Going Beyond an Incident Report with TLA$^+$

Finn Hackett, Joshua Rowe, Markus Alexander Kuppe

July 27, 2023

Abstract

Incident reports describe what happened when a system's performance was degraded, or when it stopped working entirely. They document the measurements made, the changes applied in order to rectify the problem, and the contributing factors identified. What they don't do is help us think beyond those facts. Human reasoning capability is limited, and site reliability engineers can benefit from automated reasoning technology just as much as system developers.

We propose that modeling techniques, which are normally used during system design to predict and avoid incidents, are just as valuable after the fact. Modeling allows site reliability engineers to go beyond a prose-and-figures incident reports and produce precise, interactive and analyzable models of incidents. We back up these claims by demonstrating our analysis of a 28-day outage at Microsoft and show the added clarity gained by building a model of what happened. All of the models used in our demonstration are released as open-source.

1 Proposed Workflow

We propose an extension to the traditional incident workflow that incorporates modeling the behaviors involved via a specification language designed for use with concurrent and distributed systems. In many cases, this can lead to greater understanding of the factors that contributed to the incident, as well as providing a mechanism to explore the effectiveness of proposed strategies for preventing similar incidents from occurring in the future.

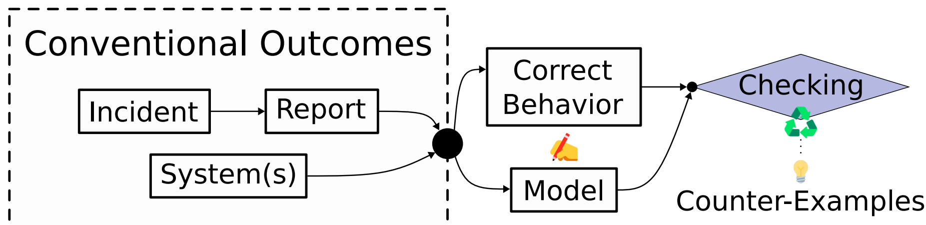

Figure 1: Proposed Additions to Incident Report Workflow

In Figure 1, we visualize how our contributions add to the existing state of the art. Considering an incident that has already been mitigated, the engineers who responded to the incident will already have collected a large amount of information regarding what happened and what was done. This is the "conventional outcome", in which case all insight will be described using prose and diagrams. The problem with prose is that it can be hard to interpret after the fact, and it can hide subtle logical and factual inconsistencies. To make things more precise, once the facts are established, engineers should also produce a machine-analyzable model representing the systems involved and the configuration that lead to the incident scenario.

Thinking more broadly, if an incident report contains a machine-analyzable representation of what happened and why, its value increases significantly. Beyond recording the facts and the ideas of those involved, such a report can be used to predict future incidents and analyze potential after-effects of the incident's mitigation. Many events observed in large-scale production environments may not be reproducible under test. Even using chaos engineering [1], only less-common behaviors can be explored with some probability, as opposed to exploring an exhaustive list of possible events. Tools do exist that help developers reason about unusual behaviors their code might exhibit [2, 4], but they have to stochastically sample a subset of possible implementation-level events rather than produce a comprehensive summary. As a result, a light-weight machine-analyzable representation of all of a system's allowable behaviors is an important asset when reasoning about production incidents.

Note that while we use TLA^+ (modeling language based on the Temporal Logic of Actions) [13] and TLC (model checker) [16], the principles we discuss apply to any reasoning tool with similar capabilities. For example, the Alloy [11], Promela [8], and P [5] modeling languages cover a related design space.

2 Motivating Incident

Our motivating incident is a 28-day outage at Microsoft Azure. The outage involved two system-specific communicating services: a work dispatcher and one or more workers that perform some unspecified task. The outage also involved two core Azure services: Azure Service Bus and Azure Cosmos DB. Azure Service Bus provides a First-In-First-Out queue with which services may communicate, and Cosmos DB is a distributed key-value store that can store arbitrary application data.

