite prefix of $H$ is final-
state $\Diamond$-linearizable.
Definition 6. A system (modelled as a set of shared objects) is \(\diamond\)-linearizable, if every history it produces is \(\diamond\)-linearizable.

In the above two definitions \(\diamond\) can be either real-time, commit-real-time, write-real-time, update-real-time, program order, or arbitrary order.

Note that, we require that \(S\) respects the \(\diamond\)-order of \(\bar{H}\) and not of \(H\). It is because \(S\), which is equivalent to \(\bar{H}\), may lack some of the operations that are pending in \(H\) (by the definition of completion of \(H\)).

5.2. Discussion

\(\diamond\)-linearizability relaxes the operation ordering guarantees of the original definition of linearizability [14], similarly as \(\diamond\)-opacity does it for transaction ordering guarantees of the original definition of opacity [3]. Note that we consider abortable shared objects as well as the ordinary ones. Not only this way we can encompass a wider array of systems (shared object implementations), but it also allows us to use \(\diamond\)-linearizability to describe the behaviour of transactional systems from the point of view of external clients (see Section 5.3).

The difference between write-real-time- and update-real-time linearizability is to some extent similar as between write-real-time- and update-real-time opacity. Write-real-time linearizability relaxes the real-time order requirement for read-only operations, i.e., operations that never change the state of the object, such as a read operation on a register. Update-real-time linearizability relaxes real-time also for operations that in a given execution did not change the state of the object, i.e., a nonblocking pop operation performed on an empty stack.

Now let us highlight the subtle difference in the analogy between write/update-real-time opacity and write/update-real-time linearizability. Consider an example in Figure 2. \(X\) is a shared object implementing an initially empty stack on which two processes execute push and pop operations (push is an always updating and pop is a potentially updating operation that immediately returns, if the stack is empty). These operations are executed directly on \(X\) in history \(H\). In t-history \(H_t\), processes operate on a stack t-object \(x\) in an analogous way as in \(H\), but all operations are executed within separate transactions. Trivially,
Figure 2: History $H$ is not write-real-time linearizable, but is update-real-time linearizable. T-history $H_t$ is neither write- nor update-real-time opaque.

Figure 3: History $H_a$ is update-real-time linearizable, whereas $H_b$ is not.

$H$ is not write-real-time linearizable. If it were, the first $X.pop$ would have to return 5 instead of $null$, because it was issued after $X.push(5)$ committed and write-real-time linearizability requires that committed and updating or potentially updating operations respect the real-time order of operation execution. However, $H$ is update-real-time linearizable: $H$ is completed and there exists a legal sequential history $S = (X.pop \rightarrow null, X.push(5) \rightarrow ok, X.pop \rightarrow 5)$ which is equivalent to $H$ and respects the update-real-time order of $H$ ($X.pop \rightarrow null$ does not modify the state of $X$ in this particular execution).

On the other hand, $H_t$ is neither update-real-time- nor write-real-time opaque. Even though transaction $T_2$ does not modify the state of any t-object (including $x$), we still treat $T_2$ as an updating transaction. This is because update-real-time opacity does not differentiate between transactions which performed potentially updating operations that modified the state of some t-objects, and those, which happened not to change the value of the t-objects (see Section 4.2).

Now let us focus on the way $\Diamond$-linearizability is formalized. Naturally, our property is based on linearizability [14]. However, unlike the original definition,
Figure 4: Both histories \( H_a \) and \( H_b \) are commit-real-time linearizable but also real-time linearizable. (The return value \( \bot \) means that an operation aborted.)

\( \diamond \)-linearizability is defined indirectly, through final-state \( \diamond \)-linearizability, similarly as opacity/\( \diamond \)-opacity are defined through final-state opacity/\( \diamond \)-opacity. This is because final-state \( \diamond \)-linearizability is not prefix-closed, if we consider any order relation weaker than real-time order.

To better understand why introducing the intermediate step in the definition of \( \diamond \)-linearizability is necessary, see Figure 3 with two example histories \( H_a \) and \( H_b \). Our main motivation to relax real-time order was to enable executions such as the one in history \( H_a \), where one process \( (p_1) \) modifies the state of some object and then, long after the operation committed, a lagging process \( (p_2) \) observes a stale value of this object by performing a read-only operation. Naturally, \( H_a \) is final-state update-real-time linearizable, so is \( H_b \), where one operation \((X.\text{read} \rightarrow 5)\) observes the future state of some object. Clearly, \( H_b \) represents an execution, which should not be allowed. The difference between \( H_a \) and \( H_b \) lies in the fact that every prefix of \( H_a \) is update-real-time linearizable, and there exists a prefix of \( H_b \), which is not \((H'_b = (X.\text{read} \rightarrow 5))\). Hence, we require that every finite prefix of a \( \diamond \)-linearizable history is final-state \( \diamond \)-linearizable.

Next we show that \( \diamond \)-linearizability, as \( \diamond \)-opacity, is a safety property.

**Theorem 2.** \( \diamond \)-linearizability is a safety property.

As proven in [33], linearizability is a safety property only for objects which are finitely non-deterministic. For other objects, linearizability is not limit-closed and thus is not a safety property. On the other hand, \( \diamond \)-linearizability, and thus also real-time linearizability, are trivially limit-closed for infinitely non-
deterministic shared objects. It means that real-time linearizability is equivalent to the original definition of linearizability for objects which are finitely non-deterministic.

**Corollary 2.** A system (modelled as a set of shared objects) is ♦-linearizable, if and only if every finite history it produces is final-state ♦-linearizable.

♦-linearizability preserves two important properties of linearizability: locality and non-blocking [14]. The former requires that a system is ♦-linearizable, if and only if every shared object managed by the system is ♦-linearizable. The latter requires that every finite ♦-linearizable history has an extension that is also ♦-linearizable.

**Theorem 3.** ♦-linearizability is nonblocking and satisfies locality.

Now, let us show an interesting result that highlights the relationship between commit-real-time- and real-time linearizability. Consider the histories from Figure 3. In both histories, X.push(5) → ok pushes a value onto the stack X whereas X.pop → ⊥ intends to remove a value from X, but aborts. Clearly, the order of execution of these two operations does not matter, both histories are commit-real-time and real-time linearizable. It would not be so, if X.pop had not have aborted. However, an aborted operation does not change the state of the object. As the special value ⊥ carries no information on the current state of the object or why the operation failed, we can prove the following result.

**Theorem 4.** Commit-real-time linearizability is equivalent to real-time linearizability.

Note that an aborted operation is substantially different from an aborted transaction as one can witness results of a partial transaction execution (e.g., a return value of a TM operation o executed prior to the transaction’s abort). Hence, commit-real-time opacity and real-time opacity are not equivalent.

5.3. Relation between ♦-linearizability and ♦-opacity

In order to show the relationship between ♦-linearizability and ♦-opacity, we introduce a gateway shared object, whose sole purpose is to simply execute
a transactional program on some TM object.

A gateway shared object \( G \) is an abortable shared object implemented using a TM object \( M \), as shown in Algorithm 1. The interface of \( G \) consists of a single operation \( G.\text{perform}(\text{prog}_k) \rightarrow \{v_k, \bot\} \) (for process \( p_i \)). The argument \( \text{prog}_k \) is a sequence of steps that either perform some local computation, operate on t-objects managed by \( M \) or control the flow of the transaction. Upon execution of \( \text{perform}(\text{prog}_k) \), a process \( p_i \) performs steps of \( \text{prog}_k \) one by one starting from the first one. The effects of step execution are recorded in a special variable called \( \text{context} \) (line 2). It stores values of temporary variables, state of the program execution (which step is next, etc.) and any other information required to execute the program (as defined by \( \text{prog}_k \) itself). All the operations on t-objects, which form a part of some step, are executed by \( p_i \) through object \( M \) (we assume that each step contains at most one such operation). More precisely, for every call of operation \( \text{op} \) on a t-object \( x \), \( p_i \) invokes \( M.\text{texec}(T_k, x.\text{op}) \) (line 7). The first execution of the \( \text{texec} \) operation marks the start of the transaction \( T_k \). If \( \text{texec} \) returns \( A_k \), the execution of the rest of \( \text{prog}_k \) is canceled, \( T_k \) is aborted and \( \bot \) becomes the return value for the \( G.\text{perform}(\text{prog}_k) \) call (line 8). Otherwise, the step execution is simulated with the use of return value \( v \) obtained from \( M.\text{texec}(T_k, x.\text{op}) \) (line 9). It means that all the other instructions of the step (e.g., any assignments to temporary variables) are executed normally, whereas for any operation \( \text{op} \) on \( x \), the return value of \( \text{op} \) is substituted by \( v \). Transaction \( T_k \) is aborted on demand, if the execution of \( \text{prog}_k \) produces a step containing an \textit{abort} command (line 12). The execution of the last operation specified within \( \text{prog}_k \) is followed by the invocation of \( M.\text{tryC}(T_k) \) (line 14), which attempts to commit the current transaction. If this operation succeeds (\( M.\text{tryC}(T_k) = C_k \)), \( G.\text{perform}(\text{prog}_k) \) returns \( \text{context} \) (line 16); otherwise, \( \bot \) is returned (line 15).

The sequential specification \((Q, q_0, INV, RES, \delta)\) of \( G \) is given as follows:

- \( Q \) is a set of all possible combined states of all t-objects in \( M \),
- \( q_0 \) is the combined initial state of all t-objects in \( M \),
Algorithm 1 Implementation of a gateway shared object $G$ for process $p_i$

1: function $\text{perform}(\text{text } \text{prog}_k)$
2:    $context \leftarrow \text{null}$
3: while true do
4:    $step \leftarrow \text{fetchNextStep}($\text{prog}_k$, context)$
5:    if $step$ is local computation then $context \leftarrow \text{execute}(step)$
6:    if $step$ involves an operation $op$ on a t-object $x$ then
7:        $v \leftarrow M.\text{texec}(T_k, x.\text{op})$
8:        if $v = A_k$ then return $\perp$
9:    $context \leftarrow \text{simulateStepExecution}(step, v)$
10:   if $step$ is an abort command then
11:       $M.\text{tryA}(T_k)$
12:       return $\perp$
13:   if $step$ is null (there are no more steps in $\text{prog}_k$) then
14:       $v \leftarrow M.\text{tryC}(T_k)$
15:       if $v = A_k$ then return $\perp$
16:   return $context$

- $\text{INV}$ is the set containing all operations $\text{perform}(\text{prog}_k)$, where $\text{prog}_k$ is any (correct) program that eventually terminates,

- $\text{RES}$ is the set of all possible results obtained by executing any $\text{prog}_k$,

- $\delta$ is a transition relation such that $\delta \subseteq Q \times \text{INV} \times \text{RES} \times Q$.

