Understanding Inconsistency in Azure Cosmos DB with TLA+

Finn Hackett
University of British Columbia
Vancouver, Canada
fhackett@cs.ubc.ca

Joshua Rowe
Microsoft
Redmond, USA
joshua.rowe@microsoft.com

Markus Alexander Kuppe
Microsoft Research
Redmond, USA
makuppe@microsoft.com

Abstract—**Beyond implementation correctness of a distributed system, it is equally important to understand exactly what users should expect to see from that system. Even if the system itself works as designed, insufficient understanding of its user-observable semantics can cause bugs in its dependencies. By focusing a formal specification effort on precisely defining the expected user-observable behaviors of the Azure Cosmos DB service at Microsoft, we were able to write a formal specification of the database that was significantly smaller and conceptually simpler than any other specification of Cosmos DB, while representing a wider range of valid user-observable behaviors than existing more detailed specifications. Many of the additional behaviors we documented were previously poorly understood outside of the Cosmos DB development team, even informally, leading to data consistency errors in Microsoft products that depend on it. Using this specification, we were able to raise two key issues in Cosmos DB's public-facing documentation, which have since been addressed. We were also able to offer a fundamental solution to a previous high-impact outage within another Azure service that depends on Cosmos DB.**

Index Terms—**cloud computing, formal methods, model checking, documentation**

I. Introduction

Consistency guarantees for distributed databases are notoriously hard to understand. Not only can distributed systems inherently behave in unexpected and counter-intuitive ways due to internal concurrency and failures, but they can also lull their users into a false sense of functional correctness: most of the time, users of a distributed database will witness a much simpler and more consistent set of behaviors than what is actually possible. Only timeouts, fail-overs, or other rare events will expose the true set of behaviors a user might witness [1]. Testing for these scenarios is difficult at best: reproducing them reliably requires controlling complex concurrency factors, latency variations, and network behaviors. Even just producing usable documentation for developers is fundamentally challenging [2], [3], [4], and explaining these subtle consistency issues via documentation comes as an additional burden to distributed system developers and technical writers alike. Formal methods have long been applied to the design of distributed systems, including in industry [5], [6], [7], [8], but these are years-long high-effort projects that focus on implementation correctness, not explaining the system to users. Rather than focus on this difficult task, we address a simpler and more fundamental question: ignoring the implementation, what kind of behavior *should* a client be able to witness while interacting with a service?

We use TLA$^+$ to answer this simpler question for Cosmos DB, a planet-scale key-value store. In practice, Cosmos DB offers a rich interface featuring multiple query APIs, and has complex operational behaviors involving georeplication and partitioning of data. As our focus is on data consistency from a client perspective, we model only the core read and write operations underlying the system's semantics relating to their 5 configurable consistency levels. We show that this minimal client-focused specification of a large-scale service offers important design- and documentation-level insights, while keeping buy-in cost low.

We document the 2 person-month development process of our specification, which consists of iterative prototyping using the public documentation [9], feedback from author 2, a Cosmos DB developer, and the specification and model checking of a collection of formal properties based on our understanding. Aside from the specification itself, we discuss a pair of key issues it helped us discover within Cosmos DB's documentation, and how both have since been addressed. We also use our specification to explain the previously-unclear root cause of a 28-day high-priority outage within Microsoft Azure.

We describe the following results: (1) a concise (390 LOC) client-focused specification of Cosmos DB, a large-scale distributed system; (2) a pair of key documentation bugs we found by developing our specification — statements in Cosmos DB's public documentation [9] that have now been corrected; and (3) using our specification, a novel and concise mechanized explanation of a high-severity Cosmos DB-related outage within Azure that took 28 days to identify and mitigate.

Beyond our work so far, we expect our specification to be useful in future design work as Cosmos DB's implementation evolves, aided by its ability to precisely and abstractly state a client's expectations of system behavior. Services depending on Cosmos DB may also benefit from incorporating our work into TLA$^+$ specifications of their own processes, in which case our work may be used to prevent future outages similar to the one we describe in this paper.

II. Background

Our work uses the TLA$^+$ specification language [10], which can be used to describe state machines using set-theoretic constructs and temporal logic. Models written in TLA$^+$ have no direct correspondence to implementations, with users focusing instead on analyzing design decisions and verifying model-level correctness properties. This philosophy allows specification writers to leave out irrelevant details and focus on expressing a specification's core semantics as simply as possible.

In addition to plain TLA$^+$, model developers can also write models in PlusCal [11], a high-level imperative language that is more like contemporary programming languages. We use PlusCal to model the incident discussed in Section IV-C2, which demonstrates how to use our existing TLA$^+$ definitions from PlusCal.

It is possible to check model properties using the explicit-state model checker TLC [12], [13], the symbolic model checker Apalache [14], and the manual proof assistant TLAPS [15]. In this work, we relied on the TLC model checker to analyze our specification.

As well as model checking temporal properties, it is also possible to express and check refinements [16] in TLA$^+$. A refinement proves that one specification implements another – meaning that one specification exhibits every behavior that another specification exhibits, given an appropriate translation between the two specifications' state spaces. We use this technique to show that our new specification produces a superset of the behaviors produced by existing TLA$^+$ specifications of Cosmos DB.

III. A Simple Model of Cosmos DB

To fully illustrate our claim to simplicity, this section describes our full formalization of Cosmos DB's semantics in a few pages, including most of the core TLA$^+$ definitions in-text. While simple, our specification aggregates the expected observable behavior of client read and write operations at arbitrary scale without explicitly specifying details like replicas, server lifecycle, real time, or network traffic. Our specification abstractly represents an arbitrary number of clients communicating with an arbitrary number of Cosmos DB servers, including multiple regions, and considers all failure scenarios for which Cosmos DB is designed. The failure scenarios we consider include arbitrary machine and data center failures, and arbitrary loss and restoration of communication between any machines at any point.

Note that we specify only read and write operations. We consider the query APIs provided on top of Cosmos DB out of scope, since they must internally use the raw read/write mechanism that we do cover. We do not consider multi-region writes, which generally offer only weak consistency guarantees. We also leave out Cosmos DB's transaction mechanisms, limiting our work so far to arbitrary sets of concurrent single reads and writes at arbitrary consistency levels. We leave transactions as future work.

Our specification process was based on iterative discussion with author 2, a principal engineer working on the Cosmos DB implementation. We followed existing user-facing documentation, asked for feedback, learned more about the realities of Cosmos DB's design, and incorporated that new knowledge into our specification. We repeated this feedback loop until we found no more corrections. Instead, our model began to predict counter-intuitive but possible behaviors of the real system. See Subsection IV-D for in-depth analysis of such behaviors. See https://github.com/tlaplus/azure-cosmos-tla/tree/master/simple-model for the full TLA$^+$definitions.

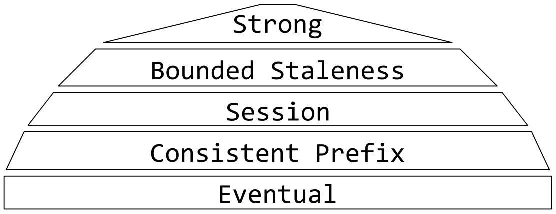

Fig. 1. Hierarchy of consistency levels in Cosmos DB, with strongest at the top and weakest guarantees at the bottom.

A. Consistency Levels

Cosmos DB offers 5 consistency levels that affect read and write behavior. A system administrator must configure all writes to follow a single consistency level per Cosmos DB deployment. Read operations may either match the configured write consistency level or weaken it according to the hierarchy defined in Figure 1. We discuss high-level prose descriptions of these consistency levels, which we complement with precise TLA$^+$ descriptions later on.