Figure 2 illustrates the key elements of the problem scenario as it was originally presented in an incident report. For each request serviced by the work dispatcher, it would (1) write some metadata to Cosmos DB, and then it would (2) enqueue the ID of the metadata it just wrote and a request to do work onto the Azure Service Bus. At the receiving end of the service bus, some worker service would (3) dequeue the work request including the ID of relevant metadata. The worker would then (4) look up that metadata in Cosmos DB and fail to find it. Item (4) is the only anomalous step: the intended scenario is that

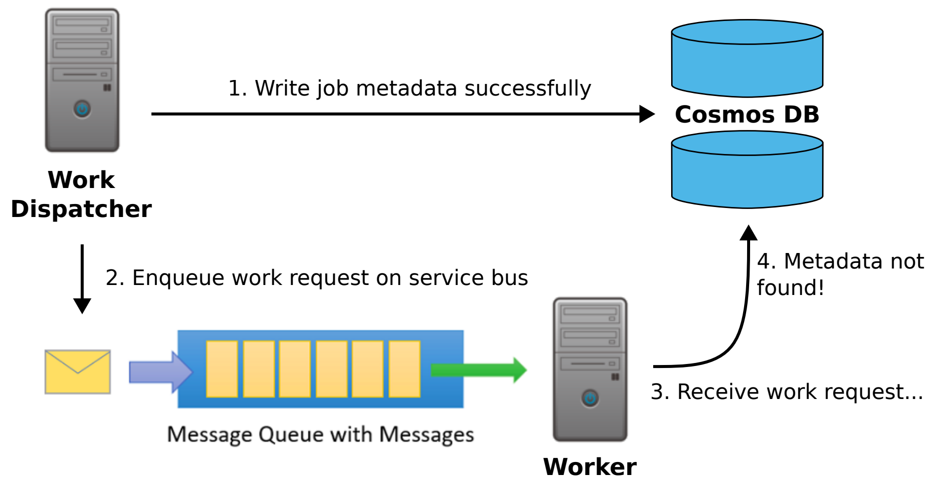

Figure 2: Flowchart visualization of our motivating problem scenario.

the worker successfully reads the metadata it needs and gets to work. Note that a majority of requests did not fail in this way – rather, a specific type of request was disproportionately affected, such that global error rates did not reveal the outage despite a specific group of users being impacted.

The original incident report identified the problem scenario above and included measurements indicating that a reduction in latency between (1) and (3) was the main contributing factor leading to the outage. The resolution, therefore, was to revert a set of performance improvements that reduced that latency.

The outcome was satisfactory in that the customers that experienced the outage were no longer measurably affected, but at the design level the insight was unsatisfactory. While a careful reader might intuit that there was some data consistency problem, the incident report lacked key details needed to understand the issue in depth. Without a clear picture of what was going wrong or how to fix it beyond leaving the existing feature revert in place, the only way forward became launching a multi-year redesign of the entire service. Including a model of the incident alongside the original report would have allowed developers to build a clearer picture of the issue, as well as evaluate the correctness of smaller and more targeted design changes that might also have addressed the underlying issue. Compared to what actually happened, we believe our modeling techniques would have saved significant time and effort.

In the following sections, we describe our experience building such a model for this incident. We also identify a design-level change that could fundamentally prevent the outage from re-occurring, even with the previously-reverted performance improvements.

3 Deciding What to Model

When representing an incident, the goal is not to model everything, or to represent what is modeled completely. Rather, the goal is to identify and represent the set of factors that contributed to the incident happening, such that the resulting model provides a useful abstraction over the problem domain. In this example, we focus on modeling Cosmos DB's consistency guarantees and the interactions with the system which experienced the incident. It might be possible to broaden the inquiry, like addressing the many other reasons a system might end up in a state where workers are unable to read data for dequeued jobs, but generality is not always better. An over-general model can be harder to understand, and it may be harder to see that the model does not represent reality. This is similar to general-purpose strategies like a reprocessing mechanism that retries on failure. They can make a system more robust, but might also cause outages by unexpectedly interfering with system load [9]. With modeling, we think it is more reliable to start with specific concerns like the one we address here, and build up the insight needed to think about more general issues over time.