$(q, op, res, q') \in \delta$ if either $q = q'$, $op = \text{perform}(\text{prog}_k)$ and $res = \perp$, or $op = \text{perform}(\text{prog}_k)$ and $res = v_k$ ($v_k \neq \perp$), which is obtained as a result of execution of $\text{prog}_k$ on t-objects managed by $M$ in state $q$. Then, $q'$ is the state of (possibly) modified t-objects in $M$.

We allow $context$ to contain any data obtained from the execution of $\text{prog}_k$, which are consistent with the desired behaviour of the application (defined by the set of acceptable programs $\mathcal{P}_G$). In particular, it may reflect the whole interaction between $G$ and $M$ during the execution of transaction $T_k$. However, if for any reason $T_k$ aborts, no results of partial execution of $\text{prog}_k$ (and thus $T_k$), are returned to $p_i$. This way the sequential specification of $G$ is never compromised, in case $T_k$ does not commit.

30
Theorem 5. Let $M$ be a TM object and let $G$ be a gateway shared object of $M$. If $M$ is $\Diamond$-opaque then $G$ is $\Diamond$-linearizable.

This result allows us to reason about the properties of a TM system, when transactions are invisible to processes using the system. This result is particularly important when we consider a system built with several TM implementations, each used by clients as a simple (non-transactional) shared object.

Note that an implication opposite to the one from Theorem 5 does not necessarily hold. The reason for this stems from the results presented in [34] and in [4]. In the former, the authors prove that, when none of the intermediate results of a transaction execution are available after the transaction aborts, TMS1 is necessary and sufficient for observational refinement. It means that a programmer cannot distinguish between the execution of a transaction on a TM object that satisfies TMS1 and on an abstract TM that would execute all transactions sequentially. In the latter paper, the authors show that TMS1 is incomparable with opacity. We believe that a notion of a gateway shared object can be considered as an alternative to observational refinement for comparing guarantees offered by TM systems as seen by external clients.

Corollary 3. Let $M$ be a TM object and let $G$ be a gateway shared object of $M$. If $M$ is commit-real-time opaque, then $G$ is real-time linearizable.

This result shows that a TM system, which allows transactions to read stale data, is still real-time linearizable if it eventually aborts every such transaction.

6. Deferred Update Replication

In this section, we focus on the Deferred Update Replication (DUR) protocol, a popular replication scheme aimed for databases and DTM systems. We use DUR to showcase the usefulness of new properties: update-real-time opacity and update-real-time linearizability. We follow the description of DUR in its simplest form, as in [35]. We then formally prove DUR’s guarantees.
Algorithm 2 Deferred Update Replication for process $p_i$

1: integer $LC ← 0$
2: set $Log ← ∅$
3: function $getObject$($txDescriptor t$, $objectId oid$)
4: if $(oid, obj) ∈ t.updates$ then
5:   value ← obj
6: else
7:   value ← retrieve object $oid$
8: return value
9: function $certify$($integer start$, set readset)
10: lock {
11:   $L ← \{t ∈ Log : t.end > start\}$
12: for all $t ∈ L$ do
13:   $writeset ← \{oid : ∃(oid, obj) ∈ t.updates\}$
14:   if $readset ∩ writeset \neq ∅$ then return failure
15: return success

Thread $q$ on request $r$ from client $c$ (executed on one replica)
13: enum $outcome_q ← \text{failure}$  // type: enum { success, failure }
14: response $res_q ← \text{null}$
15: $txDescriptor t ← \text{null}$  // type: record (id, start, end, readset, updates)
16: upon init
17: wait until $LC ≥ r.clock$
18: TRANSACTION()
19: return $(r.id, LC, res_q)$ to client $c$
20: procedure TRANSACTION
21: $t ← \{\text{a new unique id, 0, 0, 0, 0,} \emptyset\}$
22: lock {$t.start ← LC$}
23: $res_q ← \text{execute } r.prog \text{ with } r.args$
24: if $outcome_q = \text{failure}$ then TRANSACTIOn()
25: function $read$($objectId oid$)
26: $t.readset ← t.readset ∪ \{oid\}$
27: lock { if $\text{certify}(t.start, \{oid\}) = \text{failure}$ then $\text{RETRY()}$
28: else return $getObject(t, oid)$
29: procedure $write$($objectId oid$, object $obj$)
30: $t.updates ← \{(oid', obj') ∈ t.updates : oid' \neq oid\} ∪ \{(oid, obj)\}$
31: procedure $\text{RETRY}$
32: stop executing $r.prog$
33: procedure $\text{ROLLBACK}$
34: stop executing $r.prog$
35: $outcome_q ← \text{success}$
procedure COMMIT
stop executing r.prog

if t.updates = ∅ then
outcome_q = success
return

if CERTIFY(t.start, t.readset) = failure then return

TO-BROADCAST t // blocking

The main thread of DUR (executed on all replicas)

upon TO-DELIVER (txDescriptor t)

if CERTIFY(t.start, t.readset) = success then

lock { t.end ← LC
Log ← Log ∪ {t}
apply t.updates
LC ← LC + 1 }

if transaction with t.id executed locally by thread q then outcome_q ← success

6.1. Specification

DUR typically assumes full replication of shared data items (or objects) on which transactions operate. In our pseudocode, presented in Algorithm 2, each shared object is identified by a unique value of a special type objectId. For simplicity, we assume that each shared object can only be read or written to.

Transactions are submitted to the system by clients. Each client request consists of three elements: prog, which specifies the operations to be executed within a transaction and calls the COMMIT procedure at some point, args, which holds the arguments needed for the program execution and clock, a special integer value necessary for ensuring that all earlier requests issued by the client are serialized before the most recent client’s request.

Each process maintains two global variables. The first one, LC, represents the logical clock which is incremented every time a process applies updates of a transaction (line 48). LC allows the process to track whether its state is recent enough to execute the client’s request (line 17). Additionally, LC is used to mark the start and the end of the transaction execution (lines 22 and 45). The transaction’s start and end timestamps, stored in the transaction descriptor
(line 13), allow us to reason about the precedence order between transactions. Let $H$ be some execution of DUR, $T_i$ and $T_j$ some transactions in $H$ and $t_i$ and $t_j$ be their transaction descriptors, respectively. Then, $t_i.end \leq t_j.start$ holds only if $T_i \prec_H T_j$. The second variable, $Log$, is a set used to store the transaction descriptors of committed transactions. Maintaining this set is necessary to perform transaction certification.

DUR detects conflicts among transactions by checking whether a transaction $T_k$ that is being certified read any stale data (shared objects modified by a concurrent but already committed transaction). To this end, DUR traces accesses to shared objects independently for each transaction. The identifiers of objects that were read and the modified objects themselves are stored in private, per transaction, memory spaces. On every read, an object’s identifier is added to the readset (line 26). Similarly, on every write a pair of the object’s identifier and the corresponding object is recorded in the updates set (line 30).

Then, the certify function compares the given readset against the updates of all committed transactions in $Log$ concurrent with $T_k$. If it finds any non-empty intersection of the sets, $T_k$ is aborted and forced to retry; otherwise, it can proceed. Note that a check against conflicts is performed upon every read operation (line 27). This way $T_k$ is guaranteed to always read from a consistent snapshot.

When a transaction’s code calls COMMIT (line 30), the committing phase is initiated. If $T_k$ is a read-only transaction ($T_k$ did not modify any objects), it can commit straight away without performing any further conflict checks or process synchronization (line 38). All possible conflicts would have been detected earlier, upon read operations (line 27). If $T_k$ is an updating transactions, it is first certified locally (line 41). This step is not mandatory, but allows the process to detect conflicts earlier, and thus sometimes avoids costly network communication. Next, the transaction’s descriptor containing readset and updates is broadcast to all processes using TO-BROADCAST (line 42). The message is

\[ \text{Moreover, if both } T_i \text{ and } T_j \text{ are committed updating transactions, } T_i \prec_H T_j \text{ and } t_i.end > t_j.start, \text{ then } T_i \text{ and } T_j \text{ must not be in conflict (as otherwise } T_j \text{ would be aborted).} \]
delivered in the main thread, where the final certification takes place (line 44). Upon successful certification, processes apply $T_k$’s updates and commit $T_k$ (lines 45–48). Otherwise, $T_k$ is rolled back and reexecuted by the same process.

To manage the control flow of a transaction, the programmer can use two additional procedures: ROLLBACK and RETRY. ROLLBACK (line 33) stops the execution of a transaction and revokes all the changes it performed so far. RETRY (line 31) forces a transaction to rollback and restart.

For clarity, we make several simplifications. Firstly, we use a single global lock to synchronize operations on $LC$ (lines 22, 45, 48), $Log$ (lines 8 and 46) and the accesses to transactional objects (lines 5 and 47). Secondly, we allow $Log$ to grow indefinitely. $Log$ can easily be kept small by garbage collecting information about the already committed transactions that ended before the oldest live transaction started its execution in the system. Thirdly, we use the same certification procedure for both the certification test performed upon every read operation (line 27) and the certification test that happens after a transaction descriptor is delivered to the main thread (line 44). In practice, doing so would be very inefficient, because upon every read operation we check for the conflicts against all the concurrent transactions (line 8), thus performing much of the same work again and again. However, these repeated actions can be easily avoided by associating the accessed shared objects with a version number equal to the value of $LC$ at the time the objects were most recently modified.

6.2. Correctness

Now we formally prove DUR’s guarantees. We use DUR as specified in Algorithm 2 but we consider only t-histories of DUR, i.e., histories limited to events that are related to operations on t-objects and controlling the flow of transactions such as commit and abort events. In this sense, we treat the implementation of DUR as some TM object $M$, and reason about t-histories $H|M$, similarly as we do in Section 5.3.

**Theorem 6.** Deferred Update Replication does not satisfy write-real-time opacity.
**Corollary 4.** *Deferred Update Replication does not satisfy real-time opacity.*

**Theorem 7.** *Deferred Update Replication satisfies update-real-time opacity.*

Now we can easily show how DUR can be used to build an update-real-time linearizable system.