Strong consistency. Reads and writes are linearizable [17], as long as the operation does not fail. See Section IV-D for possible behaviors given failures.

Bounded staleness. Writes older than a given time bound are durable and consistently readable, whereas writes younger than the given bound are not. The time bound is defined two ways: a bound in wall-clock time, and a maximum bound on the number of eventually-consistent writes allowed at once. If the bounds are in danger of being exceeded, additional writes will be refused in order to allow all replicas to catch up. For modeling simplicity, we ignore the wall-clock temporal aspect of this mode's semantics, and consider only operation count. While modeling real time is possible in TLA$^+$, we use non-determinism to capture conceptually equivalent high-level behaviors without including a clock in our model's state space.

Session consistency. Reads and writes are tagged with *session tokens*. Operations with the same session token are linearizable relative to one another, but no guarantees are provided across different session tokens. Session consistency writes are not guaranteed durable, and tokens may be invalidated by data loss.

Consistent prefix. Reads are monotonic: a client may only read newer values than it has already read. Section IV-A describes how we used model checking to determine that this is ultimately equivalent to eventual consistency.

Eventual consistency. This level offers no ordering guarantees, but does provide a notion of eventual convergence over an arbitrary period of time[1].

[1] We mean logical time, as our specification does not consider wall-clock time.

B. Data Definitions

Our specification of Cosmos DB is defined to have 4 state variables, and allows them to evolve over time via some simple actions. Each variable relates to a specific aspect of the system being modeled. By defining actions that are allowed to non-deterministically alter these variables over time, these definitions are sufficient to represent the full range of Cosmos DB's expected behavior given an arbitrary deployment and scale.

log. The `log` is a sequence of writes, represented as key-value pairs. For example, $[$`key` $\mapsto$ `k1`, `value` $\mapsto$ `v1`$]$ pairs key `k1` with value `v1`. The sequence lists all writes that are stored anywhere in Cosmos DB's implementation, irrespective of replication or durability.

readIndex. The `readIndex` marks either a position in the `log` or 0. For any element of the `log`, if its index is less than or equal to `readIndex`, then it is replicated universally across servers within the current Cosmos DB deployment. Representing eventually-complete propagation of writes, the log prefix defined by `readIndex` behaves identically to a single key-value store.

commitIndex. The "commit index" marks a position in the `log` or 0. For any element of the `log`, if its index is less than or equal to `commitIndex`, then it is replicated at a global majority of replicas, and is therefore durable due to consensus. It follows from this definition that `readIndex` $\leq$ `commitIndex` must always hold.

epoch. The `epoch` is a monotonically increasing counter of fail-overs. If `epoch` remains constant, fail-over behavior such as data loss may not be observed. If it increases, some data loss may be observed at the point of increase.

The specification has six constants: `Keys` and `Values`, which are the sets of keys and values respectively. These sets can be redefined based on the use case — they can be generalized to infinite sets like "all strings", or restricted to a small finite set of constant values in order to allow exhaustive model checking. `NoValue` is a constant indicating the absence of a value. `VersionBound` and `StalenessBound` are natural numbers that affect when writes are allowed. `WriteConsistencyLevel` represents the currently configured consistency level for write operations, one of the 5 consistency levels.

We chose to base our specification on a sequential log because Cosmos DB, like any consensus-based system, determines a total order in which clients should consider their requests to have occurred. This is why many parts of our specification, including several state variables, identify writes by log index.

Building on these definitions, we can express our first two fundamental actions.

$$
\begin{aligned}
&\mathit{IncreaseReadIndexAndOrCommitIndex} \triangleq \\
&\quad \land\ \mathit{commitIndex}' \in \mathit{commitIndex} \mathrel{..} \mathit{Len}(\mathit{log}) \\
&\quad \land\ \mathit{readIndex}' \in \mathit{readIndex} \mathrel{..} \mathit{commitIndex}' \\
&\quad \land\ \text{UNCHANGED}\ \langle \mathit{log},\ \mathit{epoch} \rangle
\end{aligned}
$$

IncreaseReadIndexAndOrCommitIndex models the concept of data replication, that is, `readIndex` and/or `commitIndex` advancing. `readIndex` and `commitIndex` may non-deterministically gain new values between `readIndex` and `commitIndex`$'$, and `commitIndex` and **Len**(`log`), respectively. Neither `log` nor `epoch` may change. This ensures that both values may only grow, that they never point beyond end of the log, and that `readIndex` $\leq$ `commitIndex` remains true.

$$
\begin{aligned}
&\mathit{TruncateLog} \triangleq \\
&\quad \exists\, i \in (\mathit{commitIndex} + 1) \mathrel{..} \mathit{Len}(\mathit{log}) : \\
&\quad\quad \land\ \mathit{log}' = \mathit{SubSeq}(\mathit{log},\ 1,\ i - 1) \\
&\quad\quad \land\ \mathit{epoch}' = \mathit{epoch} + 1 \\
&\quad\quad \land\ \text{UNCHANGED}\ \langle \mathit{readIndex},\ \mathit{commitIndex} \rangle
\end{aligned}
$$

Log models the concept of data loss: if there exists any index i such that `commitIndex` $< i$, then `log` may be truncated non-deterministically such that its new length is $i - 1$. In-progress operations may watch for changes in `epoch`'s value to detect and respond to failures, meaning `epoch` acts as a failure detector.

Because these actions may happen non-deterministically, any combination of replication and fail-over may occur at any time, interleaved with other actions. A short sequence of such actions can represent a complex series of implementation-level possibilities.

C. Write Operations

In Cosmos DB, write operations are not atomic. They may sometimes appear atomic under certain configurations[2], but their underlying structure needs to be broken down into multiple steps.

As a consequence of writes' multi-step nature, we need to record the state of in-progress writes. For portability, we don't require any particular state retention mechanism, as the specifics might vary depending on how our core specification is used. Instead, we break up the two conceptual stages of a Cosmos DB write into re-usable parts that we describe individually. As a result, these definitions are only complete when combined appropriately, including for example additional `UNCHANGED` declarations where needed. What we present here completely encapsulates the core behavior of write operations, and we make public specific usage examples alongside our specification.

$$
\begin{aligned}
&\mathit{WritesAccepted} \triangleq \\
&\quad \land\ \mathit{Len}(\mathit{log}) - \mathit{readIndex} < \mathit{VersionBound} \\
&\quad \land\ ((\mathit{WriteConsistencyLevel} = \mathit{BoundedStaleness}) \Rightarrow \\
&\quad\quad \mathit{Len}(\mathit{log}) - \mathit{commitIndex} < \mathit{StalenessBound})
\end{aligned}
$$

1) Beginning a Write Operation: WritesAccepted determines whether a write may be attempted at all. It constrains writes based on two factors: `VersionBound` and

[2] For instance, a client performing only strongly consistent reads and strongly consistent writes will never witness an in-progress write. Weaker consistency levels do not provide any such guarantees, however. See Section IV-D for specific examples.

