The Polyglot's Dilemma: Conformance Testing a Dozen Specs in as Many Languages

A. Jesse Jiryu Davis
MongoDB Research
New York, New York, USA
jesse@mongodb.com

Jeremy Mikola
Independent
Hoboken, New Jersey, USA
jmikola@gmail.com

Jeff Yemin
MongoDB, Inc.
New York, New York, USA
jeff.yemin@mongodb.com

***Abstract*—MongoDB maintains client libraries in a dozen programming languages, used by tens of thousands of organizations and millions of developers. Most are implemented natively rather than as wrappers around a shared core. Ensuring consistent behavior across these libraries, comprising millions of lines of code, is hard but essential. Over eleven years, we developed a specification-based testing approach: tests are written once in YAML and executed by language-specific interpreters for each library. We describe the evolution from many ad-hoc formats to a Unified Test Format, which allowed us to delete over 22,000 lines of test code. The rate of nonconformance bugs fell up to 86% in drivers that adopted YAML tests (though results varied). We report lessons learned about declarative test design, test architecture, schema evolution, and the limits of unification.**

***Index Terms*—NoSQL, MongoDB, protocol testing, conformance testing**

I. Introduction

Database client libraries are often built as thin wrappers around a shared core written in C. This architecture minimizes duplication, but shared-core libraries impose costs on users. Compiling and installing a C dependency is unfamiliar for many programmers, the library cannot participate in language-native async event loops, and its API rarely feels idiomatic.

At MongoDB, we chose a different path. We maintain a dozen client libraries (the number has changed over time), which we call "drivers." We wrote the entire driver logic and API nine times in C, C#, Go, Java, Node.js, Python, Ruby, Rust, and Swift. In addition, we wrote idiomatic layers in C++ and PHP that wrap the C Driver, and in Scala and Kotlin wrapping the Java Driver. This approach lets each driver integrate naturally with its ecosystem—it is easy to install, obeys native logging conventions, cooperates with async event loops, and presents an API that feels familiar to developers in that language. The cost is that we maintain the same client-server protocol, connection pooling, server discovery, and other behaviors in at least nine code bases.

Prose specifications help, but natural language is ambiguous. Differential testing [1] can detect how implementations diverge, but it cannot say which implementation is correct. It also cannot detect when all implementers make the same mistake—so-called common-mode failures—because the spec is unclear or because some errors are easy to make [2].

Our solution is a *domain-specific test language*, or DSTL: we write machine-executable tests in YAML that encode the spec author's intent. Each driver implements a test runner that interprets these YAML files, runs the specified operations against a MongoDB deployment, and asserts that the results match expectations. When specs change, we update the YAML tests in a central repository, and the changes propagate automatically to the individual drivers.

Starting in 2015, we developed and refined this approach. Our early efforts produced multiple test formats, each tailored to a particular specification: one DSTL for create, read, update, delete (CRUD) operations, another for transactions, another for connection pooling, and so on. By 2020, this proliferation had become a maintenance burden. We consolidated these formats into a Unified Test Format (UTF), validated by a JSON Schema. By migrating to UTF we deleted over 22,000 lines of redundant test-runner code, while gaining schema validation and a more expressive test language.

UTF introduced several mechanisms that we will explain in detail below. An *entity map* is a registry of objects—clients, databases, collections (i.e. database tables), cursors—created during a test, allowing operations to reference them by name later in the test. *Command monitoring* captures every message the driver sends to the server and every reply it receives, enabling tests to check wire-level behavior. *Fail points* are server-side hooks that simulate errors, dropped connections, and slow responses, providing controlled conditions for testing error handling.

This paper reports on eleven years of industrial experience applying this methodology across a dozen driver implementations, comprising millions of lines of production code. Our drivers are used by tens of thousands of customers and millions of open source users. Even subtle inconsistencies in their behaviors are costly. While our techniques are not novel compared to recent research prototypes, we offer something complementary: empirical evidence that declarative, cross-language conformance testing scales to production systems over extended timeframes. We report what worked, what did not, and where we reached the limits of unification.