For our motivating incident, we chose to represent the situation as we summarized it in Figure 2. We modeled one work dispatcher, one worker, a message queue, and an instance of Cosmos DB. There is no need to model the system end-to-end, since the incident report already indicates which parts of the system would be interesting to model We don't need to model what the work is, how the expected network communication happens, or any particular details of where the dispatcher gets its instructions. If the worker can gather the necessary metadata, then the issue has not occurred, and further events are not of interest. Similarly, all signs indicate that the message queue was working as designed: work was successfully enqueued at one end and then dequeued at the other. Without further evidence of it doing anything controversial, we chose to model it as simply as we could.

Cosmos DB, on the other hand, was the service whose behavior was directly involved in the problem outputs. That is not to say that it had a bug – it was in fact behaving as designed. This issue is an example of a cross-system interaction [15], and the underlying issue could be categorized similarly to a grey failure [10]. The fact that it was behaving in a way that was not fully explained meant that we needed to model its behavior in more detail, because an over-simplified representation of its expected behavior would simply not exhibit the issue. On the other hand, an overly detailed model would just complicate the incident model without increasing its usefulness.

With sufficient care, small ad-hoc abstractions can be used to summarize the relevant behaviors of target systems, which can be much easier to build than a complete model. That said, the ideal scenario is that the vendors of a given system would already provide a ready-to-use model of its expected behavior. A generic model authored by a system's developer team could be better trusted to represent real system behaviors, reducing the need for ad-hoc modeling of a given system. Of course, the usefulness and trustworthiness of any model will always be relative; we are talking about a matter of practical utility and

Consistency Level	Effect on Client View
Strong Consistency	Global order; clients always see latest versions
Bounded Staleness	Old values visible for limited (bounded) time
Session Consistency	Order only between clients with shared token
Consistent Prefix	Generally similar to Eventual Consistency
Eventual Consistency	Old values should eventually stop showing up

Table 1: Summary of data consistency levels supported by Azure Cosmos DB

reducing manual effort. Optimistically, the growing use of formal methods in practical system development [3, 6, 12, 14] points toward this concept becoming more realistic over time.

By using a generic model of Cosmos DB's behavior that we had previously built [7], and having sufficiently simplified our representation of all the other necessary components, it took us about 1 person day to recreate the interaction shown in Figure 2.

4 Need-to-know About Azure Cosmos DB

This section covers details of Cosmos DB's design that are necessary to understand our model. Cosmos DB is a distributed service that replicates stored data across multiple servers and geographical regions in order to withstand incidents like machine failure, network failure, datacenter failure, and so forth. Since clients may connect to any Cosmos DB server at any time, servers must maintain consensus regarding which version of a given value a client should be shown.

Cosmos DB supports 5 configurable "levels" of consistency. Table 1 provides a summary of how each level works. Generally speaking, the higher up the table, the less efficient and more predictable the behavior. The top of the table provides a globally consistent view of the stored data, but it is considered a heavy-weight setting. Similarly, the lower down the table, the better the performance and the less predictable the behavior. The bottom of the table provides no guarantees other than clients eventually seeing more recently-written data.

Of interest to our model is "session consistency", which is a recommended default setting. This setting operates via tokens, which are opaque identifiers that indicate a "session". This was the mode used in the incident scenario. Clients may share tokens and connect to Cosmos DB while identifying themselves using a shared token. If a client connects without a token, it will be issued an arbitrary one. Session consistency has two possible behaviors: if two clients identify themselves with the same token, then they will see each other's written data as if operating under strong consistency. If two clients do not have the same token, then no guarantees are offered and they will see each other's behavior as if operating under eventual consistency. In the incident in question, this detail was fundamental to the bug observed: due to incorrect token handling, the system saw eventual consistency guarantees rather than the expected strong consistency-like guarantees.