**Corollary 5.** *Let $G$ be a gateway shared object implemented using Deferred Update Replication. Then, $G$ satisfies update-real-time linearizability.*

### 7. Conclusions

In this paper we introduced $\Diamond$-opacity and $\Diamond$-linearizability, two families of safety properties that can be used to formalize the behaviour of a wide range of strongly consistent replicated services. The new properties relax the real-time order requirement on request execution in opacity and linearizability for, e.g., read-only or aborted requests. We precisely define the relationship between the properties of the two families. In particular, we establish a formal relationship between opacity and linearizability in their original definitions which, to our best knowledge, has not been done before.

We used Deferred Update Replication, a well known optimistic concurrency control scheme, to showcase the usefulness of $\Diamond$-opacity and $\Diamond$-linearizability. We formally proved that DUR does not satisfy the original definition of opacity but satisfies update-real-time opacity, a weaker property introduced in this paper as a member of the $\Diamond$-opacity family. We also showed that a system which can be modelled as a shared object implemented using DUR satisfies update-real-time linearizability. We argue that both properties, although weaker than their original counterparts, still provide useful guarantees for the programmers. (e.g., transactions never read from an inconsistent state, updating transactions respect real-time order).

Our approach to relaxing the real-time order requirement can be further explored in the context of other safety properties. We believe that it may be of specific importance for eventually consistent systems.
References


[16] L. Lamport, How to make a multiprocessor computer that correctly executes multiprocess programs, IEEE Transactions on Computers C-28 (9).


Appendix

This supplemental material is an appendix of the paper: *Relaxing Real-time Order in Opacity and Linearizability*, containing the proofs of lemmas and theorems. See the manuscript for the definition of terms and symbols that appear in the proofs below.

Appendix A. ♦-opacity is a safety property

Now we prove that ♦-opacity is a safety property [21] [22]. The use of order relations other than real-time does not influence opacity’s prefix-closeness and limit-closeness. Therefore, the proof of the following theorem is identical as for the original definition of opacity [3].

**Theorem 1.** ♦-opacity is a safety property.

*Proof.* In order to prove that ♦-opacity is a safety property, it is necessary to show that it is non-empty, prefix-closed and limit-closed.

Trivially an empty t-history \( H_\perp = \langle \rangle \) is ♦-opaque. Therefore, ♦-opacity is non-empty.

By the definition of ♦-opacity, if a t-history \( H \) is ♦-opaque, then every \( H_k \), a finite prefix of \( H \), is final-state ♦-opaque. Since for every \( H_k \), all its prefixes \( (H_i, i < k) \) are final-state ♦-opaque, \( H_k \) is also ♦-opaque. Therefore, every finite prefix of \( H \) is ♦-opaque, which means that ♦-opacity is prefix-closed.

Now, let us consider an infinite sequence \( H_0, H_1, H_2, \ldots \) of finite t-histories, such that each \( H_k \) is a prefix of \( H_{k+1} \), and \( H_k \) is ♦-opaque. Let \( H \) be a t-history which is the limit of this sequence. Since each \( H_k \) is finite and ♦-opaque, \( H_k \) is final-state ♦-opaque. Therefore by definition of ♦-opacity \( H \) is ♦-opaque. Thus, ♦-opacity is limit-closed.

To conclude, we have shown that ♦-opacity is non-empty, prefix-closed and limit-closed. Therefore it is a safety property. \( \square \)

**Corollary 1.** A system (modelled as a TM object) is ♦-opaque if, and only if every finite history it produces is final-state ♦-opaque.

41
Proof. The proof follows directly from Theorem 1.

Appendix B. ♦-linearizability is a safety property

Theorem 2. ♦-linearizability is a safety property.

Proof. The proof is analogous to the proof of Theorem 1.

Corollary 2. A system (modelled as a set of shared objects) is ♦-linearizable, if and only if every finite history it produces is final-state ♦-linearizable.

Proof. The proof follows directly from Theorem 2.

Appendix C. ♦-linearizability is nonblocking and satisfies locality

The proof of Theorem 3 is inspired by the proof in [14].

Lemma 1. Let $X \in \mathcal{X}$ be a shared object with a sequential specification $(Q, q_0, INV, RES, δ)$. Let $op \in INV$ be a total operation on $X$. If a finite ♦-linearizable history $H$ contains the invocation event $X.inv_l(op)$ of some pending operation execution $o_k$ (for some process $p_l$), then there exists a response $v \in RES$, such that the history $H'$ obtained by appending the response event $X.resp_l(v)$ to $H$ is ♦-linearizable.

Proof. Since $H$ is a finite ♦-linearizable history, $H$ is also final-state ♦-linearizable. Let $\bar{H}$ be some completion of $H$ such that there exists $S$, a sequential history equivalent to $\bar{H}$ such that $S$ is legal and $S$ respects the ♦-order of $\bar{H}$. We have two cases to consider. If $S$ contains $X.inv_l(op)$, then $S$ also contains $X.resp_l(v)$, as $S$ is complete. Therefore $S$ is equivalent to $\bar{H}$ and $\bar{H}$ is ♦-linearizable.

Now consider a case when $S$ does not contain $X.inv_l(op)$. Because $X.inv_l(op)$ is an invocation of a total operation, a response to this operation is defined for every state of $X$. Therefore, there exists a response $v \in RES$ such that $S' = S \cdot (X.inv_l(op), X.resp_l(v))$ is legal. Let $\bar{H}'$ be a completion of $H$ constructed in the same way as $\bar{H}$ but without removing $X.inv_l(op)$ and by appending $X.resp_l(v)$ at the end of the history. Then $S'$ is equivalent to $\bar{H}'$. 

42
Now we have to show that $S'$ satisfies the $\Diamond$-order of $\bar{H}'$. We give the proof by a contradiction. Assume that $S'$ does not satisfy the $\Diamond$-order of $\bar{H}'$. Then there exist two operation executions $o_i$ and $o_j$, $i \neq j$, such that $o_i \prec_{\bar{H}'}^\Diamond o_j$ and $o_i \not\prec_{S'} o_j$. Since $o_i \prec_{\bar{H}'} o_j$, we know that $\prec_{\bar{H}'}^\Diamond \neq \prec_{\bar{H}'}^\Box$ (\(\prec_{\bar{H}'}^\Box = \emptyset\)), and that $o_i \prec_{\bar{H}'} o_j$ and $o_i \not\prec_{S'} o_j$ (every order relation other than arbitrary order assumes real-time precedence). Note that, since $\prec_{\bar{H}'}^\Diamond \subseteq \prec_{S'}^\Diamond$ and $S' = S \cdot (X.inv_l(op), X.resp_l(v))$, then $\prec_{\bar{H}'}^\Diamond \subseteq \prec_{S'}^\Diamond$. Therefore, if $o_i \prec_{\bar{H}'} o_j$ and $o_i, o_j \in \bar{H}$, then $o_i \prec_{S'}^\Diamond o_j$. It means that either $o_i = o_k$ or $o_j = o_k$. In the former case, $o_k \prec_{\bar{H}'} o_j$. By construction of $\bar{H}'$ we know also that the response event of $o_k$ is the last event in $\bar{H}'$ and therefore no operation executions can succeed $o_k$ in $\bar{H}'$, a contradiction. In the latter case, we know that $o_i \not\prec_{S'} o_k$. Therefore, $o_k \prec_{S'} o_i$ because $S'$ is totally ordered. But it is impossible, since $o_k$ is the last operation execution in $S'$, a contradiction. Therefore the assumption was false and $S'$ respects the $\Diamond$-order of $\bar{H}'$, what concludes the proof.

\begin{lemma}
A history $H$ is $\Diamond$-linearizable if and only if for every shared object $X \in \mathcal{X}$, $H|X$ is $\Diamond$-linearizable.
\end{lemma}

\begin{proof}
First we show that if $H$ is $\Diamond$-linearizable then, for every shared object $X \in \mathcal{X}$, $H|X$ is $\Diamond$-linearizable. From the assumption we know that $H$ is $\Diamond$-linearizable and thus every finite prefix $H'$ of $H$ is final-state $\Diamond$-linearizable. Let $\bar{H}'$ be a completion of $H'$ such that there exists a legal sequential history $S'$ equivalent to $\bar{H}'$, which respects the $\Diamond$-order of $\bar{H}'$. Since $S'$ is legal, for each $X$, $S'|X$ is legal. Trivially, for every $X$, $\bar{H}'|X$ and $\bar{H}'|X$ is a completion of $H'|X$. Therefore, for each $S'|X$ is equivalent to $\bar{H}'|X$, $S'|X$ satisfies the $\Diamond$-order of $X$, $H'|X$ is final-state $\Diamond$-linearizable. Obviously, $H'|X$ is a prefix of $H|X$. Since, every finite prefix of $H|X$ is final-state $\Diamond$-linearizable, by definition, $H|X$ is $\Diamond$-linearizable.

Now we show that if $H$ is a history such that for every shared object $X \in \mathcal{X}$, $H|X$ is $\Diamond$-linearizable then $H$ is $\Diamond$-linearizable. Let $H'$ be any finite prefix of $H$. Now let us consider any $X$, and let $H'_X = H'|X$. Then, $H'_X$ is final-state $\Diamond$-linearizable. Let $\bar{H}'_X$ be a completion of $H'_X$ such that $S'_X$ is a legal sequential
history equivalent to $\bar{H}'_X$ and $\preceq^{\odot}_{S'_X}$ be an order relation in $S'_X$ which respects
the $\odot$-order of $\bar{H}'_X$ ($\preceq^{\odot}_{H'_X} \subseteq \preceq^{\odot}_{S'_X}$).

Let $\bar{H}'$ be a completion of $H'$ which contains all events of every $\bar{H}'_X$ such
that $X \in \mathcal{X}^r$, and satisfies the order $\preceq_{\bar{H}'}$, which is defined as follows. Let $e_i$ and
$e_j$ be any two events in $\bar{H}'$:

- if $e_i, e_j \in H'$ and $e_i$ precedes $e_j$ in $H'$ then $e_i$ precedes $e_j$ in $\bar{H}'$ ($e_i \preceq_{\bar{H}'} e_j$),
- if $e_i \in H'$ and $e_j \not\in H'$ then $e_i \preceq_{\bar{H}'} e_j$,
- if $e_i, e_j \not\in H'$, then the order of $e_i$ and $e_j$ in $\bar{H}'$ is arbitrary.