`StalenessBound`. `VersionBound` is a global limit on how many partially-replicated writes may exist in a Cosmos DB instance at any one time. `StalenessBound` is a global limit on how many non-durable writes may exist in a Cosmos DB instance at any one time, used to enforce bounded staleness consistency.

$$\begin{array}{l} \mathit{WriteInit}(\mathit{key}, \mathit{value}) \triangleq \\ \quad \mathit{log}' = \mathit{Append}(\mathit{log}, [\mathit{key} \mapsto \mathit{key}, \mathit{value} \mapsto \mathit{value}]) \end{array}$$

WriteInit defines the initial stage of any permitted write operation, appending a new key-value pair to the log. This means that at least one replica now holds the new key-value pair. The lack of distinction between incomplete and complete writes is intentional here: Cosmos DB replicas unconditionally begin serving writes as soon as they accept them. The Cosmos DB client libraries are the ones that enforce Cosmos DB's read semantics, and they may perform multiple read requests against multiple replicas until they get a consistent answer that can be returned to an end-user. Cosmos DB replicas require no additional logic restricting which writes should be visible to which read requests.

$$\begin{array}{l} \mathit{WriteInitToken} \triangleq \\ \quad [\mathit{epoch} \mapsto \mathit{epoch}, \mathit{checkpoint} \mapsto \mathit{Len}(\mathit{log}) + 1] \end{array}$$

WriteInitToken defines a unique identifier, or token, with which we can keep track of a write's progress. This token is structurally identical to a session token, the data used to identify a client's session at session consistency. Note that in practice, these tokens represent the flow of a request from client to server and back. We use this abstraction to concisely summarize an otherwise complex mix of network semantics and client-server interaction.

We have model-checked a uniqueness property for all session tokens given up to 6 writes and any one failure event.

2) Completing a Write Operation: Once it has begun, a write operation may complete *at any time that it is allowed to*. An in-progress write may also non-deterministically fail at any time, due to timeouts, spurious network failures, and so forth.

$$\begin{array}{l} \mathit{WriteCanSucceed}(\mathit{token}) \triangleq \\ \quad \land\ \mathit{SessionTokenIsValid}(\mathit{token}) \\ \quad \land\ (\mathit{WriteConsistencyLevel} = \mathit{StrongConsistency} \Rightarrow \\ \qquad\quad \land\ \mathit{token.epoch} = \mathit{epoch} \\ \qquad\quad \land\ \mathit{token.checkpoint} \leq \mathit{commitIndex}) \end{array}$$

Given a token identifying an in-progress write, *WriteCanSucceed* defines when the write is allowed to succeed. There are 3 conditions for success.

First, a write may succeed if its token is valid, that is, *SessionTokenIsValid*(`token`) is true. This will be the case if `token.checkpoint` $\leq$ **Len**(`log`), and `token.epoch` = `epoch`.

Second, writes must still be in the log. If data loss occurred and the written data is gone, success cannot be claimed. This condition also accounts for replacement, where log entries are lost then written again with the same index. Since data loss always increments `epoch`, rejecting writes from a different `epoch` cleanly disallows writes interrupted by data loss events.

Lastly, if `WriteConsistencyLevel` is set to `StrongConsistency`, then `token.checkpoint` must be less than or equal to `commitIndex`. By the semantics of `commitIndex`, this requirement means that "all strongly consistent writes observed by a client must be durable".

D. Read Operations

We define read semantics for Cosmos DB as stateless, read-only operators that describe the set of allowed read results for any given read request. Unlike write operations, individual read operations can be set to any consistency level that is weaker than or equal to the configured write consistency. That is why we define them as different operators, which makes it possible to perform and compare all possible types of read operation without changing the system's state. We define the read operation for each consistency level separately, but we use a common underlying definition called *GeneralRead* to avoid duplication.

$$\begin{array}{l} \mathit{GeneralRead}(\mathit{key}, \mathit{index}, \mathit{allowDirty}) \triangleq \\ \quad \text{LET } \mathit{maxCandidateIndices} \triangleq \{i \in \text{DOMAIN } \mathit{log} : \\ \qquad\qquad \land\ \mathit{log}[i].\mathit{key} = \mathit{key} \\ \qquad\qquad \land\ i \leq \mathit{index}\} \\ \qquad\quad \mathit{allIndices} \triangleq \{i \in \text{DOMAIN } \mathit{log} : \\ \qquad\qquad \land\ \mathit{allowDirty} \\ \qquad\qquad \land\ \mathit{log}[i].\mathit{key} = \mathit{key} \\ \qquad\qquad \land\ i > \mathit{index}\} \\ \quad \text{IN} \quad \{[\mathit{logIndex} \mapsto i, \mathit{value} \mapsto \mathit{log}[i].\mathit{value}] \\ \qquad\qquad : i \in \mathit{allIndices} \cup (\\ \qquad\qquad \text{IF} \quad \mathit{maxCandidateIndices} \neq \{\} \\ \qquad\qquad \text{THEN } \{\mathit{Max}(\mathit{maxCandidateIndices})\} \\ \qquad\qquad \text{ELSE } \{\})\} \cup \\ \qquad\quad (\text{IF} \quad \mathit{maxCandidateIndices} = \{\} \\ \qquad\quad\ \text{THEN } \{\mathit{NotFoundReadResult}\} \\ \qquad\quad\ \text{ELSE } \{\}) \end{array}$$

GeneralRead takes 3 parameters: `key`, whose value is being read; `index`, a log index indicating the reader's "point of view" in the log; and `allowDirty`, which determines whether the read operation should have exactly one result, or non-deterministically many. All members of the resulting set will be pairs of `logIndex` and `value`, which are resulting value and its log index, respectively. `logIndex` allows read results to be totally ordered, which is useful for both verifying correctness properties, and for correctly describing session tokens.

`index` defines a prefix of the log, selecting all indices $i \leq$ `index`. Within this prefix, *GeneralRead* will always include the latest mapping from `key` to some value. If there is no such mapping, *GeneralRead* returns the marker value *NotFoundReadResult*. Additionally, if `allowDirty` is true, then *GeneralRead* will also include values bound to `key` in log entries with index $i >$ `index`. This models non-deterministic reads: it allows reading writes that are not durable, still in progress, or simply arbitrarily more recent than `index`.

Note that each of the following read operations are only valid for compatible values of `WriteConsistencyLevel`. Figure 1 illustrates the intended hierarchy of consistency levels.

```
$StrongConsistencyRead(key) \triangleq$
    $GeneralRead(key,\ commitIndex,\ \text{FALSE})$
```

1) Strongly Consistent Reads: Strongly consistent reads for any given key follow `commitIndex`, and return one single consistent value in all cases. Aligning these reads with `commitIndex` means that only durable writes may be read.

```
$BoundedStalenessRead(key) \triangleq$
    $GeneralRead(key,\ commitIndex,\ \text{TRUE})$
```

2) Bounded Staleness Reads: Bounded staleness reads also follow `commitIndex`. Unlike strongly consistent reads, bounded staleness reads may see arbitrary information beyond `commitIndex`. The span of log entries between `commitIndex` and **Len**(`log`) represents the non-durable reads allowed, which may be arbitrarily witnessed in addition to durable data before `commitIndex`.

```
$SessionConsistencyRead(token,\ key) \triangleq$
  IF   $\lor\ epoch = token.epoch$
       $\lor\ token = NoSessionToken$
  THEN LET $sessionIndex \triangleq Max(\{token.checkpoint,$
                                  $readIndex\})$
       IN  $GeneralRead(key,\ sessionIndex,\ \text{TRUE})$
  ELSE $\{\})$
```

3) Session Consistent Reads: Session consistent reads operate using a session token which defines a position in the log to read from: a `checkpoint`, and the `epoch` from which the token originates.

The first check made during a session consistency read is whether the session token is from the current `epoch`. If the epochs differ, and the session token isn't the placeholder value *NoSessionToken*, then no reads are permitted. Session consistency offers no durability guarantees: if data loss occurs, it becomes impossible to guarantee that writes referenced by a session token remain intact. Not all session tokens will be invalidated on every data loss event in practice, but we have yet to find a need for modeling the invalidation of only some session tokens.