II. Test Architecture and Components

The components for testing a MongoDB driver in any language with UTF are shown in Figure 1. The UTF test

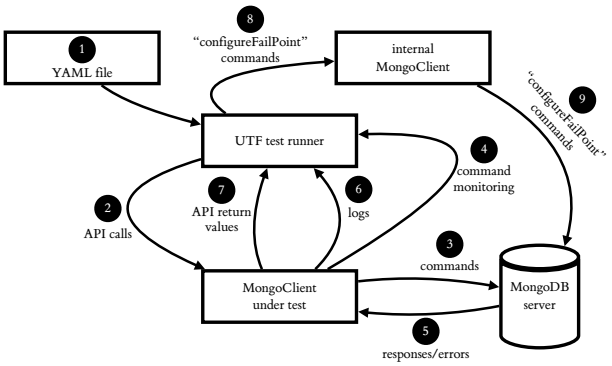

Fig. 1. MongoDB driver specification testing architecture.

runner reads a YAML file (1), and creates two client objects (MongoClients): a MongoClient under test, and an internal MongoClient for out-of-band communications with the MongoDB server. The test runner executes the YAML file's list of operations. For each operation it calls a method on the MongoClient under test (2), which sends commands to the server (3), and also reports them back to the test runner (4) via the command monitoring API. The server responds to the client (5), which may log to a file-like stream created by the test runner (6), and finally the original API call returns a value or error (7). The YAML file's operations may include fail points that the test runner should enable or disable to control the server's behavior; these are interleaved with other test operations. The test runner sends `configureFailPoint` commands to the server via its internal MongoClient to avoid interference with the MongoClient under test. The test runner runs one command at a time, synchronously, on either MongoClient, to ensure tests are deterministic.

We originally designed the command monitoring API for application performance monitoring (APM), but we realized we could use it to test drivers' network messages. Fail points are also vital: we can command the server to misbehave in various ways, e.g., return an error the next *N* times a certain command is called, or block a command for a certain period, or close the connection. Compared to stress testing or "chaos" testing, our approach is fast, deterministic, and simple, while still exercising drivers' error paths. We use APM and fail points to test behavior that is invisible from the API surface, as in the following example.

Figure 2 shows a complete test file in UTF. The test creates a client, database, and collection as named *entities*, referenced by later operations using YAML anchors. The test populates a collection with initial data and tells the server to close the connection on the next `update` command. The runner calls the client's `updateOne` API. The test runner uses APM to check that the driver sends exactly two `update` commands. It uses `waitForEvent` to asynchronously check that the driver eventually marks the server's type as `Unknown` after the network error, meaning the driver will re-check which server is the primary. Without APM and fail points these checks would be inconvenient or unreliable. Finally, `expectResult` asserts that the API call succeeded, and `outcome` confirms the document was updated once.

```
schemaVersion: "1.4"
runOnRequirements:
  - {minServerVersion: "4.4", topologies: [replicaset]}
createEntities:
  - client:
      id: &client client0
      observeEvents:
        - commandStartedEvent
        - poolClearedEvent
        - serverDescriptionChangedEvent
  - database:
      id: &db db0
      client: *client
      databaseName: &dbName mydb
  - collection:
      id: &coll coll0
      database: *db
      collectionName: &collName mycoll
initialData:
  - collectionName: *collName
    databaseName: *dbName
    documents: [{_id: 1, x: 1}]
tests:
  - description: "updateOne retries after network error,
                  exactly-once execution"
    operations:
      - name: failPoint
        object: testRunner
        arguments:
          client: *client
          failPoint:
            configureFailPoint: failCommand
            mode: {times: 1}
            data:
              failCommands: [update]
              closeConnection: true
      - name: updateOne
        object: *coll
        arguments:
          filter: {_id: 1}
          update: {$inc: {x: 1}}
        expectResult:
          matchedCount: 1
          modifiedCount: 1
          upsertedCount: 0
      - name: waitForEvent
        object: testRunner
        arguments:
          client: *client
          event:
            serverDescriptionChangedEvent:
              newDescription: {type: Unknown}
          count: 1
    expectEvents:
      - client: *client
        eventType: command
        events:
          - commandStartedEvent: {commandName: update}
          - commandStartedEvent: {commandName: update}
      - client: *client
        eventType: cmap
        events: [{poolClearedEvent: {}}]
    outcome:
      - collectionName: *collName
        databaseName: *dbName
        documents: [{_id: 1, x: 2}] # x incremented once
```

Fig. 2. A UTF test file: entity definitions, initial data, and test operations.