Let $\preceq$ be a transitive closure of $\bigcup_X \preceq^r_{S_X}$ and $\preceq^{\odot}_{\bar{H}'}$. Assuming that $\preceq$ is a
partial order, we can construct a sequential history $S'$ which is equivalent to $\bar{H}'$ and respects $\preceq$ (i.e., by performing a topological sort of $\preceq$). Then $S'$ is
legal, because for every $X$, $S'|_X = S'_X$ ($\preceq^{\odot}_{S'_X} \subseteq \preceq$ and $\preceq^{\odot}_{S'_X}$ is a total order) and
we know that $S'_X$ is legal. Also by construction, $S'$ respects the $\odot$-order of $\bar{H}'$
($\preceq^{\odot}_{\bar{H}'}, \subseteq \preceq$). Therefore $H'$ is final-state $\odot$-linearizable. We now show that $\preceq$ is indeed a partial order. We give a proof by contradiction.

Assume that the cycle of minimal length in $\preceq$ is $o_1 \preceq o_2 \preceq \ldots \preceq o_n \preceq o_1$.
Then each edge in the cycle follows either from the relation $\preceq^r_{S_X}$ for some shared
object $X$ or from the relation $\preceq^{\odot}_{\bar{H}'}$. Suppose that all operation executions in the
cycle pertain to the same object $X$. Since $\preceq^{\odot}_{S_X}$ is a total order, the cycle must contain two operation executions $o_i$ and $o_j$ such that $o_i \preceq^r_{S_X} o_j$ and $o_j \preceq^{\odot}_{\bar{H}'} o_i$.
Because $o_j \preceq^{\odot}_{\bar{H}'} o_i$, we know that $\langle \odot_{\bar{H}'} \neq \preceq^a_{\bar{H}'}, (\preceq^a_{\bar{H}'} = \emptyset)$, and $o_j \preceq^{\odot}_{\bar{H}'} o_i$ (every
order relation other than arbitrary order assumes real-time precedence). Note
that by the construction of $\bar{H}'$, for any two operation executions $o_k$ and $o_l$ on
the same object $X$, if $o_k \prec^r_{\bar{H}'}, o_l$ then $o_k \prec^r_{\bar{H}'}, o_l$, because $\bar{H}'|_X = H'_X$. In turn
$o_j \prec^r_{\bar{H}'}, o_i$ and $o_j \preceq^r_{S_X} o_i$ (because $\preceq^{\odot}_{H'_X} \subseteq \preceq^{\odot}_{S'_X}$). Since $\preceq^{\odot}_{S_X}$ is a total order,
both $o_i \preceq^r_{S_X} o_j$ and $o_j \preceq^r_{S_X} o_i$ cannot be true, a contradiction. Therefore the
cycle must contain operation executions on at least two different shared objects.

Let $o_j$ be an operation execution on $X$. Let $o_k$ be the first operation execution
after $o_j$ in the cycle, such that $o_j$ is an operation execution on $Y \neq X$. 
Also let $o_i$ be the first operation execution before $o_j$ in the cycle, such that $o_i$ is an operation execution on $Z \neq X$ (possibly $Y = Z$). Then $o_i \ll o_j \ll o_k$. Since $o_i$ and $o_j$ are on different objects, $o_i <^H_H o_j$. Analogically, $o_j <^H_H o_k$. Therefore $o_i <^H_H o_k$. If $o_i = o_k$, then $o_k <^H_H o_j$ and $o_j <^H_H o_k$, which is impossible. Therefore $o_i \neq o_k$. Since $o_i \neq o_k$, $o_i <^H_H o_k$ and $<^H_H \subseteq \ll$, there exists a shorter cycle $o_1 \ll o_2 \ll \ldots \ll o_i \ll o_k \ll \ldots \ll o_n \ll o_1$ which contradicts the assumption that the original cycle was the shortest one. Therefore, the assumption that $\ll$ is not a partial order was false, thus reassuring that $H'$ is final-state $\diamond$-linearizable.

Since, every prefix of $H$ is final-state $\diamond$-linearizable, $H$ is $\diamond$-linearizable, what concludes the proof.

**Theorem 3.** $\diamond$-linearizability is nonblocking and satisfies locality.

**Proof.** The proof follows directly from Lemma 1 and Lemma 2.

**Appendix D. Commit-real-time linearizability is equivalent to real-time linearizability**

**Lemma 3.** Let $S = \langle o_1, o_2, \ldots \rangle$ be a sequential history and $S' = \langle o_1, o_2, \ldots, o_k, o_a, o_{k+1}, \ldots \rangle$ be a sequential history constructed by inserting an aborted operation execution $o_a = X.\text{op} \rightarrow_r \bot$ to $S$ (for some process $p_r$). $S$ is legal if and only if $S'$ is legal.

**Proof.** First let us assume that $S$ is legal. By definition of legality, $S|X$ satisfies the sequential specification of $X$. Now let $o_i$ and $o_j$ be two operation executions on $X$ such that $i$ is the largest possible number lower or equal $k$ and $o_j$ is the next operation execution after $o_i$ in $S|X$ ($j$ is greater or equal $k + 1$; $o_i$ and $o_j$ may or may not exist).

Now let $q_i$ be the state of $X$ after execution of $o_i$ (or the initial state of $X$ if $o_i$ does not exist). Naturally, $q_i$ is also the state of $X$ that $o_j$ operates on. Since $o_a$ is aborted, the transition $(q_i, \text{op}, \bot, q_i)$ belongs to the sequential specification of $X$ (by definition of aborted operation). Therefore, adding $o_a$ to
$S|X$ in between $o_i$ and $o_j$, and thus creating $S'|X$, does not break the sequential specification of $X$. In turn $S'$ is also legal.

Now let us assume that $S'$ is legal. By the definition of legality, $S'|X$ satisfies the sequential specification of $X$. Now let $o_i$ be an operation execution which directly precedes $o_a$ in $S'|X$ and let $o_j$ be an operation execution which directly succeeds $o_a$ in $S'|X$ ($o_i$ and $o_j$ may or may not exist). Additionally, let $q_i$ be the state of $X$ after the execution of $o_i$ (or the initial state of $X$ if $o_i$ does not exist). Then the transition of $o_a$ is equal $(q_i, op, ⊥, q_i)$ (by definition of aborted operation). It means that the state that $o_j$ operates on is equal to $q_i$.

Therefore, removing $o_a$ from $S'|X$ (and thus creating $S|X$) still satisfies the sequential specification of $X$. In turn, $S$ is legal.

Since we proved implications in both directions, $S$ is legal if and only if $S'$ is legal.

**Theorem 4.** Commit-real-time linearizability is equivalent to real-time linearizability.

**Proof.** In order to prove that commit-real-time linearizability is equivalent to real-time linearizability we have to show that every commit-real-time linearizable history is real-time linearizable and vice versa.

Actually, it suffices to show that every final-state commit-real-time linearizable history is final-state real-time linearizable and vice versa. To show why, let us assume that the above is true. Let us consider a commit-real-time linearizable history $H$. Every finite prefix of $H$ is final-state commit-real-time linearizable, and thus also final-state real-time linearizable. Since, every finite prefix of $H$ is final-state real-time linearizable, $H$ is real-time linearizable. The same applies to the other direction. Therefore, it remains to show that indeed every final-state commit-real-time linearizable history is final-state real-time linearizable and vice versa.

Since, $\prec_H \subseteq \prec_H$, by the definition of final-state $\bowtie$-linearizability, every final-state real-time linearizable history is trivially also final-state commit-real-time linearizable. Now, we proceed to prove that every commit-real-time linearizable
history is also real-time linearizable.

Let \( H \) be any final-state commit-real-time linearizable history. We now show how to construct a sequential history which is equivalent to some completion of \( H \), such that it is legal and respects the real-time order of that completion.

**Part 1. Construction of the sequential history.**

Since \( H \) is final-state commit-real-time linearizable, we know that there exists a sequential history \( S \) equivalent to some completion of \( H \) called \( \bar{H} \), such that \( S \) is legal and \( S \) respects the commit-real-time order of \( \bar{H} \).

Let us denote by \( DSG(H) \) a directed graph which represents partial order induced by operation execution precedence in \( H \). Operation executions constitute the vertices of the graph, which are connected by a directed arch if and only if the operation execution represented by the first vertex precedes (in real-time) the operation execution represented by the second vertex. Let \( o_i \) and \( o_j \) be two operation executions in \( H \) such that \( o_i <_H o_j \). Then, naturally, it is impossible that \( o_j <_H o_i \) (\( <_H \) is asymmetric). The precedence relation (the precedence in real-time) is transitive and asymmetric, therefore, we know that \( DSG(H) \) is acyclic.

Let us construct a history \( S' \) by stripping all aborted operation executions from \( S \). This way \( S' \) is a sequential history consisting of only committed operation executions. Now let us construct a directed graph \( G' \) by extending the graph \( G = DSG(\bar{H}) \) in the following way. For any two operation executions \( o_i \) and \( o_j \) in \( S' \) such that \( o_i <_{S'} o_j \), we add an arch from \( o_i \) to \( o_j \) to \( G' \).

**Claim 1.** \( G' \) is acyclic.

**Proof.** Because \( G \) is acyclic we know that there are no two vertices \( u \) and \( v \) in \( G \), such that \( u \) is reachable from \( v \) and \( v \) is reachable from \( u \) (there exists a path from \( u \) to \( v \) and from \( v \) to \( u \)). Let us call this the no mutual reachability invariant. The condition is necessary and sufficient for a graph to be acyclic. We have to show that \( G' \) also exhibits this property.

Because \( S \) respects the commit-real-time order of \( \bar{H} \), we know that for any
two committed operation executions \(o_i\) and \(o_j\) in \(S\) such that \(o_i < S o_j\), either \(o_i < H o_j\), or \(o_i\) and \(o_j\) are concurrent in \(H\). Therefore, \(o_j \not< H o_i\).

\(S'\) contains the same committed operation executions as \(S\). Thus the above holds also for operation executions in \(S'\). Let as assume that \(S' = \langle o_1, o_2, ... \rangle\). Then, for any two operation executions \(o_i\) and \(o_j\) in \(S'\), such that \(i < j\), \(o_i\) is not reachable from \(o_j\) in \(G\). We call this the path invariant.