After checking the `epoch`, the `checkpoint` is combined with `readIndex`. Since the `readIndex` indicates the log prefix that has been replicated to every single replica in the current Cosmos DB deployment, it would be unsound to have a `sessionIndex` smaller than `readIndex`.

We set `allowDirty` to `TRUE`, meaning that a session consistent read may arbitrarily read log entries beyond its session token. This possibility represents clients' ability to non-deterministically witness the effects of other concurrent sessions.

Note that the "empty" value, *NoSessionToken*, corresponds to $[\texttt{epoch} \mapsto 0, \texttt{checkpoint} \mapsto 0]$. Its `epoch` of 0 makes it incomparable to other session tokens, and its checkpoint of 0 places no constraint on the outcome of a session consistency read.

```
$UpdateTokenFromRead(origToken,\ read) \triangleq [$
    $epoch \mapsto epoch,$
    $checkpoint \mapsto Max(\{origToken.checkpoint,$
                              $read.logIndex\})$
$]$
```

Once a read is performed with a given token, a client must update its session token. This is done with *UpdateTokenFromRead*, which combines the log index from a read result with the checkpoint of an existing session token. This combination monotonically increases a client's session token, ensuring that each client may only witness increasingly recent information.

```
$EventualConsistencyRead(key) \triangleq$
    $GeneralRead(key,\ readIndex,\ \text{TRUE})$
```

4) Consistent Prefix and Eventual Consistency Reads: Consistent prefix and eventual consistency being known equivalent, as discussed in Section IV-A, they have identical definitions. Their behavior is minimally constrained, requiring only that values overwritten at or before `readIndex` cannot be read.

E. Validation

To validate that our specification exhibits behaviors of which Cosmos DB's implementation is capable, and in order to ensure that we cover as wide a variety of these behaviors as possible, we have leveraged a combination of model checking correctness properties, model checking our specification's relationship with comparable specifications via refinement, and manual expert review of behaviors implied by our specification. This subsection focuses on the properties we checked, while particularly interesting specific behaviors will be discussed alongside our results in Section IV.

1) Correctness Properties: The correctness properties we check are a collection of the ones listed in Cosmos DB's external documentation [9], properties derived from existing TLA$^+$ specifications of Cosmos DB [18] (which are also referenced as authoritative by Cosmos DB's documentation), and properties inherent to our particular specification's design. To aid in our verification process, we extend our base behavior specification with an auxiliary `writeHistory` state variable. `writeHistory` provides a history of all attempted writes, including which key, which value, a write token indicating at which epoch and log index the write began, and a state that will transition at most once from `WriteInitState` to either `WriteSucceededState` or `WriteFailedState`.

Using this extended specification, we verify a total of 10 liveness properties and 14 safety properties across the 4 distinct data consistency levels offered by Cosmos DB, excluding basic type safety invariants. Our verification process is based on model checking, using a combination of exhaustive state space exploration of logs up to length 6, and depth-first random simulation of execution traces exploring up to 100 steps.

$$
\begin{aligned}
&\mathit{PointsValid} \triangleq \\
&\quad \Box[\ \land\ \mathit{readIndex} \leq \mathit{commitIndex} \\
&\qquad \land\ \mathit{readIndex} \leq \mathit{readIndex}' \\
&\qquad \land\ \mathit{commitIndex} \leq \mathit{commitIndex}'\,]_{vars}
\end{aligned}
$$

For example, PointsValid defines the relationship between readIndex and commitIndex: readIndex cannot be beyond commitIndex, and they must increase monotonically. For the sake of concision, we describe the other properties via prose summary. The full set is available alongside our complete specification at https://github.com/tlaplus/azure-cosmos-tla/tree/master/simple-model.

Read your writes. For strong consistency and session consistency with the same token, after any write, only the written value or some later write may be read.

Read after write. It is related to *read your writes*, but at a client scope rather than a global scope. It is not materially different in our specification, since we abstract away communication in its entirety, but we check it as an alternate form of *read your writes* for completeness.

Monotonic reads. For strong consistency and session consistency with the same token, reads may only make visible later writes, and will never return older data than they already have.

Bounded staleness. Bounded staleness consistency should never accept more than StalenessBound uncommitted writes at once.

Session token lifetime. For any arbitrary session token that is valid, it will either remain valid or become invalid, in which case it will never become valid again.

readIndex as lower bound. No reads may return values that have been overwritten by other operations within the log prefix defined by readIndex.

Write completion. All writes eventually complete, either with success or failure.

2) Linearizability: The strongest consistency property offered by Cosmos DB is the linearizability [17], [19] of write operations at the StrongConsistency consistency level. Linearizability means that, for any operation on some concurrent object (here, a key in Cosmos DB), we can choose a point in time between the beginning and end of that operation at which it has atomically occurred. For our simple specification, that point is when commitIndex is incremented. Due to our non-atomic modeling of writes, this point will occur at some unspecified point in between the beginning and end of a successful write, whenever *IncreaseReadIndexAndOrCommitIndex* takes place. We wrote a refinement specification CosmosDBLinearizability, verifying that every behavior of our Cosmos DB specification with strong consistency reads and writes corresponds to the same series of atomic reads and writes applied to a TLA$^+$ function.

3) Refining Existing Specifications: Cosmos DB already has publicly available TLA$^+$ specifications for some of its behavior [18], so we used refinement to verify our new specification does not disagree with existing specifications. Our refinement was possible using a direct mapping between states, and we provide the necessary definitions alongside our specification in the file RefineGeneralModel.tla.

We found that our work offers a superset of previously-specified behavior, despite the old specification including concepts we do not explicitly deal with, representing individual servers and network messages. Our specification's behavior is specifically a strict superset of the old one's, because we noted that the existing specifications made no attempt at modeling data loss or relaxed reads.

4) Refining Read Consistency Levels: It is strongly hinted in Cosmos DB's public documentation [9] that different consistency levels represent a hierarchy of possible behaviors, with stronger consistency guarantees forming subsets of weaker consistency guarantees. We used our specification to investigate this property, and determined under what conditions the implication made by Cosmos DB's public documentation holds.

We found that, *for the same configured write consistency*, different consistency reads form behavioral subsets directly matching the documented hierarchy illustrated in Figure 1. Keeping the write consistency level constant, each stronger read consistency allows a subset of the behavior of each weaker read consistency. Note that we consider all possible session token choices together for session consistency.

Counter-intuitively, this relationship does not hold when comparing write consistency levels. Consider that strong consistency allows more non-durable writes than bounded staleness, because bounded staleness fundamentally relies on throttling writes to preserve its semantics, whereas strong consistency does not. See Section IV-D3 for discussion.

IV. Results

Beyond our specification itself, we showcase two key issues it helped us raise with Cosmos DB's documentation, both of which have been addressed. We also present the previously-unclear root cause of a 28-day high-priority outage within Azure, alongside a collection of other properties of Cosmos DB made explicit by our project.

While our specification itself does not need to reference the underlying architecture of Cosmos DB, it is necessary to do so when discussing the intuition behind our results. We begin with a brief glossary of the relevant architectural details of Cosmos DB.

Replicas. A Cosmos DB deployment is composed of a number of replica servers ("replicas"), each of which maintains an independent version of the database's contents. All replicas respond to load-balanced client requests. Cosmos DB's client libraries are responsible for retry logic.

Regions. A Cosmos DB deployment's replicas are grouped into regions based on physical proximity in order to manage communication latency. Some operations require consensus only within a single region ("local" consensus), and some

TABLE I
READ AND WRITE STRATEGIES AT DIFFERENT CONSISTENCY LEVELS, TAKEN FROM PUBLIC COSMOS DB DOCUMENTATION [9].