III. Evolution of the Unified Test Format

A. Choosing YAML for Writing Specification Tests

We first introduced cross-driver test files with our Server Discovery and Monitoring (SDAM) specification in 2014. These tests checked drivers' behavior when connecting to a

MongoDB cluster and responding to topology changes (e.g. a change in the number of servers in a *replica set*, MongoDB's consensus group [3]). A test runner fed simulated server messages to a driver's internal state machine and asserted on its state transitions.

We considered the Cucumber testing framework [4], with its English-like Gherkin syntax. But Gherkin is intended to express business logic to non-programmers. Describing binary formats and TCP/IP connections was inconvenient and silly-looking; our engineers rebelled at the proposal. We settled on YAML for its programmer-friendly syntax, support for comments, and the availability of parsing libraries in most target languages. YAML allows basic factoring with *anchors* and *aliases*. Test files could be converted to JSON for languages that lack a YAML parser. The payload of most MongoDB wire protocol messages is BSON, a JSON-like binary format. Since YAML is a superset of JSON, payloads could be naturally embedded in YAML files. In 2017, we developed a MongoDB Extended JSON format that is valid JSON and converts losslessly to and from BSON; e.g., a BSON 64-bit integer is expressed as `{"$numberLong": "42"}`.

These first YAML tests and all the drivers' test runners were specific to the SDAM spec. We began to proliferate YAML test formats for other specs. By 2015 we had various DSTLs and test runners for driver functionality ranging from connection string parsing, to BSON serialization, to server selection. For the CRUD spec, which defined APIs for read and write operations, we introduced a DSTL called CRUD v1, which would be the foundation of UTF.

B. From CRUD v1 to the Unified Test Format

The initial CRUD v1 format allowed a driver to execute one operation and assert either a result or an error, and it could check the final state of the database. In 2015 we specified a *command monitoring* API for drivers, allowing applications to observe every message between the driver and the server. This permitted us to extend CRUD v1 with an `expectEvents` field containing a list of messages and other events to match. In 2017, MongoDB introduced idempotent retry of failed operations. To test this, we began using the server's `configureFailPoint` command systematically in driver testing—our YAML format could instruct the test runner to set a fail point before running a command. Our multiple test formats had become very powerful, but poorly factored and unmaintainable.

UTF debuted in 2020 with a reenvisioned DSTL, backed by a JSON schema and a thoroughly specified test runner. It superseded all previous formats. At first we manually ported a sample of old tests to prove UTF covered all our requirements. Once UTF was formalized, we wrote scripts for each legacy test format to mechanically convert it to UTF.

There are 28 revisions to the UTF schema to date. Old files validate with the latest schema, and the newest test runners can interpret the oldest files. Each UTF file includes a top-level `schemaVersion` field, which specifies the JSON schema version to which the file conforms (when mechanically converted from YAML to JSON). This serves two purposes. First, a test runner can determine at a glance if it is capable of interpreting the file. Second, the CI tests for the specifications repository can validate each test file against its schema.

UTF files include a `runOnRequirements` array containing one or more sets of conjunctive criteria. If at least one set of criteria is satisfied, the test runs:

```
# Txns were first implemented for replica
# sets only, then sharded clusters.
runOnRequirements:
  - { minServerVersion: '4.0',
      topologies: [replicaset] }
  - { minServerVersion: '4.2',
      topologies: [sharded] }
```

The UTF runner maintains a general-purpose object registry: the entity map. Entities are defined in the `createEntities` section and referenced by name. The `saveResultAsEntity` syntax saves a command result to the entity map for later use. This permits us to create and reuse any number of collections, documents, etc. during the test.

