Lightweight methods as a new trend in the system development life cycle

Andriy Manucharyan
Technical University Munich

Hüseyin Kuzu
Technical University Munich

Abstract

This paper and its research ideas are based on, the SOPS 2021 best-paper award for work on automated reasoning *Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3* [3] written by *James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen and, Andrew Warfield.* In this paper we will take their insights as the basis and use this gained knowledge in similar likely problems.

1 Introduction

Formal Methods are commonly used in the verification of software. Ensuring the correctness of software is increasingly getting harder, due to the rapidly increasing complexity of systems. Therefore there is a huge need for alternative solutions. Researchers are developing tools and techniques to overcome this issue. Ensuring that a system does what it is pretending is crucial. The latest research has shown that using lightweight formal methods to validate, in this use-case, a storage system, can be used at the production scale. However, using Lightweight-Formal-Methods does not guarantee full formal verification. It can help to automate this process and decrease the need for formal methods.

During our research, we found out that the use of formal techniques such as formal specification depends on the project and its goal. We collected shared experiences from multiple research experts in the last 30 years and made conclusions which might be useful for future verifications.

In the following, we will outline the technical overview and the contributions to this problem.

In summary, this paper makes two contributions:

- Summary of the paper focusing on the technical overview, the validation of the storage system ShardStore and lastly the conclusion and insights.
- Evaluating the usability of Lightweight-Formal-Methods in a modern system development life cycle.

2 Summary

This section will provide background on the paper *Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3*[3], focusing on ShardStore and the Validation.

2.1 ShardStore

ShardStore is a key-value storage used by Amazon in their AWS S3 services. Sharding is a database-architectural pattern, in which a database is horizontally split into multiple smaller ones. This technique brings many advantages like faster horizontal scaling, faster query handling, and a more reliable database which then leads to higher availability. Each storage node stores multiple shards, of data. Shards are fragments of stored data, which are replicated across multiple storage nodes. Each shard has a universally unique ID (UUID), which is the identifier key and the stored customer data is the value in the key-value store. Logarithmic-Structured-Merge-Tree (LSM-Tree) is a commonly used data structure in many data store management systems. It is structured as a key-value store. The key, here the shards-ID, and the value, here the pointers of the shards pointing at the chunk written in the extent where the data is stored in the disk. Chunks are physical disks dedicated to a database server data storage. The LSM-Tree is also stored in such chunks. The data structure is a disk-based one, to provide low-cost indexing. [7]

2.2 Validation

With the recent success in the storage system verification, the constantly changing source code, and the increasing complexity of the formal verification, the researchers wanted to apply lightweight formal methods to ShardStore. Their main target is finding new lightweight formal methods to validate a storage system and also advocate the automation of the validation, increasing the usability, and lastly ensuring the correctness of the system. In the following, we will focus on those three properties:

- Checking the equivalence for sequential crash-free executions (2.2.1).
- Checking for a corresponding weaker equivalence for sequential crashing executions (2.2.2).
- Checking the linearizability for concurrent crash-free executions (2.2.3).

2.2.1 Conformance Checking

Using property-based testing is a common technique to check the conformance of a system. It is used for functional correctness and is designed to test the aspect of a property that should always be true in this case the reference model, which we can guarantee that the property is correct. We check the implementation with the reference model and compare the states after random sequences. With such testing, we can assure the validation. This technique can miss some bugs, so that is why this approach needed a higher scale of testing to cover as many as possible bugs. Besides correctness testing, property-bases-testing can also be used in correct failure handling.

2.2.2 Crash Consistency Checking

Validating and ensuring crash consistency is one of the main challenges of ShardStore. For reference two crash consistency properties were defined:

- Persistence: This means if an operation is persistent before a crash, it is also after it.
- Forward Progress: This means after a reboot, every operation's dependency should be persistent.

To check, if the implementation satisfies these two crash consistency properties, the property-based-testing (2.2.1) is extended with the crash-state-checking.