5 Model and Counter-Example

Listing 1 contains the majority of the model definitions needed to reproduce our motivating incident, with some boilerplate omitted for presentation. It is written using the imperative PlusCal syntax for TLA$^+$, which can be more intuitive to read for those used to popular programming languages. To explore the model yourself, check it out from Github at `https://github.com/tlaplus/azure-cosmos-tla/tree/master/simple-model`. All necessary data and configuration are provided; opening the folder with the TLA$^+$ Visual Studio Code plugin (`https://github.com/tlaplus/vscode-tlaplus`) and selecting the `TLA+: Check Model With TLC` option is sufficient to view the counter-example we include in this article.

This model has two explicit processes, the dispatcher and the worker, and one implicit one, the Cosmos DB service.

The two explicit processes are defined by `process` definitions, which each define a process that operates asynchronously to other processes by taking a sequence of atomic steps. Each atomic step is labeled, which shares a syntax with the low-level control flow construct of the same name. For example, `dispatcherWriteTaskDataInit:` is the first label of the `dispatcher` process. Sometimes, a step must wait for a condition to be true before being able to run. In that case, the `await` statement is used to ensure that the current step cannot execute until the relevant condition, such as `serviceBus # <<>>` (which reads "the service bus is not empty"), is true.

The Cosmos DB service is defined separately as a re-usable module, and its operations are referenced with the prefix `CosmosDB!`. `WriteInit(key, value)`, `WriteInitToken`, and `WriteCanSucceed(token)` operate in combination, starting and completing a write operation to Cosmos DB. The first two operations begin a write and extract its identifier, `WriteInitToken`. The third is a predicate that is only satisfied if the write has succeeded (if the write fails, it will never be satisfied). `SessionConsistencyRead(token, key)` yields the set of values that may be read in the current system state given `token` and `key`. Depending on possible valid Cosmos DB internal states at any given time, the set may be empty or may contain multiple possible values. `UpdateTokenFromRead(token, read)` combines an existing session token with a chosen result of a read operation to produce a new, updated session token.

Note that the conspicuously absent service bus has been simplified to the extent that it is not represented as a process at all: its state is summarized by the global variable `serviceBus`, which is a plain TLA$^+$ sequence. It starts as an empty sequence, which has the syntax `<<>>`. Sending to and receiving from the service bus is performed inline by the dispatcher and worker.

Overall, the model operates exactly as Figure 2 shows, albeit without any assumption that step (4) will fail. Instead, the worker reads from the database and records the outcome in the local state variables `workerToken` and `workerValue`. While these state variables are not functionally necessary, they can be used to examine the model's behavior: we can write properties describing how their values should change over time, and we can model check whether these properties

```
variables serviceBus = <<>>; \* defined as empty sequence

\* dispatcher process
process (dispatcher = "dispatcher")
variables dispatcherToken;
{
dispatcherWriteTaskDataInit:
    \* begin writing taskKey := taskValue to Cosmos DB
    await CosmosDB!WriteInit("taskKey", "taskValue");
    dispatcherToken := CosmosDB!WriteInitToken;
dispatcherWriteTaskDataCommit:
    \* proceed once write is complete (e.g. wait for ack)
    await CosmosDB!WriteCanSucceed(dispatcherToken);
    serviceBus := <<"taskKey">>;
}

\* worker process
process (worker = "worker")
variables workerToken, workerValue;
{
workerBeginTask:
    await serviceBus # <<>>; \* read service bus
    with(taskKey = Head(serviceBus),
         \* read value from Cosmos DB with no session token
         read \in CosmosDB!SessionConsistencyRead(
                  CosmosDB!NoSessionToken, taskKey)) {
        serviceBus := Tail(serviceBus);
        \* store resulting token and value for analysis
        workerToken :=
            CosmosDB!UpdateTokenFromRead(
                CosmosDB!NoSessionToken, read);
        workerValue := read.value;
    }
}

\* standard Cosmos DB setup boilerplate omitted...
```

Listing 1: TLA$^+$ model of all relevant conditions leading up to the incident.

are universally true.