Consistency Level	Quorum Reads	Quorum Writes
Strong	Local Minority	Global Majority
Bounded Staleness	Local Minority	Local Majority
Session	Single Replica (session token)	Local Majority
Consistent Prefix	Single Replica	Local Majority
Eventual	Single Replica	Local Majority

operations require consensus across regions ("global" consensus). Table I lists the consensus requirements for read and write operations at different consistency levels within Cosmos DB. "Majority" means strictly more than 50%, and "minority" means at least 50%.

Write region refers to a single region that is allowed to process write operations. This is similar to the well-known concept of a leader, but with regions instead of individual replicas. If an entire region becomes unresponsive, it is possible to fail-over between regions, similarly to the well-known concept of leader re-election.

A. Consistent Prefix and Eventual Consistency Behave Equivalently

During our work, we have discovered that eventual consistency and consistent prefix in Cosmos DB behave identically from the point of view of a client performing individual reads and writes. We modeled and checked all properties of both consistency levels, and we were unable to find any practical distinction between the two modes for single reads and writes. We also explicitly model checked that both consistencies produce identical client views in all scenarios, and were able to confirm our findings with the Cosmos DB team. As a result, changes to Cosmos DB's public documentation were published. Below, we discuss the rationale that we developed in order to explain this result.

Since eventual consistency is the least constraining option, we will focus on whether consistent prefix could behave in any way that is distinguishable from it. For context, consider the original description of consistent prefix consistency below, which has now been rewritten by the Azure documentation team in response to our findings.

> In consistent prefix option, updates that are returned contain some prefix of all the updates, with no gaps. Consistent prefix consistency level guarantees that reads never see out-of-order writes.
> If writes were performed in the order ⟨A,B,C⟩, then a client sees either ⟨A,A,B⟩, or ⟨A,B,C⟩, but never out-of-order permutations like ⟨A,C⟩ or ⟨B,A,C⟩. Consistent Prefix provides write latencies, availability, and read throughput comparable to that of eventual consistency, but also provides the order guarantees that suit the needs of scenarios where order is important.
>
> Azure Cosmos DB Documentation on Consistent Prefix [9]

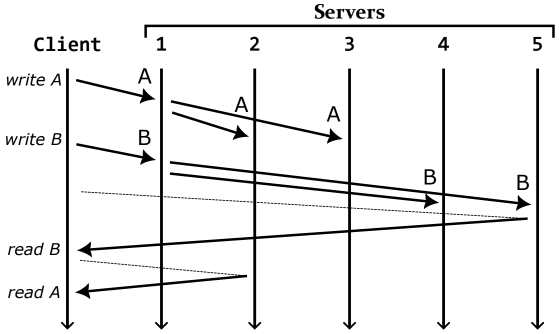

Fig. 2. Interaction diagram of a possible scenario for producing read pattern ⟨B,A⟩ under consistent prefix with a cluster of 5 servers.

Looking at the examples in the above excerpt, the documentation claimed that neither ⟨A,C⟩ nor ⟨B,A,C⟩ should be observable by clients. We assume the scenario described involves some implicit key k and values ⟨A,B,C⟩ written to key k in sequence alongside 3 concurrent read operations, all under consistent prefix. In that case, our specification of Cosmos DB allows both sequences of reads that the documentation claims are forbidden.

The first sequence, ⟨A,C⟩, is possible because the concurrent interleaving ⟨write(A), read(A), write(B), write(C), read(C)⟩ should naturally be possible, even if all operations were globally atomic. We are not sure why this counter-example was claimed to be invalid.

The second sequence, ⟨B,A,C⟩, is a more complex case. We have confirmed that read operations in Cosmos DB are load balanced to potentially any replica, and that any replica will immediately serve any data that is replicated to it. Following the information from Table I, we know that consistent prefix read operations will go to only one replica, and that consistent prefix write operations will be considered successful once data has been committed by a local majority of replicas in a single region. Figure 2 illustrates a possible scenario with one client and 5 replicas that will produce the read pattern ⟨B,A⟩ [3], while following all known implementation-level semantics of consistent prefix consistency.

First, assuming some arbitrary single key k, the client writes values A and B to local majorities. There are 5 replicas, so 3 servers must commit each write. The first write goes to replicas $1, 2, 3$, and the second write goes to replicas $1, 4, 5$. Then, the client performs two reads in quick succession. Due to arbitrary load balancer behavior, the first read is served by replica 5, and the second read is served by replica 2. Each replica serves its latest local copy of the data bound to key k, which, due to how local majorities were chosen during the earlier writes, and assuming no replication has time to take place, produces the sequence of reads ⟨B,A⟩.

Together, these two counter-examples negate the only documented difference between consistent prefix and eventual consistency for atomic writes to the same key.

[3] We omit C from our example, as additionally writing then reading C after seeing ⟨B,A⟩ is intuitive, strongly consistent behavior.

B. Regions Do Not Affect Safety Guarantees

Building on the idea that consistent prefix and eventual consistency behaviors are identical, we arrive at a second question regarding Cosmos DB's public documentation: why is data consistency so strongly dependent on how regions are configured? To illustrate, Cosmos DB's consistency documentation [9] contains 13 bullet points across 3 sections indicating consistency expectations that depend on the region in which a client is interacting with Cosmos DB. 12 of those bullet points list either consistent prefix or eventual consistency as the expected behavior, which we found to be equivalent.

Given how many of these bullet points be argued redundant according to our specification, we gave thought to whether they could all be removed. Making the documentation simpler in this way would be a net positive to potential readers who seek to understand Cosmos DB's consistency guarantees.

The 13th bullet point that lists a consistency level other than eventual consistency or consistent prefix applies to bounded staleness, when bounded staleness reads go to the same region as writes. That bullet point claims that, under those conditions, bounded staleness offers guarantees identical to strong consistency. Given that both reads and writes under bounded staleness perform region-local consensus, we can understand why this case would often be equivalent to strong consistency in practice: within the same region, it would be impossible for a client to see any out-of-order artifacts. The missing condition is durability: Cosmos DB supports write region fail-over, whereby the write region can be changed if the original write region has become unavailable. In that case, the new write region might not have replicated all of the data in the original write region, or might lag behind other regions that are still available, allowing both data loss and stale reads. This would not be the case for strong consistency, which requires global consensus during writes, meaning that changing write regions would not create any client-visible inconsistencies.

When our issues were addressed, it was confirmed that for atomic single reads and writes, our arguments are valid. These bullet points remain due to an additional detail that is out of scope for our specification: transactions. Cosmos DB supports optimistic concurrency via a transaction engine layered on top of the raw reads and writes our specification supports, which acts differently under consistent prefix and eventual consistency. Formally specifying this new information, as we have done for the original, may be an interesting direction for future work.

C. Investigating a High-Impact Production Outage

Our work was motivated by a production outage which impacted the ability of thousands of Azure customers to deploy or update their critical cloud resources for over a month[4]. The outage was the result of a subtle and hard to detect consistency bug introduced in an attempt to improve system performance, and the full repair of this issue has necessitated a costly, multi-year, redesign of the underlying system's storage architecture. The fact that this change was introduced by, and under the guidance of, experts on this system highlights the challenges of manually verifying the consistency models of large distributed systems, and the risks associated with their failure.