Let us consider the one-by-one insertion of arches between vertices induced by \(S'\). We denote by \(G_k\) the modified graph \(G\) after \(k\) insertions. We start with \(G_0 = G\), and assume that every \(G_k\) satisfies both, the no mutual reachability invariant, and the path invariant. The graph \(G'\) can be expressed as \(G^n\), where \(n\) is the total number of arches induced by \(S'\). Let us take an arbitrary arch from \(o_i\) to \(o_j\), where both \(o_i\) and \(o_j\) are in \(S'\), and \(o_i < S o_j\) (\(i < j\)). We insert it into \(G^k\). Now we show that the resulting \(G^{k+1}\) satisfies both invariants.

By the path invariant we know that \(o_i\) is not reachable from \(o_j\) in \(G^k\). When an arch from \(o_i\) to \(o_j\) is inserted in \(G^k\), then \(o_j\) becomes reachable from \(o_i\) in \(G^{k+1}\) (if it was not in \(G^k\)). However, the opposite is not true – \(o_i\) is still not reachable from \(o_j\). Adding this arch also does not change anything in mutual reachability of other operation executions as we now show by contradiction. Assume \(o_p\) and \(o_q\) are not mutually reachable in \(G^k\) (either \(o_q\) is not reachable from \(o_p\), or \(o_p\) is not reachable from \(o_q\)), but the new arch from \(o_i\) to \(o_j\) changes it in \(G^{k+1}\). Without loss of generality let us assume that \(o_q\) is reachable from \(o_p\) in \(G^k\), but that \(o_p\) becomes reachable from \(o_q\) only in \(G^{k+1}\) due to the insertion of a new arch. It means that there exists a directed path from \(o_q\) to \(o_p\) in \(G^{k+1}\) that includes the new arch from \(o_i\) to \(o_j\). For that to be possible, \(o_i\) has to be reachable from \(o_q\) and \(o_p\) has to be reachable from \(o_j\). This would mean that also \(o_q\) is reachable from \(o_j\) (\(o_q\) is reachable from \(o_p\)), and in consequence also \(o_i\) is reachable from \(o_j\). But we know that \(o_i\) is not reachable from \(o_j\) in \(G^{k+1}\), a contradiction. Therefore, the no mutual reachability invariant holds.

Now let us assume that \(o_p\) and \(o_q\) belong to \(S'\), and that \(p < q\). By the path invariant, \(o_p\) is not reachable from \(o_q\) in \(G^k\). By the same reasoning as above, the insertion of an arch from \(o_i\) to \(o_j\) cannot make \(o_p\) reachable from \(o_q\).
Therefore, also the path invariant holds.

Since, every graph $G^k$ satisfies the no mutual reachability invariant, $G'$ is acyclic.

Since $G'$ is acyclic and includes an arch for every pair of preceding operation executions in $\tilde{H}$, a topological sort of $G'$ can only yield a serialization which maintains this precedence. Let $S_{G'}$ be a history obtained by performing a topological sort on $G'$. Then $S_{G'}$ is a sequential history that respects the real-time order of $\tilde{H}$. Additionally, since $G'$ contains all the operation executions of $\tilde{H}$, so does $S_{G'}$. Thus, $S_{G'}$ is equivalent to $\tilde{H}$.

**Part 2.** Proving the legality of $S_{G'}$.

We now show that the constructed history $S_{G'}$ is legal. We do so in three steps. First, we consider a history $S''$, which is constructed by removing from $S_{G'}$ all the aborted operation executions, and show that it is equivalent to $S'$. Then, we use Lemma 3 to prove that $S'$ is legal. In consequence $S''$ is also legal. Finally, we use Lemma 3 again to prove that $S_{G'}$ is legal.

**Claim 2.** $S''$ is equal to $S'$.

*Proof.* First, we show that $S'$ and $S''$ consist of the same operation executions. $S'$ is constructed by removing all aborted operation executions from $S$. Since $S$ is equivalent to $\tilde{H}$ (a completion of $H$), $S'$ contains all committed operation executions of $\tilde{H}$. On the other hand, $S''$ is constructed by removing all aborted operation executions from $S_{G'}$, a history obtained by a topological sort of graph $G'$. $G'$ is created by adding some arches to graph $G = DSG(\tilde{H})$. Therefore, both $G$ and $G'$ consist of all operation executions of $\tilde{H}$. It means that $S''$, similarly as $S'$, consists of all committed operation executions of $\tilde{H}$.

Now we have to show that the order of operation executions in $S'$ and $S''$ is the same. Since both histories are sequential, there exist total orders $<_S$ and $<_S'$. We therefore have to prove that for any two (committed) operation executions $o_i$ and $o_j$, $o_i <_S o_j$ if $o_i <_S' o_j$. 49
We give a proof by a contradiction. Assume that \( o_i <_{S'} o_j \) and \( o_i \notin_{S'} o_j \). Since \( G' \) contains an arch from \( o_i \) to \( o_j \), the result of a topological sort on \( G' \) can only yield a serialization in which \( o_i \) precedes \( o_j \). Therefore, \( o_i <_{S'G'} o_j \). \( S'' \) is constructed by removing aborted operation executions from \( S'G' \), therefore it contains the same committed operation executions as \( S'G' \) arranged in the same order. As \( S' \) contains only committed operation executions, both \( o_i \) and \( o_j \) are committed. Therefore, \( o_i <_{S''G'} o_j \), a contradiction.

We have shown that \( S' \) and \( S'' \) consist of the same operation executions arranged in the same order. Therefore, \( S' \) and \( S'' \) are equal.

From the assumptions we know that \( S \) is legal. Since \( S' \) is constructed by removing all aborted operations executions from \( S \), then \( S' \) is legal as well (by Lemma 3). In consequence, \( S'' \) is also legal (from Claim 2). Since \( S'' \) is legal and \( S'' \) contains all the committed operations executions of \( S'G' \) and none of its aborted operation executions, then also \( S'G' \) is legal (by Lemma 3).

Since \( S'G' \) is a sequential history equivalent to \( \tilde{H} \) (a completion of \( H \)), \( S'G' \) is legal and \( S'G' \) respects the real-time order of \( \tilde{H} \). \( H \) is final-state real-time linearizable.

Appendix E. The relation between \( \Diamond \)-linearizability and \( \Diamond \)-opacity

**Proposition 1.** Let \( H \) be some implementation history and let \( G \) be a gateway shared object of a TM object \( M \). If an operation execution \( G.\text{perform}(\text{prog}_k) \to_i v_k \) is completed (for some process \( p_i \) and transaction \( T_k \)) in \( H|G \), then transaction \( T_k \) is t-completed in \( H|M \).

**Proof.** The execution of \( \text{perform}(\text{prog}_k) \) ends through calling the return statement in lines 8, 12, 15, or 16 of Algorithm 1. We now consider each case independently. In the first case, \( T_k \) is forcefully aborted during execution of operation \( x.op \) (line 7). In the second case, \( T_k \) is aborted on demand as the result of execution of \( M.\text{tryA}(T_k) \) in line 11. In the third and forth cases, \( T_k \) is either aborted or committed, depending on the result of execution of \( M.\text{tryC}(T_k) \) in line 14. Therefore, in either case \( T_k \) is completed in \( H|M \).
Proposition 2. Let \( H \) be some implementation history and let \( G \) be a gateway shared object of a TM object \( M \). If transaction \( T_k \) is commit-pending in \( H|M \), then the function \( \text{PERFORM}(\text{prog}_k) \) completed execution of all steps of \( \text{prog}_k \).

Proof. Transaction \( T_k \) becomes commit-pending after executing \( M.\text{tryC}(T_k) \) in line 14. It means that the function \( \text{FETCHNEXTSTEP} \) in line 4 returned \( \bot \), thus signaling that there are no more steps left to complete in \( \text{prog}_k \).

Proposition 3. Let \( H \) be some implementation history, \( G \) be a gateway shared object of a TM object \( M \) and \( \text{prog}_k \) be some algorithm executed by invoking an operation execution \( o_k = G.\text{perform}(\text{prog}_k) \rightarrow_i v_k \) (for some process \( p_i \)). The state of any t-object \( x \in Q \) does not change if \( o_k \) aborts.

Proof. Abort of \( o_k \) is indicated by the return value \( \bot \). Function \( \text{PERFORM} \) returns \( \bot \) in three cases:

1. when an operation on a t-object aborts (line 7),
2. when \( \text{prog}_k \) requests an abort on demand (line 11),
3. when \( \text{prog}_k \)'s request to commit the transaction \( T_k \) ends with failure (line 14).

In all three cases \( M \) aborts \( T_k \), thus revoking any changes to any t-objects performed by \( T_k \).

Proposition 4. Let \( H \) be some implementation history, \( G \) be a gateway shared object of a TM object \( M \) and \( \text{prog}_k \) be some algorithm executed by invoking an operation execution \( o_k = G.\text{perform}(\text{prog}_k) \rightarrow_i v_k \) (for some process \( p_i \)). Then, for transaction \( T_k \in H|M \) all events of \( T_k \) occur in \( H \) after the invocation event of \( o_k \) and before the response event of \( o_k \), if such an event exists in \( H \).

Proof. Trivially, if \( T_k \) has no operations, i.e., \((H|M)|T_k \) is empty, then the proposition is satisfied. Let us consider the case where \((H|M)|T_k \) is not empty. The first event of transaction \( T_k \) is the invocation of the first operation \( M.\text{texec}(T_k, x.\text{op}) \) (line 7) executed within the function \( \text{PERFORM}(\text{prog}_k) \). Trivially, this event must appear after the invocation of \( o_k \). Moreover, from Proposition 1 we know
that $T_k$ must be completed in $H|M$ before $o_k$ can return. From the definition of a transaction, we know that $T_k$ cannot execute any other operations after it completes. Therefore, all events of $T_k$ always appear after invocation of $o_k$ and before the response event of $o_k$, if such a response event exists in a given execution.

Proposition 5. Let $H$ be some implementation history and let $G$ be a gateway shared object of a TM object $M$. If there exists a (possibly pending) operation execution $o_k = G.\text{perform}(\text{prog}_k) \rightarrow_i v_k$ in $H$ for some process $p_i$ and $(H|M)|T_k$ is not empty then $T_k$ is executed by $p_i$.