For our purposes, we want to assert that `<>workerValue = "taskValue"`, or, "eventually `workerValue` will have the value `"taskValue"`". That is, the worker will always read the intended value and not the "404" placeholder. If the property is always true, then the incident we are trying to model cannot happen. Replicating the incident, model-checking shows that the property does not hold for the system as modeled.

```
1: <Initial predicate>
    /\ readIndex = 0
    /\ log = <<>>
    /\ workerValue = defaultInitValue
    /\ pc = [dispatcher |-> "dispatcherWriteTaskDataInit",
             worker |-> "workerBeginTask"]
    /\ serviceBus = <<>>
    /\ commitIndex = 0
    /\ dispatcherToken = defaultInitValue
    /\ workerToken = defaultInitValue
2: dispatcherWriteTaskDataInit
    /\ log = <<[value |-> "taskValue", key |-> "taskKey"]>>
    /\ pc = [dispatcher |-> "dispatcherWriteTaskDataCommit",
             worker |-> "workerBeginTask"]
    /\ dispatcherToken = [epoch |-> 1, checkpoint |-> 1]
3: cosmos
    /\ commitIndex = 1
4: dispatcherWriteTaskDataCommit
    /\ pc = [dispatcher |-> "Done",
             worker |-> "workerBeginTask"]
    /\ serviceBus = <<"taskKey">>
5: workerBeginTask
    /\ workerValue = "404"
    /\ pc = [dispatcher |-> "Done", worker |-> "Done"]
    /\ serviceBus = <<>>
    /\ workerToken = [epoch |-> 1, checkpoint |-> 0]
```

Listing 2: Counter-example generated by model-checking the model in Listing 1.

Listing 2 summarizes a counter-example scenario, edited for presentation but illustrative of the ASCII output when TLC finds a counter-example. The counter-example is a chronological series of events expressed as assignments to model state variables. Some of the variables are the same as in Listing 1, while others are defined separately. The Cosmos DB internals are summarized by `readIndex`, `commitIndex`, `epoch`, and `log`, which can completely represent any client-observable state Cosmos DB might reach. The `pc` variable stands for *Program Counter*, and is a proxy for control flow within the model. For the dispatcher and worker, it indicates which action each process should attempt

next, corresponding directly to the labels Listing 1.

Paraphrased from the tool read-out, the counter-example reads as follows:

dispatcherWriteTaskDataInit Begin writing the key-value pair "taskKey", "taskValue". This adds the data to the database, but does not require any consensus-type actions.

Cosmos DB Some partial consensus action. A majority, but not all, Cosmos DB servers will have the new value. This is indicated by commitIndex advancing to point to index 1 of the log, while readIndex stays at 0.

dispatcherWriteTaskDataCommit The write process is considered complete. The dispatcher sends "taskKey" via the service bus.

workerBeginTask The worker receives "taskKey" via the service bus, and reads that key from Cosmos DB. It stores the value it read in workerValue, which is "NoValue".

The counter-example makes clear a logical problem with how the worker and dispatcher behave. Speaking intuitively, because Cosmos DB is operating in session consistency mode, if they do not share a session token then the clients are opting out of any consistency between reads and writes. It is entirely possible that the worker accesses a Cosmos DB server that has not yet received the value written by the dispatcher, and without a session token that server will just respond with the information it has on hand: "taskKey" does not refer to a value. With an up to date session token, the server would see evidence encoded into the token itself that its state is stale, and it would have to wait before responding.

To further demonstrate that the lack of forwarding session tokens caused the read failures, editing the dispatcher and worker as shown in Listing 3 to send a session token via the service bus changes the model checking outcome. If the two processes share a session token, then the model checker can no longer find any counter-examples to the property <>workerValue = "taskValue".