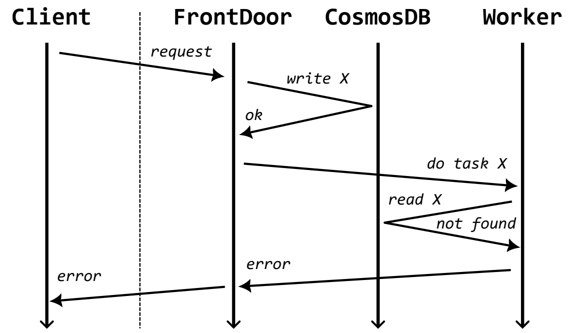

Fig. 3. Interaction diagram of the error underlying a high-impact outage within Azure Cloud.

We have used our work to model the semantics underlying the outage. As a result, we have been able to identify the previously-unidentified safety issue underlying the outage. We presented our analysis to the author of the original outage postmortem, and they confirmed that our explanation made sense within the context of their work. Note that our presentation here summarizes the structure of a system that depends on Cosmos DB. Cosmos DB itself should be understood to work as it is presented in Section III, and additional components are part of the surrounding system whose investigation we are presenting.

1) Outage Postmortem and Investigation: Figure 3 illustrates the underlying structure of the outage, as reported in the postmortem. It is already clear that a data consistency issue is occurring: the *FrontDoor* server makes a complete, successful write to Cosmos DB, and then the *Worker* server tries to read that write and fails. The reported explanation for this issue was based on regions and latency. *FrontDoor* was performing writes in one cloud region, and *Worker* was reading those writes in another region. Prior to the change, these writes and reads were occurring in the same region, and errors were not happening at a noticeable rate. The change in routing lead to a change in latency, which caused *Worker* to read out of date information from Cosmos DB.

From our analysis, while the original postmortem's comments were correctly diagnosing the change in latency, some correctness-critical factors were not discussed. We found that Cosmos DB was configured for session consistency, and we confirmed that each server (*FrontDoor*, *Worker*) was working with unconfigured, arbitrary session tokens. From the definition of session consistency, without sharing session tokens, the two servers were only guaranteed eventually consistent reads. So, the semantic problem had always existed, but it was only exposed in practice by a change in region configuration.

2) Modeling the Outage: Our TLA$^+$ specification allows us to verify the abstract scenario from Figure 3, and check our understanding against our specification of Cosmos DB. In Listing 1, we define a collection of actions corresponding to two

[4] Documentation for this outage is not public, and we explain any necessary context in-text. Its internal write-up is available here for those with access: https://portal.microsofticm.com/imp/v3/incidents/postmortem/521677.

```
CosmosDB ≜ INSTANCE CosmosDB

VARIABLES serviceBus, frontdoorPC, frontdoorToken,
          workerPC, workerToken, workerValue

Init ≜
  ∧ WriteConsistencyLevel = SessionConsistency
  ∧ serviceBus = ⟨⟩
  ∧ frontdoorPC = "frontdoorWriteTaskDataInit"
  ∧ frontdoorToken = NoSessionToken
  ∧ workerPC = "workerBeginTask"
  ∧ workerToken = NoSessionToken
  ∧ workerValue = NoValue
  ∧ CosmosDB ! Init

frontdoorWriteTaskDataInit ≜
  ∧ frontdoorPC = "frontdoorWriteTaskDataInit"
  ∧ CosmosDB ! WriteInit("taskKey", "taskValue")
  ∧ frontdoorToken′ = CosmosDB ! WriteInitToken
  ∧ frontdoorPC′ = "frontdoorWriteTaskDataCommit"
  ∧ UNCHANGED ⟨serviceBus, workerPC, workerToken,
                workerValue⟩

frontdoorWriteTaskDataCommit ≜
  ∧ frontdoorPC = "frontdoorWriteTaskDataCommit"
  ∧ CosmosDB ! WriteCanSucceed(frontdoorToken)
  ∧ serviceBus′ = ⟨"taskKey"⟩
  ∧ frontdoorPC′ = "Done"
  ∧ UNCHANGED ⟨frontdoorToken, workerPC,
                workerToken, workerValue⟩

frontdoorDone ≜
    ∧ frontdoorPC = "Done"
    ∧ UNCHANGED vars

workerBeginTask ≜
  ∧ workerPC = "workerBeginTask"
  ∧ serviceBus ≠ ⟨⟩
  ∧ LET taskKey ≜ Head(serviceBus)
    IN  ∧ serviceBus′ = Tail(serviceBus)
        ∧ ∃ read ∈ CosmosDB ! SessionConsistencyRead(
                workerToken, taskKey) :
              ∧ workerToken′ =
                    CosmosDB ! UpdateTokenFromRead(
                      workerToken, read)
              ∧ workerValue′ = read.value
              ∧ workerPC′ = "Done"
              ∧ UNCHANGED ⟨frontdoorToken,
                            frontdoorPC⟩

workerDone ≜
  ∧ workerPC = "Done"
  ∧ UNCHANGED vars
```

Listing 1: A TLA^{+} specification of the behavior underlying the events in Figure 3.

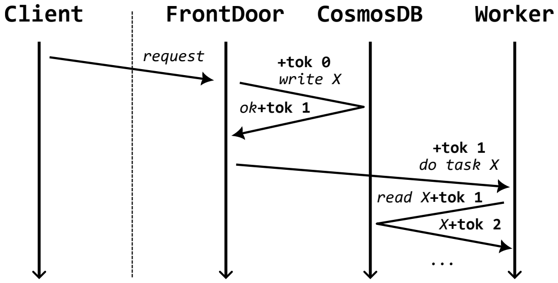

Fig. 4. Amended interaction diagram showing a correction of the problem in Figure 3.

processes that interact with Cosmos DB. We reference our core Cosmos DB definitions using `INSTANCE`, and which brings into scope our existing definitions with the prefix `CosmosDB!`. We include the complete set of actions allowed for each client, but omit boilerplate for presentation. Control flow constraints use the variables ending in `PC`. The full text is available under the name `show521677simple.tla` alongside our core specification. There is an equivalent PlusCal version named `show521677simplePCal.tla`.

Corresponding to Figure 3, the two processes in Listing 1 are called `frontdoor` and `worker`, which match 2 of the 4 processes in the figure. The processes communicate with each other using the shared state variable `serviceBus`. Process-local variables are prefixed by the name of their owner. Note that we omit the client shown in Figure 3 by starting our specification at the point where `frontdoor` receives the client request. The behavior of Cosmos DB is taken from our existing definitions.

Cosmetic differences aside, the underlying series of actions is the same as in Figure 3: the `frontdoor` writes some value (here, `X` is `"taskValue"`). The write occurs in two steps: one to begin the write, and one to await the write's success. Assuming the write succeeds, `frontdoor` writes `"taskKey"` to `serviceBus`, requesting that the worker perform some task named `"taskKey"` (`X` in the figure). To perform the task, `worker` reads the task data from Cosmos DB using session consistency with a null session token, storing the value and updated session token that it receives in `workerValue` and `workerToken` respectively.

3) Counter-Example: Based on the original issue's data consistency expectations, we can formulate an expected property for our specification in temporal logic: $\Diamond$ `workerValue` = `"taskValue"`. That is, eventually the state variable `workerValue` will hold the value `"taskValue"` written by `frontdoor`. Model-checking our property generates a counter-example. In that counter-example, `readIndex` and `commitIndex` were both at 0, meaning no replication had taken place, and that unconstrained session-consistency reads could go to replicas that did not have our single write of `"taskValue"`. Using a small amount of PlusCal alongside our specification of Cosmos DB, we were able to accurately recreate the semantic issue underlying a high-impact outage.