Proof. A process $p_i$ that invokes the function $\text{PERFORM}(\text{prog}_k)$, executes it without any help from other processes until the function returns. Since a transaction $T_k$ can only exist through execution of $\text{prog}_k$, all operations of $T_k$ on $M$ are issued by $p_i$. □

Proposition 6. Let $H$ be some implementation history and let $G$ be a gateway shared object of a TM object $M$. If there exists a committed operation execution $o_k = G.\text{perform}(\text{prog}_k) \rightarrow_i v_k$ in $H$ for some process $p_i$, then transaction $T_k$ is committed in $H|M$.

Proof. Operation execution $o_k$ is committed if it returns a value $v_k \neq \perp$. That only happens if operation $M.\text{tryC}(T_k)$ (line 14) return $C_k$ ($T_k$ is successfully committed). □

Proposition 7. Let $H$ be some implementation history and let $G$ be a gateway shared object of a TM object $M$. If there exists a committed operation execution $o_k = G.\text{perform}(\text{prog}_k) \rightarrow_i v_k$ in $H$ for some process $p_i$ such that $o_k$ is potentially updating, then transaction $T_k$ is committed and potentially updating in $H|M$.

Proof. From Proposition 6 we know that $T_k$ has to be committed since $o_k$ is committed. Since $o_k$ is potentially updating, there exist a state for which $o_k$ is updating. It means that potentially $o_k$ may change the state of $G$. In this
case, \( T_k \) would perform some updating operations. It means that the algorithm \( \text{alg}_k \) which defines \( T_k \) may produce an execution in which \( T_k \) contains some updating operations. Therefore, \( T_k \) cannot be declared-read-only, and thus it is potentially updating in \( H|M \).

Now let us introduce some auxiliary definitions that are necessary for the following proposition and the proof of the theorem that comes after the proposition.

Let \( H \) be an implementation history and let \( G \) be a gateway shared object of \( M \). If there exists a t-sequential t-history \( S_M \) equivalent to some t-completion of \( H_M = H|M \), such that \( S_M \) respects the \( \triangleright \)-order of \( H_M \) and every transaction in \( S_M \) is legal in \( S_M \), then we can construct a history \( S_G \) of events on \( G \) in the following way.

Let us first construct a completion \( \bar{H}_G \) of the history \( H_G = H|G \) by completing every pending operation execution in \( H_G \) with an appropriate response, as follows. For every pending operation execution \( o_k \) of operation \( G.\text{perform}(\text{prog}_k) \) in \( H_G \), there are two cases are possible:

1. if \( T_k \) is aborted in \( S_M \), then the response event for \( o_k \) in \( \bar{H}_G \) is equal \( \perp \), otherwise
2. the response value of \( o_k \) in \( \bar{H}_G \) is equal to the final value of \( \text{context} \) from the execution of \( \text{alg}_k \) in \( H \) (after \( p_i \) executes \( M.\text{tryC}(T_k) \)).

The completion of \( H_G \), denoted \( \bar{H}_G \), follows the completion of \( H_M \). Naturally, \( \bar{H}_G \) contains all the operation executions which are completed in \( H_G \). Note that if \( o_k \) is completed in \( H_G \), by Proposition 1 \( T_k \) is t-completed in \( H_M \). On the other hand, if \( o_k \) is pending, then \( T_k \) may be t-completed or not. If \( T_k \) is live in \( H_M \), it is either aborted (case 1) or committed (case 2) in \( S_M \). However, \( T_k \) may also be t-completed in \( H_M \) but \( o_k \) may simply lack a response event in \( H_G \) (cases 1 and 2, depending on whether \( T_k \) is aborted or committed). In case 2 we know that \( T_k \) is either commit-pending or committed in \( H_M \). Therefore, by Proposition 2 we know that \( p_i \) completed all steps of \( \text{prog}_k \) and \( o_k \) can complete successfully and return the final value of \( \text{context} \).
Now, let us construct a sequential completed history $S_G$ equivalent to $\bar{H}_G$.

Let $S_G$ contain all operation executions of $\bar{H}_G$, but sequentially ordered according to the order of the corresponding transactions in $S_M$. An operation execution $o_k = G.\text{perform}(\text{prog}_k) \rightarrow i v_k$ in $\bar{H}_G$ corresponds to transaction $T_l$ in $S_M$, for some process $p_i$ if $k = l$.

We say that $S_G$ is induced by $S_M$.

For any transaction $T_k$ let us denote by $V_k$ the history $\text{visible}_{S_M}(T_k)$ in case $T_k$ is committed. Otherwise, let $V_k$ be equal the history $\text{visible}_{S_M}(T_k)$ with $T_k$ omitted. Since, every transaction $T_k$ in $S_M$ is legal in $S_M$, the history $\text{visible}_{S_M}(T_k)$ is t-legal and so is $V_k$ (by the definition of a t-legal history, every its prefix is also t-legal). For any t-object $x$ the subhistory $V_k|x$ satisfies the sequential specification of $x$. It means that there exists a sequence $w_{k,x} = \langle q_0, q_1, \ldots, q_l \rangle$ of states in $Q_x$, where $Q_x$ is the set of all possible states of $x$, and $q_l$ is the last state that $x$ takes in $V_k$. Let us denote by $\text{state}_{S_M}(T_k)$ the combined state that includes every last state of all t-objects in $V_k$ according to the sequences $w_{k,x}$.

Let $q_i$ be the combined state of all the t-objects in $M$ after the execution of the $i$-th transaction in $S_M$, i.e. $q_i = \text{state}_{S_M}(T_k)$, where $T_k$ is the $i$-th transaction in $S_M$. We say that the sequence $W = \langle q_0, q_1, \ldots \rangle$ is state-induced by $S_M$. It can be shown (as we demonstrate later) that every such a sequence is a witness history of the induced history $S_G$.

**Proposition 8.** Let $H$ be some implementation history and let $G$ be a gateway shared object of a TM object $M$. Let $S_G$ be a sequential history equivalent to $\bar{H}_G$, a completion of $H_G = H|M$, such that $S_G$ is induced by some history $S_M$, and let $W_G$ be its witness history state-induced by $S_M$. If there exists a committed operation execution $o_k = G.\text{perform}(\text{prog}_k) \rightarrow i v_k$ in $\bar{H}_G$ (for some process $p_i$), such that $o_k$ is updating in $S_G$ according to $W_G$, then transaction $T_k$ is committed and is updating in $H|M$.

**Proof.** From Proposition 6 we know that $T_k$ has to be committed since $o_k$ is committed. Since $o_k$ is updating in $S_G$ according to $W_G$, the states $q_{i-1}$ and
$q_i$ in $W_G$, which correspond to $o_k$ ($o_k$ is the $i$-th operation execution in $S_G$) are different. Since the states $q_{i-1}$, $q_i$ in $W_G$ are the combined states of all the t-objects in $M$ after the execution of transaction $T_k$, the state of some t-object $x$ had to be changed by $T_k$ (according to the sequence $w_{k,x}$). This can only happen if $T_k$ executed a not read-only operation in $S_M$. This operation has to be also present in $H|M$. Therefore, $T_k$ is updating in $H|M$.

\[ \square \]

**Theorem 5.** Let $M$ be a TM object and let $G$ be a gateway shared object of $M$. If $M$ is $\Diamond$-opaque then $G$ is $\Diamond$-linearizable.

**Proof.** In order to show that $G$ is $\Diamond$-linearizable, we need to prove that any finite history produced by $G$ is final-state $\Diamond$-linearizable (by Corollary 2). Therefore, we assume some finite implementation history $H$, show how to construct a sequential history $S_G$ of events on $G$ that is equivalent to $\bar{H}_G$, a completion of $H_G = H|M$. Next, we prove that $S_G$ is legal. Finally, we show that $S_G$ respects the $\Diamond$-order of $\bar{H}_G$.

**Part 1.** Construction of a sequential history $S_G$ that is equivalent to a completion of $H|M$.

Since $M$ is $\Diamond$-opaque, there exists a t-sequential t-history $S_M$ such that $S_M$ is equivalent to some t-completion of $H_M = H|M$, $S_M$ respects the $\Diamond$-order of $H_M$ and every transaction $T_k$ in $S_M$ is legal in $S_M$. Therefore, there exists a history $S_G$ induced by $S_M$.

**Part 2.** Proof of legality of $S_G$.

From assumptions we know that every transaction in $S_M$ is legal, thus also the last committed transaction $T_l$ in $S_M$ is legal. Therefore, $vis_l = visible_{S_M}(T_l)$ contains all the committed transactions in $S_M$. Moreover, $vis_l$ is t-legal. It means, that for any t-object $x$, $vis_l|x$ satisfies the sequential specification of $x$.

Let $steps_H(o_k)$, where $H$ is any implementation history and $o_k$ be a (possibly pending) operation execution of operation $G.perform(prok_k)$, be a function which returns all the events in $H$ executed within the operation execution $o_k$.
(excluding the operations on $G$ itself), which are related to the execution of $\text{prog}_k$. Therefore, $\text{steps}_H(o_k)$ contains all the steps of $\text{prog}_k$’s execution.

Let $S'_G$ be a history obtained from $S_G$ by removing all the aborted operation executions and assume $S'_G$ is a sequence of operation executions $\langle o_1 \cdot o_2 \cdot \ldots \rangle$. We construct a sequence $\alpha = \langle \text{steps}_H(o_1) \cdot \text{steps}_H(o_2) \cdot \ldots \rangle$. Now we show that $\alpha|_M = \text{vis}_t$ by a contradiction.

Assume that $\alpha|_M \neq \text{vis}_t$. It means that there exists $o_k = G.\text{perform}(\text{prog}_k) \rightarrow_i v_k$ in $H$ such that $\text{steps}_H(o_k)|_M \neq S_M|T_k$. There are two cases to consider:

1. $\exists op \in H : op \in \text{steps}_H(o_k)|_M \land op \notin S_M|T_k$. It means that $op$ was defined in $\text{prog}_k$ and was executed by $p_i$ through $G$ and then $M$ as part of transaction $T_k$ (by definition of $G$). Therefore, $op \in S_M|T_k$, a contradiction.

2. $\exists op \in H : op \in S_M|T_k \land op \notin \text{steps}_H(o_k)|_M$. It means that $op$ was executed by $p_i$ on $M$ within transaction $T_k$. The only possibility that it happened is by specifying $op$ as part of $\text{prog}_k$ (by definition of $G$).