Looking at this explanation, it also becomes clear why the issue could not be found using conventional testing, nor could it easily be reproduced at the implementation level. The issue was of similar complexity to a grey failure [10], being rare and load-dependent, meaning that any implementation-level reproduction would have to reliably recreate those complex and difficult to control factors. That is why incident reports are usually prose and figures, because without an alternative strategy like modeling, it is simply impractical to do anything else. With the addition of high-level modeling however, creating a demo of this kind of rare issue becomes possible.

6 Take-Aways

Combined with the original incident report, this model made it possible to clearly explain the design-level problem. It also helped validate the correctness of a proposed fix that could fundamentally prevent the issue going forward.

```
    dispatcherWriteTaskDataCommit:
        \* proceed once write is complete (e.g. wait for ack)
        await CosmosDB!WriteCanSucceed(dispatcherToken);
-       serviceBus := <<"taskKey">>;
+       serviceBus := <<[taskKey |-> "taskKey",
+                        token |-> dispatcherToken]>>;
    }
***************
    await serviceBus # <<>>; \* read service bus
-   with(taskKey = Head(serviceBus),
+   with(msg = Head(serviceBus),
         \* read value from Cosmos DB with no session token
         read \in CosmosDB!SessionConsistencyRead(
-                   CosmosDB!NoSessionToken, taskKey)) {
+                   msg.token, msg.taskKey)) {
        serviceBus := Tail(serviceBus);
        \* store resulting token and value for analysis
        workerToken :=
            CosmosDB!UpdateTokenFromRead(
-               CosmosDB!NoSessionToken, read);
+               msg.token, read);
        workerValue := read.value;
```

Listing 3: Changes to Listing 1 needed to address the underlying design issue.

The key benefit of this style of modeling to incident reports is the ability to consider all possible systems states in aggregate. The model we developed does not only represent the error scenario: it shows how the error is possible, while also representing many situations where the error could coincidentally not happen. That is, checking the property `[]workerValue # "taskValue"`, that `workerValue` will *never* have `"taskValue"`, will also yield counter-examples. Cosmos DB is also allowed to propagate `"taskValue"` to the worker correctly; it just isn't required to under session consistency with no shared tokens.

As a result, the style of model for which we are advocating is strictly more expressive than devices for illustrating one or a small set of possible events. Figure 2 on its own, for example, is incomplete. A model like the one we show can be used to summarize millions of possible events in relatively little text, while allowing humans to query all of those possibilities in aggregate during design discussions.

Anecdotally, when we validated our work with the owner of the incident report we were using, they were pleasantly surprised that we could demonstrate the issue so cleanly. They were not surprised what the issue was, however. It is often the case that modeling does not expose strictly new information. Rather, it makes clear which suspicions developers and operators already have might actually be the case, converting abstract hypotheses into practical demos.

In summary, we have presented an extension to the conventional incident report workflow: formal modeling. We were able to derive greater insight into a complex high-impact incident, and were able to suggest a fundamental fix to the underlying bug. Our method is light-weight, and, with the right training in modeling and pre-existing tooling, can be followed in about 1 person day of effort. We hope to inspire adoption of this method as part of incident report writing at Microsoft and beyond. This hope is in part based on successful adoption of both TLA^+ specifically and other automated reasoning tools in industry. While incident reports can often be seen as special cases or single examples for which automated reasoning is unnecessary, the complexity and subtlety of cross-system or grey failures at scale merits more automated, systematic analysis.

To learn more about the tools we used, you can visit `http://tlapl.us`. To examine the model we developed, it is open-source at `https://github.com/tlaplus/azure-cosmos-tla/tree/master/simple-model`.

References

[1] Ali Basiri et al. "Chaos Engineering". In: *IEEE Software* 33.3 (2016), pp. 35–41. DOI: 10.1109/MS.2016.60.