Our proposed semantic fix is to pass a session token from `frontdoor` to `worker`, and use it as a starting value for `workerToken` instead of `CosmosDB!NoSessionToken` when the worker reads from Cosmos DB. Making this change and re-checking the model, we find that the error no longer occurs. Figure 4 illustrates the modified behavior we expect, with versioned `+tok` 0, 1 . . . annotations indicating the process of passing along and keeping up to date a session token `tok` between *FrontDoor* and *Worker*.

D. Notable Anomalies

By writing our specification, we found multiple anomalous behaviors that we suspected to be specification bugs. However, in each case it has been confirmed that these represent real behaviors of which Cosmos DB is capable. These behaviors are not explicitly mentioned in the documentation, and we discovered them purely by model checking, manually examining the semantics of our specification, and discussing our results with author 2, a Cosmos DB expert.

1) Dirty Reads: Strongly consistent writes, which we expect to be linearizable, are only linearizable in relation to strongly consistent reads. Other read consistency levels allow dirty reads, which do not follow linearizability. Since write operations are not atomic, reads with a consistency level other than strong may see incomplete writes, because they are able to see non-durable writes in general.

Following a more implementation-focused analogy, session and eventual consistency reads are only served by one replica in a region. Each replica immediately begins serving writes it receives without waiting for the writes to fully replicate, so if a read request reaches a replica holding an unreplicated write, then that read can witness an in-progress strongly consistent write operation. A similar scenario is possible for bounded staleness reads under strong consistency, where a write might be replicated to one region but not a global majority; a bounded staleness read can likewise be served by a region that stores an in-progress strongly consistent write.

The non-atomicity of strongly consistent writes is counter-intuitive and not well-known. Previous drafts of the specification we present did not include this feature, until we described to author 2 that our formulation effectively assigned concurrency barrier semantics to strong consistency writes, which is incorrect. The resulting explanation of the true guarantees offered by strong consistency writes inspired the current version of our specification.

2) Durable Failed Writes: Clients may also read the values written by failed writes. Our specification does not remove or invalidate log entries when a client might observe a write failure, because the write may still succeed after that point. Cosmos DB will attempt to unconditionally complete a write operation even if its notification does not reach the requesting client, and all the client observes is a failed request.

While this anomaly is well known, it can be counter-intuitive to developers. Future documentation may benefit from discussing this possibility, as well as the particular trade-off made by Cosmos DB.

3) Bounded Staleness Reads are Weaker Under Strongly Consistent Writes: Because the guarantees underlying bounded staleness are enforced at write time, performing a bounded staleness read while Cosmos DB is configured for strongly consistent writes does not actually guarantee the same set of bounds as when bounded staleness writes are configured. Under strong consistency, there is no bound on the number of in-progress write requests. As a result, bounded staleness reads are not subject to any bounds either, and will return either the same result as a strongly consistent read, or a dirty read from an in-progress strongly consistent write.

We do not expect this more obscure anomaly to cause problems for developers in practice, but it is important to keep note of it in documentation and future design discussions.

V. Discussion

The results of our work specifying Cosmos DB shows that a minimal, purely client-facing specification of a sufficiently complex distributed system has many uses in practice. Our specification effort enabled us to suggest several improvements to Cosmos DB's public-facing documentation, as well as to precisely diagnose the root cause of a high-impact outage within Azure Cloud.

Our outcome is a useful intersection between focusing on implementation correctness and focusing on the purely theoretical properties of an abstract *kind of system*. By keeping our specification at the interface level, we were able to successfully avoid the complexity of Cosmos DB's low-level implementation semantics, while still producing useful practical insights into the behavior of the system we studied.

Our lack of interaction with Cosmos DB's implementation is a double-edged sword, in that it is possible that some error in our specification has escaped the notice of those reviewing it. If a similar specification were more integrated with Cosmos DB's development, techniques such as trace validation [20] could be employed to perform automatic checks that the implementation and specification match. While we did not have the opportunity to go beyond manual review, this limitation is not fundamental to the techniques we describe, nor is it a fatal flaw in our specific situation. Our TLA$^+$ specification is compact enough that isolating and fixing any error is not difficult: our core specification is only 390 lines long, including comments and whitespace. For example, once we were informed dirty reads should be possible, it took us only 2-3 days to rewrite our specification's write semantics from fully atomic to the current two-step version, then adapt and re-verify any affected correctness conditions.

This is why we believe that, despite the lack of automated linkage between our specification and Cosmos DB's implementation, it is practical to keep the specification up to date in the face of any significant design or implementation changes to Cosmos DB. In fact, analysis of what effect a design change would have on client-observable behavior would likely be beneficial to the discussion of that design change. Additionally, as we have initially explored in Section III-E3, TLA$^+$ can be used to explore refinement relationships between our client-level specification and other internal implementation-level specifications, such as those currently in use by the Cosmos DB development team.

Outside of Cosmos DB's documentation, our work can be used to precisely model individual interactions with Cosmos DB. Our work could particularly benefit systems that depend on Cosmos DB's semantics when specifying their own behavior, which could previously not be modeled in great detail due to the lack of a re-usable and precise specification of Cosmos DB's client-observable behavior. This is made possible by our focus on specifying Cosmos DB's interface, since implementation-level specifications will often be too complex, or have a larger state space than is viable for model checking, to easily be used as components of other specifications.

VI. Related Work

There exist multiple perspectives on studying the observable behavior of distributed key-value stores: abstract formal reasoning, formal methods operating on both specifications and implementations, and client-level testing tools.

Formally, database consistency properties have been well studied in the abstract [21], [17]. In particular, [22]'s focus on client-observable system states partially inspired our specification strategy for Cosmos DB.

In formal methods, efforts are ongoing to specify and verify the correctness of distributed system implementations. Verifying that an implementation satisfies a given specification can be a powerful tool, but it often requires that the implementation has a specific structure, often requiring verification to be part of the development process from the beginning [6], [7], or at least deeply integrated into the development process [8]. The adoption cost of such techniques may prove prohibitive for existing large, unverified codebases.

Tools to explore possible behaviors of an unmodified implementation have been successfully developed [23], [24], but these tools focus on exposing implementation bugs rather than studying the set of valid client-observable behaviors.

Client-level testing tools also exist [25], [26], but this work focuses on more general-purpose anomalies, or relies on user-provided definitions for dependencies like databases. Database semantics, especially quirks of a specific implementation, are hard to define and reason about. Mock implementations and simulation modes for complex database services cannot be built as an afterthought. Our work provides a well-reasoned starting point for building any client-level testing tools specialized to Cosmos DB and its anomalies.

MonkeyDB [27] provides a general-purpose definition of database consistency semantics, which it uses to simulate client code interactions with databases. We believe our approach is complementary to this kind of more general-purpose simulator, in that our implementation-specific specification offers a different set of semantics to simulate, potentially including quirks that are unique to Cosmos DB.

Elle [28] automatically validates database consistency guarantees by analyzing the outcomes of synthetic query sequences. It may be useful in both exploring the actual semantics of a black-box database implementation, and in data consistency bug-finding.

VII. Conclusion

We have presented what can best be described as *the lightest-weight useful specification of Azure Cosmos DB's semantics in TLA$^+$*. Despite its structural simplicity, our specification covers all 5 advertised data consistency levels available to clients. It represents behaviors with arbitrary configurations of regions and replicas, including arbitrarily complex scenarios involving delayed replication, server and region failure, and otherwise data loss.

Our new specification has been validated by a combination of model checking, refinement with existing incomplete specifications, and expert review. While we are now confident in our specification's correctness, should any bugs be found in it, our specification is also small enough that fixing them would not require inordinate amounts of work.