UTF specifies rules for matching logic in great detail, including: numeric comparisons; selectively ignoring optional fields in BSON documents; ignoring key order when comparing documents; and special operators for complex assertions, e.g., asserting whether a field exists, asserting its type, matching against a dynamic value stored in the entity map.

UTF is still evolving. Some syntax has been deprecated, but none has been removed. In addition to command monitoring, drivers support event-based monitoring of server discovery (SDAM) and connection pooling. A forward-thinking design in `expectEvents` required minimal changes when incorporating these new event types. When we added concurrent SDAM testing to UTF, we defined a new "thread" entity type and associated operations (`runOnThread`, `waitOnThread`), reusing the existing `operations` syntax. Testing for standardized logging warranted a new `expectLogMessages` field, closely modeled after `expectEvents`. We used a similar approach for `expectTracingMessages` to test OpenTelemetry integration.

C. New Tests and Specs

The requirements for MongoDB drivers are defined in dozens of specifications. When we create or update a spec and its UTF tests, we determine whether we need new UTF syntax or test runner behavior. If so, we change the spec, UTF, and the test runners simultaneously. We usually prototype spec changes in at least two drivers before general publication.

Historically, MongoDB engineers manually synced YAML tests from the specs repository into drivers' repositories. Beginning in 2024, several drivers transitioned to including the specs repository as a Git submodule. We configured GitHub's Dependabot to automatically create pull requests to update the submodule weekly. If CI reports that the driver's test suite passes with the test changes, the pull request is automatically merged. Since test runners skip files with an unsupported

`schemaVersion`, there is generally no issue with updating test files before a driver can implement the latest specs and test format changes.

D. Testing the UTF Test Runner and Its Tests

The UTF test runner has its own set of tests, which are divided into three categories: "invalid" test files are expected to fail schema validation, "valid-fail" test files should pass schema validation but fail at runtime (e.g., an assertion failure), and "valid-pass" tests should pass without error. Collectively, these tests can assert correctness of the JSON schema. The set of "valid-fail" and "valid-pass" tests can additionally be used to verify correctness of a driver's test runner. Automated checks on GitHub pull requests validate YAML files against their declared JSON schema version, catching mistakes that were common prior to UTF. The individual drivers' test runners also cross-validate the tests: a failure indicates there is a bug *somewhere*, whether in the test, the driver, or the runner.

IV. Results

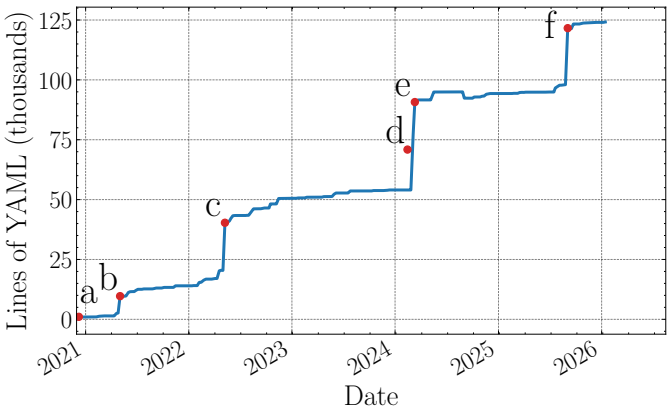

Fig. 3. UTF corpus growth.

Since 2020 our corpus of UTF tests has grown from a prototype with 7 files and 1046 lines of non-comment non-whitespace YAML, to 606 files with 124,168 lines. Figure 3 shows the major events in UTF's growth: we created UTF (a), ported some CRUD tests (b), added Client-Side Operations Timeout tests (c), ported transactions tests (d), ported remaining CRUD and retryable writes/reads tests (e), and ported Client-Side Field-Level Encryption (f). Most drivers have replaced most of their runners for the various test formats in favor of one UTF runner, and they deleted test code on net. The Java driver had the most dramatic savings of 6038 LOC (Figure 4).