For this type of testing the Dependency and also the Reference-Model are used. With the Dependency-Graph we can compare ShardStore and those operations if they have the same state. The Reference Model, in this case, a Hash-Map, have the same input data. Comparing the Reference Model with the ShardStore after crash-state errors, crash consistency can be detected.

2.2.3 Concurrent Executions Checking

Till now the tests only covered single-threaded and sequential operations. Amazon S3 is a highly concurrent service, and therefore ShardStore also has to be highly concurrent, to be used in production. Testing and debugging concurrency is too difficult, due to the non-determinism. To cover such types of testing, stateless-model-checker tools like Loom are used. Stateless-Model-Checkers can validate concurrency properties, like consistency, and also test for deadlocks. It uses an in-memory model. This leads to huge drawbacks, the scalability and therefore linearizability. For that reason, SMCs are only used for checking for correctness critical code. To overcome this, the researchers have developed an open-sourced Stateless-Model-Checker called Shuttle. Shuttle implements randomized algorithms such as probabilistic concurrency testing to cover a larger test harness and check also the end-to-end tests.

2.3 Conclusion of the paper

Summing up this part, the researchers have made great progress using lightweight formal methods and enhancing the verification by applying stronger verification techniques. In the beginning, two goals were set: Detecting bugs as early as possible in the development stage, and secondly ensuring that the validation work remains relevant as the code changes over time.

3 Research Ideas

3.1 Our Focus

After reading and analyzing the paper, we decided to make some research progress in the following directions:

- Research what are the differences and similarities between lightweight formal methods and classical ones
- Discuss what are the criteria for choosing different techniques
- Take a closer look at an open-source project RocksDB and analyze application of formal methods.

3.2 Our Approach

We can roughly divide our approach into 3 parts:

1) Read paper[3] and references to collect information about lightweight formal methods

2) Define advantages and constraints of techniques that were used during validation of different systems

3) Make conclusions and analyze the future of formal methods

3.3 Novelty of "lightweightness"

Software development is changing very frequently. Having more and more complex systems the costs for development are increasing while verification is still needed to prove the robustness, and correctness of the system. It is important to point out as papers researchers mentioned that depending on the project it is sensible to sacrifice full formal verification in favour of making the approach easier and less time-consuming as these are one of the important aspects of software development.

As IEEE Computer Society members pointed out already in 1998 [4], even while having volatile requirements it is still worthwhile to use lightweight methods as the formal model itself can be discarded if the requirements change significantly, while the experience and lessons learned from it are retained. Interactions between changing requirements can be hard to identify, thus formal methods help to challenge the system by defining properties that should hold.

Daniel Jackson and Jeannette Wing [5] stated also more than 20 years ago: "The problem with testing is not that it cannot show the absence of bugs, but that it fails to show their presence. A model checker that exhausts an enormous state space finds bugs much more reliably than conventional testing techniques, which sample only a minute proportion of cases."

One of the clear drawbacks of lightweight formal methods [3] , is for example the fact that property-based testing selects random sequences of operations to test. This means not all bugs by far can be found. One can still prevent the majority of the bugs with its help, but assuming that the system runs as expected is insufficient. This forces them to combine different techniques to minimize the risk.

This means the software/system development team should decide whether the advantages of lightweight formal methods outweigh their disadvantages before actually using them.

3.4 Choosing techniques

It is generally difficult to decide which concrete techniques should be used depending on the system. However, in some cases, there are clear drawbacks. For example, Amazon researchers had the following issue while using the formal specification for validation of S3 [6]. They defined two classes of problems with systems:

1) Real bugs, behaviour which doesn't match expectations

2) Emergent performance degradation

According to their paper, it is possible to find and solve most of the problems of the first class, but difficult to specify and cover all the scenarios which could cause problems in the second class. One possible example would be suddenly increasing response time which would make the user have to retry requests although no functional bugs exist. Of course, different specification languages (e.g. TLA+), as well as in-built tools, allow to specify for instance upper bound on response time, but in this concrete case with S3 the infrastructure didn't allow to realistically guarantee the defined behaviour of the system.