   Therefore, $op \in \text{steps}_H(o_k)|_M$, a contradiction.

Since both cases lead to a contradiction, the initial assumption was false and $\alpha|_M = \text{vis}_t$.

Since for any t-object $x$, $\text{vis}_t|x$ satisfies the sequential specification of $x$, so does $\alpha|_M$. Note that any $\text{prog}_k$ executed through $G$ operates in isolation and may interact with other processes only through t-objects ($G$ allows only local computation besides operations on $M$). However, all operations on t-objects induced by executing $G.\text{perform}(\text{prog}_k)$ in $S'_G$ satisfy the sequential specifications of these t-objects. Therefore, the execution of algorithms in $S'_G$ transitions t-objects in $M$ from some initial state $q_0 \in Q$, through states $q_1, q_2, \ldots$ to some correct state $q' \in Q$, thus satisfying the sequential specification of $G$. Therefore $S'_G$ is legal and $W'_G = \langle q_0, q_1 \ldots \rangle$ is its witness history.

By Proposition 3, aborted operation executions removed from $S_G$ to produce $S'_G$ do not change the state of $G$. Therefore, also $S_G$ is legal. If we extend the sequence $W'_G$ with a repeated state for every aborted operation execution, we obtain $W_G$, a witness history of $S_G$.  

56
Note that $W_G$ is state-induced by $S_M$, because for each $q_i \in W_G$, $q_i$ is the combined state of all the t-objects in $M$ after the execution of the $i$-th transaction in $S_M$ (and thus also after the execution of $i$-th operation execution in $S_G$).

**Part 3.** Proof that $S_G$ respects the ◦-order of $\vec{H}_G$.

To show that $S_G$ respects the ◦-order of $\vec{H}_G$, we have to show that for any two operation executions $o_i$ and $o_j$ on $G$ ($o_i$ invokes $G.perform(prog_i)$, while $o_j$ invokes $G.perform(prog_j)$) such that $o_i <_{\vec{H}_G}^\circ o_j$, the relation $o_i <_{S_G}^\circ o_j$ holds as well. In case of update-real-time, we have to show that there exists a witness history $W_G$, such that for all $o_i$ and $o_j$ such that $o_i <_{H_M}^{u} (S_G, W_G) o_j$, the relation $o_i <_{S_G}^{u} (S_G, W_G) o_j$ holds ($W_G$ was given in the previous part of the proof).

If $<_{\vec{H}_G}^\circ = <_{\vec{H}_G}^a$ then there are no operation executions $o_i$ and $o_j$, such that $o_i <_{\vec{H}_G}^a o_j$ (by definition $<_{\vec{H}_G}^a$ is equivalent to $\emptyset$).

Now we consider $<_{\vec{H}_G}^\circ \neq <_{\vec{H}_G}^a$. Note that every order definition apart from arbitrary order assumes precedence in real-time between the two operation executions (besides making some additional requirements on the operation executions). Therefore, we now assume that $o_i <_{\vec{H}_G}^r o_j$. By definition of real-time order, it means that the response event of $o_i$ precedes the invocation event of $o_j$ in $\vec{H}_G$. By Proposition 4 all events of $T_i$ precede all events of $T_j$ in $H$. In particular, the last event of $T_i$ precedes the first event of $T_j$ in $H$. Therefore $T_i$ completed in $H_M$ before $T_j$ started in $H_M$. It means that if $<_{\vec{H}_G}^\circ = <_{\vec{H}_G}^r$ then $T_i \prec_{H_M} T_j$. In case $<_{\vec{H}_G}^\circ \neq <_{\vec{H}_G}^r$, additionally:

- if $<_{\vec{H}_G}^\circ = <_{\vec{H}_G}^c$ then both $o_i$ and $o_j$ are committed or both are executed by the same process. It means that both $T_i$ and $T_j$ are committed or both are executed by the same process (by Propositions 5 and 6). Thus $T_i \prec_{H_M} T_j$.

- if $<_{\vec{H}_G}^\circ = <_{\vec{H}_G}^w$ then both $o_i$ and $o_j$ are potentially updating and committed or both are executed by the same process. It means that both $T_i$ and $T_j$ are committed and are potentially updating or both are executed by the same process (by Propositions 5 and 7). Thus $T_i \prec_{H_M} T_j$.  

57
• if \( \prec_{H_G} = \prec_{H_G}(S_G, W_G) \) then both \( o_i \) and \( o_j \) are updating in \( S_G \) according to \( W_G \) and are committed, or both are executed by the same process. It means that both \( T_i \) and \( T_j \) are committed and updating or both are executed by the same process (by Propositions 5 and 8). Thus \( T_i \prec_{H_M} T_j \).

• if \( \prec_{H_G} = \prec_{H_G}^p \) then both \( o_i \) and \( o_j \) are executed by the same process. It means that both \( T_i \) and \( T_j \) are executed by the same process (by Proposition 5). Thus \( T_i \sim_{H_M} T_j \).

Since there is no ambiguity between naming convention of orders in \( \prec \)-opacity and \( \prec \)-linearizability, we can simply write that \( T_i \sim_{H_M} T_j \).

Since \( S_M \) respects the \( \prec \)-order of \( H_M \) then also \( T_i \sim_{S_M} T_j \). In turn, the response event of \( o_i \) precedes the invocation event of \( o_j \) in \( S_G \), because the construction of \( S_G \) requires an order of operation execution in \( S_G \) that directly follows the order of transactions in \( S_M \). Therefore, \( S_G \) respects the \( \prec \)-order of \( H_G \) (we proved the case for \( \prec_{H_G}^a \) earlier). This way we conclude the proof.

**Corollary 3.** Let \( M \) be a TM object and let \( G \) be a gateway shared object of \( M \). If \( M \) is commit-real-time opaque, then \( G \) is real-time linearizable.

**Proof.** The proof follows directly from Theorems 4 and 5.

**Appendix F. The correctness of DUR**

In this section we use the simplified notation which we have already used in earlier Figures: 1, 2, 3, 4.

**Theorem 6.** Deferred Update Replication does not satisfy write-real-time opacity.

**Proof.** We give a proof by a contradiction. Assume that DUR satisfies write-real-time opacity. It means that every history produced by DUR is write-real-time opaque. Now consider an example in Figure 5, which shows a valid execution of DUR with two transactions \( T_1 \) (executed by \( p_1 \)) and \( T_2 \) (executed by \( p_3 \)).
Transaction $T_1$ first reads the current value of a shared object $x$, next writes to it a new value and finally calls the commit procedure (line 36) which corresponds to $tryC_1$ event in the example. Since $T_1$ modified $x$, and there are no concurrent transactions with which $T_1$ may conflict (lines 38 and 41), transaction descriptor of $T_1$ is broadcast using TO-BROADCAST.

Typically a message broadcast using TO-BROADCAST (e.g., based on the Paxos algorithm [16]) can be delivered once the majority of processes receive the message. In the example, $p_1$ and $p_2$ deliver the transaction descriptor of $T_1$ (line 43) and subsequently certify it successfully and apply the updates stored in the transaction descriptor.

The message with the transaction descriptor of $T_1$ sent from $p_1$ to $p_3$ takes more time to reach its destination. Before the reception of the message (but after $p_1$ and $p_2$ commit $T_1$, i.e., apply the updates $T_1$ produced), $p_3$ executes transaction $T_2$ which performs only a read operation on $x$. Since $T_2$ did not perform any updating operations ($T_2$ is a read-only but not a declared-read-only transaction, i.e., the predicate $DRO(T_2)$ does not hold true in this execution), it can commit without inter-process synchronization (line 38).

Since both $T_1$ and $T_2$ are committed, $T_1$ precedes in real-time $T_2$ ($T_1$ ended before $T_2$ started) and $T_2$ is not declared read-only, by definition of write-real-time opacity, the history from the example is not write-real-time opaque. Therefore, the assumption was false and DUR does not satisfy write-real-time opacity.

\[ p_1 \quad x.rd_1 \rightarrow 0 \quad x.wr_1(1) \rightarrow ok \quad tryC_1 \rightarrow C_1 \\

p_2 \quad to-broadcast \\

p_3 \quad x.rd_2 \rightarrow 0 \quad tryC_2 \rightarrow C_2 \]

Figure F.5: A valid execution of DUR which is not write-real-time opaque.

**Corollary 4.** *Deferred Update Replication does not satisfy real-time opacity.*
Proof. The proof follows directly from Theorem 6 and definitions of write-real-time opacity and real-time opacity (real-time opacity is strictly stronger than write-real-time opacity).

Proposition 9. Let $H$ be a t-history of DUR. For any two updating committed transactions $T_i, T_j \in H$ and their transaction descriptors $t_i$ and $t_j$, if $T_i \prec_H T_j$ then $t_i.end < t_j.end$.

Proof. From the assumption that $T_i \prec_H T_j$, we know that $T_i$ is committed and the first event of $T_j$ appears in $H$ after the last event of $T_i$ (the commit of $T_i$). It means that $T_i$ assigned the current value of $LC$ to $t_i.end$ and incremented $LC$ afterwards (lines 15 and 18), before the first event of $T_j$. The value of $LC$ changes only in line 18 therefore it increases monotonically. Therefore, upon $T_j$’s commit, the current value of $LC$ assigned to $t_j.end$ must be greater than $t_i.end$.

Proposition 10. Let $H$ be a t-history of DUR. Let $T_i$ and $T_j$ be two transactions in $H$ executed by some process $p_l$ and let $t_i$ and $t_j$ be transaction descriptors of $T_i$ and $T_j$, respectively. If $T_i \prec_H T_j$ then $t_i.start \leq t_j.start$.

Proof. From the assumption that $T_i \prec_H T_j$ and both are executed by the same process $p_l$, we know that $T_i$ is completed and the first event of $T_j$ appears in $H$ after the last event of $T_i$. Therefore $p_l$ assigns the current value of $LC$ to $t_i.start$ before it does so for $t_j.start$ (in line 22). The value of $LC$ changes only in line 48 and thus it increases monotonically (with commits of updating transactions). Therefore $t_i.start \leq t_j.start$.

Proposition 11. Let $H$ be a t-history of DUR and $T_k$ (with a transaction descriptor $t_k$) be some transaction in $H$. If there exists a t-object $x \in Q$, such that $T_k$ performs a read operation $r = x.read \rightarrow v$ and $T_k$ executed earlier at least one write operation on $x$ before $r$, then $v$ has been assigned to $x$ by the last write operation on $x$ in $T_k$ before $r$. 