[2] Ranadeep Biswas et al. "MonkeyDB: Effectively Testing Correctness under Weak Isolation Levels". In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485546. URL: https://doi.org/10.1145/3485546.

[3] James Bornholt et al. "Using lightweight formal methods to validate a key-value storage node in Amazon S3". In: *SOSP 2021*. 2021. URL: https://www.amazon.science/publications/using-lightweight-formal-methods-to-validate-a-key-value-storage-node-in-amazon-s3.

[4] Pantazis Deligiannis et al. "Building Reliable Cloud Services Using Coyote Actors". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '21. Seattle, WA, USA: Association for Computing Machinery, 2021, pp. 108–121. ISBN: 9781450386388. DOI: 10.1145/3472883.3486983. URL: https://doi.org/10.1145/3472883.3486983.

[5] Ankush Desai et al. "P: Safe Asynchronous Event-Driven Programming". In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2013.

[6] Liang Gu et al. "CertiKOS: A Certified Kernel for Secure Cloud Computing". In: *Proceedings of the Second Asia-Pacific Workshop on Systems*. APSys '11. Shanghai, China: Association for Computing Machinery, 2011. ISBN: 9781450311793. DOI: 10.1145/2103799.2103803. URL: https://doi.org/10.1145/2103799.2103803.

[7] Finn Hackett, Joshua Rowe, and Markus Alexander Kuppe. "Understanding Inconsistency in Azure Cosmos DB with TLA+". In: *Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '23. Melbourne, Australia: The Institute of Electrical and Electronics Engineers, 2023, pp. 1–12. ISBN: 978-1-6654-5701-9. DOI: 10.1109/ICSE-SEIP58684.2023.00006. URL: https://doi.org/10.1109/ICSE-SEIP58684.2023.00006.

[8] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*. Vol. 1003. Addison-wesley Reading, 2004.

[9] Lexiang Huang et al. "Metastable Failures in the Wild". In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. Carlsbad, CA: USENIX Association, July 2022, pp. 73–90. ISBN: 978-1-939133-28-1. URL: https://www.usenix.org/conference/osdi22/presentation/huang-lexiang.

[10] Peng Huang et al. "Gray Failure: The Achilles' Heel of Cloud-Scale Systems". In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS '17. Whistler, BC, Canada: Association for Computing Machinery, 2017, pp. 150–155. ISBN: 9781450350686. DOI: 10.1145/3102980.3103005. URL: https://doi.org/10.1145/3102980.3103005.

[11] Daniel Jackson. "Alloy: A Lightweight Object Modelling Notation". In: *ACM Trans. Softw. Eng. Methodol.* 11.2 (Apr. 2002), pp. 256–290. ISSN: 1049-331X. DOI: 10.1145/505145.505149. URL: https://doi.org/10.1145/505145.505149.

[12] Gerwin Klein et al. "SeL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 207–220. ISBN: 9781605587523. DOI: 10.1145/1629575.1629596. URL: https://doi.org/10.1145/1629575.1629596.

[13] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 032114306X.

[14] Xavier Leroy et al. "CompCert - A Formally Verified Optimizing Compiler". In: *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*. SEE. Toulouse, France, Jan. 2016. URL: https://hal.inria.fr/hal-01238879.

[15] Lilia Tang et al. "Fail through the Cracks: Cross-System Interaction Failures in Modern Cloud Systems". In: *Proceedings of the Eighteenth European Conference on Computer Systems*. EuroSys '23. Rome, Italy: Association for Computing Machinery, 2023, pp. 433–451. ISBN: 9781450394871. DOI: 10.1145/3552326.3587448. URL: https://doi.org/10.1145/3552326.3587448.

[16] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. "Model Checking TLA+ Specifications". In: *Correct Hardware Design and Verification Methods*. Ed. by Laurence Pierre and Thomas Kropf. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 54–66. ISBN: 978-3-540-48153-9.