3.5 RocksDB testing environment

To have a real example of lightweight formal methods being applied we decided to take a closer look at RocksDB [1] implementation which uses some of the techniques mentioned above. While LevelDB does not support ranged queries, RocksDB does which makes it more interesting when trying to find bugs. Developers use LLVM libFuzzer [2] as a fuzzing engine as well as protobuf and libprotobuf-mutator are needed to be able to define and mutate the input to the fuzzer itself. Protocol buffers are a data format which involves interface descriptive language in order to communicate between programs over a network for storing data. The way fuzzing of RocksDB is manipulated before running can be divided into 3 parts:

1) Design the test

2) Define input

3) Define tests with the input

In the first part, the idea is to generate a random sequence of database operations. The process is very similar to property-based testing of Amazon S3. It is also possible to provide structured and user-defined input which allows for checking richer behaviour than classical fuzzing. In the second part, the types of operations are defined with the help of protobuf as shown in Fig. 1. In the third part, we have to ensure that SST (Sorted String Table) File Writer writes only unique keys in the ascending order which is handled in the post-processing method. SST File Reader verifies the checksum of the file after the SST File writer goes through all operations defined in step 1. It would be great to introduce a bug manually and see whether fuzzer can catch it, but after multiple tries, we, unfortunately, didn't manage to link all the downloaded libraries/tools. However, according to the RocksDB team and an example provided by them the fuzzer was able to find a bug after 6 seconds on the case where the introduced bug involved the corruption when the key started with '!' and the value ended with '!'. We are not going to go through the unit tests which are also part of the validation because the experience collected here is barely new.

```
syntax = "proto2";

enum OpType {
  PUT = 0;
  MERGE = 1;
  DELETE = 2;
  DELETE_RANGE = 3;
}

message DBOperation {
  required string key = 1;
  // value is ignored for DELETE.
  // [key, value] is the range
  // for DELETE_RANGE.
  optional string value = 2;
  required OpType type = 3;
}

message DBOperations {
  repeated DBOperation operations = 1;
}
```

Figure 1: .proto file defining types of operations and a key value pair

3.6 Conclusion

Lightweight formal methods offer several advantages over traditional, heavyweight formal methods. They tend to be less expensive and time-consuming to use, and they are often easier to learn and use. Additionally, lightweight formal methods can be used for a wider range of tasks than heavyweight formal methods, making them more versatile.

Despite these advantages, there are some potential disadvantages to using lightweight formal methods. One is that they may not be as rigorous as heavyweight formal methods, they lack power in expression and breadth of coverage. Overall, lightweight formal methods are a useful tool for software development since the main idea behind them is to maximize the number of bugs being found in the early stage, but it is important to weigh the advantages and disadvantages carefully before deciding to use them.

References

[1] https://github.com/facebook/rocksdb.

[2] https://llvm.org/docs/libfuzzer.html.

[3] Bornholt, J., Joshi, R., Astrauskas, V., Cully, B., Kragl, B., Markle, S., Sauri, K., Schleit, D., Slatton, G., Tasiran, S., Van Geffen, J., and Warfield, A. Using lightweight formal methods to validate a key-value storage node in amazon s3. *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21) October 26-28, 2021*, 28 (2021), 836–850.

[4] Easterbrook, S., Lutz, R., Covington, R., Kelly, J., Ampo, Y., and Hamilton, D. Experiences using lightweight formal methods for requirements modeling. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 24, NO. 1* (1998), 4–14.

[5] Jackson, D., and Wing, J. Lightweight formal methods. *ACM Computing Surveys 1996* (1996), 21–22.

[6] Newcombe, C., Rath, T., Zhang, F., Munteanu, B., Brooker, M., and Deardeuff, M. How amazon web services uses formal methods. *Communications of the ACM, April 2015, Vol. 58 No. 4* (2015), 66–73.

[7] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E. The log-structured merge-tree (lsm-tree). *Acta Informatica 33 4 (June 1996)* (1996), 351–385.