60
Proof. Let $w = x.\text{write}(\nu') \to \text{ok}$ be the last write operation on $x$ in $T_k$ before $r$. Upon execution of $w$ if there were no prior write operations on $x$ in $T_k$ then a pair $(x, \nu')$ is added to $t_k.\text{updates}$; otherwise, the current pair $(x, \nu'')$ is substituted by $(x, \nu')$ in $t_k.\text{updates}$ (line 30). Then $\nu'$ would be returned upon execution of $r$ (line 28 and then 4), unless $T_k$ aborts. If $T_k$ aborted due to a conflict with other transaction in line 27, then $r$ would not return any value. Therefore $\nu = \nu'$ and indeed $\nu'$ was assigned to $x$ by the last write operation on $x$ in $T_k$ before $r$. □

Proposition 12. Let $H$ be a $t$-history of DUR and let $r = x.\text{read} \to \nu$ be a read operation on some $t$-object $x \in Q$ performed by some transaction $T_k$ in $H$. If $T_k$ did not perform any write operations on $x$ prior to $r$ then either there exists a transaction $T_i$ that performed $x.\text{write}(\nu) \to \text{ok}$ and committed before $r$ returns, or (if there is no such transaction $T_i$) $\nu$ is equal to the initial value of $x$.

Proof. From the assumption that $T_k$ did not perform any write operations on $x$ prior to $r$, we know that the value of $x$ is retrieved from the system state in line 5. The value of $x$ is updated only when some updating transaction $T_i$ that modified $x$ commits (line 47). Before commit, $T_i$ stores the modified values of $t$-objects in the $\text{updates}$ sets of $T_i$’s transaction descriptor. The only possibility that a new value of $x$ is stored in the $\text{updates}$ set is upon write operation on $x$ (line 30). On the other hand, if the value of $x$ in the system was never updated (through line 47), the initial value of $x$ is returned in line 5. □

Proposition 13. Let $H$ be a $t$-history of DUR, $x \in Q$ be some $t$-object and $T_i$ and $T_j$ be two transactions in $H$, such that $w = x.\text{write}(\nu) \to \text{ok}$ is the last write operation on $x$ in $T_i$, $T_j$ executed a read operation $r = x.\text{read} \to \nu'$, $T_j$ did not perform any write operations on $x$ prior to $r$ and $T_i$ committed before $r$ was issued. If there does not exist a transaction $T_i'$ that executes operation $x.\text{write}(u) \to \text{ok}$ such that $\nu \neq u$ and $T_i'$ commits after $T_i$ and before $r$ was issued, then $\nu = \nu'$.

Proof. From assumption we know that $r$ reads the value of $x$ from the system
state \((T_j\) did not perform any write operations prior to \(r\), thus the \(updates\) set of \(T_j\)’s transaction descriptor does not contain modified value of \(x\), line 30).

Moreover, we know that \(w\) is the last write operation on \(x\) in \(T_i\), which later commits. Therefore the value \(v\) is stored in system state upon the commit of \(T_i\). Since, the commit of \(T_i\) happens before \(r\), and from the assumptions we know that \(x\) has not been updated by any transaction \(T'_i\) that committed after \(T_i\) and before the return of \(r\), the value of \(x\) stored in system state is equal \(v\). Therefore, \(r\) returns \(v' = v\). \(\square\)

**Theorem 7.** Deferred Update Replication satisfies update-real-time opacity.

*Proof.* In order to prove that DUR satisfies update-real-time opacity, we have to show that every finite t-history produced by DUR is final-state update-real-time opaque (by Corollary 1). In other words, we have to show that for every finite t-history \(H\) produced by DUR, there exists a t-sequential t-history \(S\) equivalent to some completion of \(H\), such that \(S\) respects the update-real-time order of \(H\) and every transaction \(T_k\) in \(S\) is legal in \(S\).

**Part 1.** Construction of a t-sequential t-history \(S\) that is equivalent to a completion of \(H\).

Let \(update : \mathbb{N} \rightarrow \mathcal{T}\) be a function that maps the values which are assigned to \(LC\) during the system execution to a committed updating transaction which set that particular value (line 48). Let \(S = (H|update(1) \cdot H|update(2) \cdot \ldots)\). This way \(S\) includes all the committed updating transactions in \(H\). We complete every live transaction \(T_k\) in \(H\) with a \(M.\text{try}A(T_k) \rightarrow_i A_k\) operation execution, or \(resp_i(A_k)\) response event, if \(T_k\) ends with a pending operation execution in \(H\) (for some process \(p_i\) which executes \(T_k\)). Now, let us add the rest of transactions from \(H\) to \(S\) in the following way. For every such a transaction \(T_k\) with a transaction descriptor \(t_k\), find a committed updating transaction \(T_l\) (with transaction descriptor \(t_l\)) in \(S\), such that \(t_k.\text{start} = t_l.\text{end}\), and insert \(H|T_k\) immediately after \(T_l\)’s operations in \(S\). If there is no such transaction \(T_l\) (\(t_k.\text{start} = 0\)), then add \(H|T_k\) to the beginning of \(S\). If there
are multiple transactions with the same value of \textit{start} timestamp, then insert them in the same place in \( S \). Their relative order is irrelevant unless they are executed by the same process. In such a case, rearrange them in \( S \) according to the order in which they were executed by the process.

**Part 2.** Proof that \( S \) respects the update-real-time order of \( H \).

Let \( T_i \) and \( T_j \) be any two transactions such that \( T_i \prec_u H T_j \) and let \( t_i \) and \( t_j \) be transaction descriptors of \( T_i \) and \( T_j \), respectively. Then, \( T_i \prec_r H T_j \) and

1. \( T_i \) and \( T_j \) are updating and committed, or
2. \( T_i \) and \( T_j \) are executed by the same process.

In case 1, by Proposition 9 we know that \( t_i.end < t_j.end \). Both \( t_i.end \) and \( t_j.end \) increased by 1 correspond to the values assigned to LC by \( T_i \) and \( T_j \) upon their commit (lines 45 and 48). Then, by the construction of \( S \), \( T_i \) must appear in \( S \) before \( T_j \). Therefore, \( T_i \prec_S T_j \). Moreover, the construction requires that for any transaction \( T_k \in H \), \( S \) includes all events of \( H|T_k \). In turn both \( T_i \) and \( T_j \) are updating and committed in \( S \). Therefore, in this case, \( T_i \prec_u H T_j \).

Now let us consider case 2. From Proposition 10 we know that \( t_i.start \leq t_j.start \). Therefore, we have further two cases to consider. If \( t_i.start = t_j.start \) (and both \( T_i \) and \( T_j \) are executed by the same process), then the construction of \( S \) explicitly requires that \( T_i \) and \( T_j \) are ordered in \( S \) according to the order in which they were executed by this process. On the other hand, if \( t_i.start < t_j.start \) then by the construction of \( S \):

1. \( T_i \) and \( T_j \) appear in \( S \) after some committed updating transactions \( T'_i \) and \( T'_j \) with transaction descriptors \( t'_i \) and \( t'_j \) such that \( t_i.start = t'_i.end \) and \( t_j.start = t'_j.end \). It means that \( t'_i.end < t'_j.end \), therefore \( T'_i \) appears in \( S \) before \( T'_j \) (by the construction of \( S \)). Moreover, between \( T'_i \) and \( T_i \) in \( S \) there is no other committed updating transaction, since, by the construction of \( S \), \( T_i \) is inserted immediately after \( T'_i \). In turn, the four transactions appear in \( S \) in the following order: \( T'_i, T_i, T'_j, T_j \). Thus \( T_i \prec_S T_j \).
2. if such \( T'_i \) does not exist \((t_i.\text{start} = 0\); there is no committed updating transaction in \( S \) before \( T_i \)), we know that \( T'_j \) has to exist since \( t'_j.\text{start} = t_j.\text{end} > t_i.\text{start} = 0 \). Then, the three transactions appear in \( S \) in the following order: \( T_i, T'_j, T_j \). Thus also \( T_i \prec_S T_j \).

This way \( S \) respects the update-real-time order of \( H \) (trivially, for any transaction \( T_k \) executed by process \( p_i \) in \( H \), \( T_k \) is executed by \( p_i \) in \( S \)).

**Part 3.** Proof that every transaction \( T_k \) in \( S \) is legal in \( S \).

We give the proof by contradiction. Assume that there exists a transaction \( T_k \) (with a transaction descriptor \( t_k \)) such that \( T_k \) is not legal in \( S \). It means that there exits \( x \in \mathbb{Q} \) such that \( \text{vis} = \text{visible}_S(T_k)\mid x \) does not satisfy the sequential specification of \( x \).

The only type of \( t \)-object considered in DUR are simple registers (see Section 3). Sequential specification of a register \( x \) is violated when a \textit{read} operation \( r = x.\text{read} \rightarrow v \) returns a value \( v \) that is different from the most recently written value to this register using the \textit{write} operation, or its initial value if there was no such operation.

Let us first assume that the last modification to \( x \) prior to \( r \) is performed by some operation \( w = x.\text{write}(v') \rightarrow \text{ok} \) such that \( v \neq v' \). If \( w \) was performed by \( T_k \), then \( v = v' \) (by Proposition 11), a contradiction. Therefore, \( v \) depends on the current state of the system. Since we assume that some modification to \( x \) has been performed prior to \( r \) in \( S \), then there exists a transaction \( T_i \) that performed operation \( w \) and committed before \( r \) was executed (by Proposition 12). Moreover, from the assumption we can stipulate that there is no other write operation on \( x \) between \( w \) and \( r \) in \( S \), \( r \) has to return \( v' \) (by Proposition 13). Therefore \( v = v' \), a contradiction.

On the other hand, if \( x \) was not modified prior to \( r \), trivially, \( v \) has to be equal to the initial value of \( x \) (by Proposition 12), thus completing the proof by contradiction. It means that the assumption that there exists a transaction \( T_k \) that is not legal in \( S \) is false. Therefore every transaction \( T_k \) is legal in \( S \). This concludes the proof that DUR guarantees update-real-time opacity. \( \Box \)
Corollary 5. Let $G$ be a gateway shared object implemented using Deferred Update Replication. Then, $G$ satisfies update-real-time linearizability.

Proof. The proof follows directly from Theorem 5 and Theorem 7. □