We have used our specification to predict multiple under-documented anomalous behaviors of Cosmos DB, and to raise two now-addressed issues with the service's publicly-available documentation. We have also used our specification to elaborate on the root cause of a high-impact outage within Azure Cloud, successfully producing an abstract explanation for the underlying series of events.

In the future, we expect our specification to be be usable by the Cosmos DB development team to reason about their service's client-facing behavior, in conjunction with their own implementation-level TLA$^+$ specifications via refinement. Beyond benefits to Cosmos DB specifically, our compact specification can also be used to specify systems dependent on Cosmos DB, growing the set of systems for which formal verification is viable.

Our results show the value of using formal verification in industry, even without any interaction with the target system's implementation at all. The benefits in terms of understanding and documenting a system's expected behavior are still significant for end-users and developers.

Acknowledgment

We thank the authors of existing specifications of Cosmos DB, whose work we used as a starting point: Murat Demirbas, Ailidani Ailijiang, and Dharma Shukla. We also thank Microsoft engineer Ben Pannell for helping us understand the cloud incident discussed in Section IV-C. We thank Leslie Lamport, Ivan Beschastnikh, and our anonymous reviewers for their writing feedback. This work is supported in part by the Institute for Computing, Information and Cognitive Systems (ICICS) at UBC.

References

[1] H. Liu, S. Lu, M. Musuvathi, and S. Nath, "What bugs cause production cloud incidents?" in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 155–162. [Online]. Available: https://doi.org/10.1145/3317550.3321438

[2] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, 2011.

[3] J. Sillito, G. C. Murphy, and K. De Volder, "Asking and answering questions during a programming change task," *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 434–451, 2008.

[4] G. Uddin and M. P. Robillard, "How api documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[5] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "How amazon web services uses formal methods," *Commun. ACM*, vol. 58, no. 4, p. 66–73, mar 2015. [Online]. Available: https://doi.org/10.1145/2699417

[6] J. Bornholt, R. Joshi, V. Astrauskas, B. Cully, B. Kragl, S. Markle, K. Sauri, D. Schleit, G. Slatton, S. Tasiran, J. Van Geffen, and A. Warfield, "Using lightweight formal methods to validate a key-value storage node in amazon s3," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 836–850. [Online]. Available: https://doi.org/10.1145/3477132.3483540

[7] K. Bhargavan, B. Bond, A. Delignat-Lavaud, C. Fournet, C. Hawblitzel, C. Hritcu, S. Ishtiaq, M. Kohlweiss, R. Leino, J. Lorch, K. Maillard, J. Pan, B. Parno, J. Protzenko, T. Ramananandro, A. Rane, A. Rastogi, N. Swamy, L. Thompson, P. Wang, S. Zanella-Béguelin, and J.-K. Zinzindohoué, "Everest: Towards a Verified, Drop-in Replacement of HTTPS," in *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), B. S. Lerner, R. Bodík, and S. Krishnamurthi, Eds., vol. 71. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 1:1–1:12. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7119

[8] A. Chudnov, N. Collins, B. Cook, J. Dodds, B. Huffman, C. MacCárthaigh, S. Magill, E. Mertens, E. Mullen, S. Tasiran, A. Tomb, and E. Westbrook, "Continuous formal verification of amazon s2n," in *Computer Aided Verification*, H. Chockler and G. Weissenbacher, Eds. Cham: Springer International Publishing, 2018, pp. 430–446.

[9] Microsoft, "Consistency levels in Azure Cosmos DB," Jun 2022, accessed: 2022-08-17. [Online]. Available: https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels

[10] L. Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, June 2002. [Online]. Available: https://www.microsoft.com/en-us/research/publication/specifying-systems-the-tla-language-and-tools-for-hardware-and-software-engineers/

[11] ——, "The PlusCal algorithm language," in *Theoretical Aspects of Computing - ICTAC 2009*, M. Leucker and C. Morgan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 36–60.

[12] M. A. Kuppe, L. Lamport, and D. Ricketts, "The TLA+ toolbox," *Electronic Proceedings in Theoretical Computer Science*, vol. 310, pp. 50–62, dec 2019. [Online]. Available: https://doi.org/10.4204%2Feptcs.310.6

[13] Y. Yu, P. Manolios, and L. Lamport, "Model checking TLA+ specifications," in *Correct Hardware Design and Verification Methods*, L. Pierre and T. Kropf, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 54–66.

[14] I. Konnov, J. Kukovec, and T.-H. Tran, "TLA+ model checking made symbolic," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, oct 2019. [Online]. Available: https://doi.org/10.1145/3360549

[15] S. Merz and H. Vanzetto, "Automatic verification of TLA + proof obligations with SMT solvers," in *Logic for Programming, Artificial Intelligence, and Reasoning*, N. Bjørner and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 289–303.

[16] M. Abadi and L. Lamport, "The existence of refinement mappings," in *Proceedings of the 3rd Annual Symposium on Logic in Computer Science*, July 1988, pp. 165–175, lICS 1988 Test of Time Award. [Online]. Available: https://www.microsoft.com/en-us/research/publication/the-existence-of-refinement-mappings/

[17] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, p. 463–492, jul 1990. [Online]. Available: https://doi.org/10.1145/78969.78972

[18] M. Azure, "Azure/azure-cosmos-tla: Azure cosmos TLA+ specifications," Sep 2018, accessed: 2022-08-17. [Online]. Available: https://github.com/Azure/azure-cosmos-tla

[19] L. Hochstein and M. A. Kuppe, "Reading the Herlihy & Wing linearizability paper with TLA+," https://github.com/lorin/tla-linearizability, 2018, accessed: 2022-08-22.

[20] N. Rivierre, F. Horn, and F. Tran, "On monitoring concurrent systems with TLA: an example," in *Fifth International Conference on Application of Concurrency to System Design (ACSD'05)*, 2005, pp. 36–45.

[21] L. Brutschy, D. Dimitrov, P. Müller, and M. Vechev, "Serializability for eventual consistency: Criterion, analysis, and applications," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 458–472. [Online]. Available: https://doi.org/10.1145/3009837.3009895

[22] N. Crooks, Y. Pu, L. Alvisi, and A. Clement, "Seeing is believing: A client-centric specification of database isolation," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, ser. PODC '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 73–82. [Online]. Available: https://doi.org/10.1145/3087801.3087802

[23] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou, "Modist: Transparent model checking of unmodified distributed systems," in *NSDI'09*, 2009, pp. 213–228.

[24] Microsoft, "Azure chaos studio," 2022, accessed: 2022-08-17. [Online]. Available: https://azure.microsoft.com/en-ca/services/chaos-studio/

[25] P. Deligiannis, N. Ganapathy, A. Lal, and S. Qadeer, "Building reliable cloud services using coyote actors," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 108–121. [Online]. Available: https://doi.org/10.1145/3472883.3486983

[26] Microsoft, "Install and use the Azure Cosmos DB Emulator for local development and testing," https://docs.microsoft.com/en-us/azure/cosmos-db/local-emulator, 2022, accessed: 2022-08-25.

[27] R. Biswas, D. Kakwani, J. Vedurada, C. Enea, and A. Lal, "Monkeydb: Effectively testing correctness under weak isolation levels," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: https://doi.org/10.1145/3485546

[28] K. Kingsbury and P. Alvaro, "Elle: Inferring isolation anomalies from experimental observations," *Proc. VLDB Endow.*, vol. 14, no. 3, p. 268–280, nov 2020. [Online]. Available: https://doi.org/10.14778/3430915.3430918