A. Bug Prevention

Once a spec is published, do the drivers that implement its YAML tests have fewer nonconformance bugs? We analyzed the CRUD spec (one of our largest and oldest specs) and the five drivers with long gaps between the spec's publication in February 2015 and their adoption of its YAML tests. (The other drivers adopted the YAML tests almost immediately, leaving no time to measure their conformance *without* YAML tests.) We used Claude Sonnet to classify 6836 resolved tickets in the five late-adopting drivers as either CRUD nonconformance bugs or other bugs. After review by Claude Opus and a human, 125 CRUD nonconformance bugs remained. The classification prompts, validation data, and analysis pipeline are in our public artifact repository.[1] Figure 5 shows nonconformance bug rates (bugs/year) before and after YAML test adoption. (The tests were ported from the CRUD-specific format to UTF during our analysis period, but UTF had no effect on the tests' efficacy, only on their maintainability.) Four drivers show reductions of 49–86%. We found many bugs in

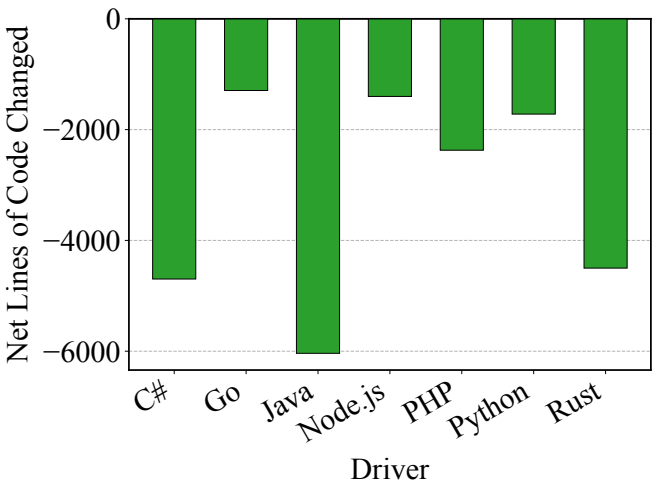

Fig. 4. Lines deleted during UTF migration.

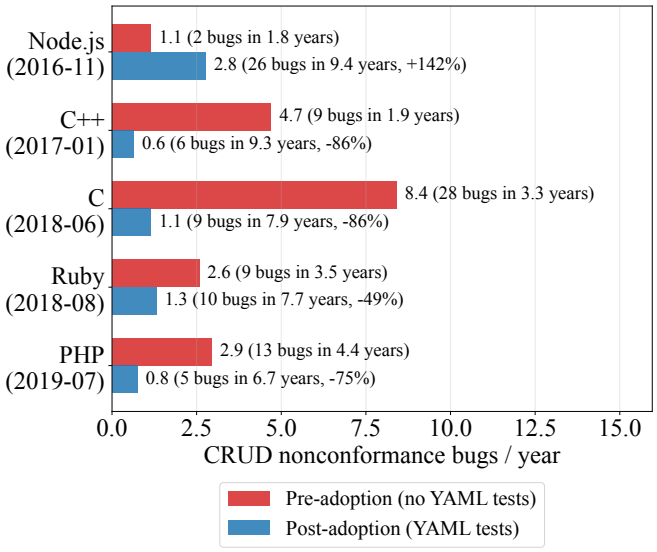

Fig. 5. CRUD nonconformance bug rates before and after YAML test adoption (5 late-adopting drivers). Pre-adoption window: spec published, no YAML tests. Post-adoption window: YAML tests adopted.

[1] https://github.com/mongodb-labs/issre-2026-polyglots-dilemma-artifacts

the C Driver early, when other drivers that wrap it implemented the spec and the YAML tests.

Even after we adopted YAML tests in these five drivers we found nonconformance bugs, because of coverage gaps in the YAML tests, or bugs in drivers' test runners. This was particularly true of the Node.js driver, where the bug rate was 142% higher after adoption. Although it passed the YAML tests, its bulk write implementation had many inconsistencies with the spec.

Threats to validity: This is a retrospective study with no control arm. As the drivers matured we would expect bug rates to decline regardless of YAML adoption. There were just a handful of nonconformance bugs per driver, so a few misclassifications would have a large effect. We mitigated this with multiple review passes (see the artifact repository).

B. Lessons Learned

We cannot unify everything: UTF is not suitable for every spec. We still have specialized YAML formats and runners for testing BSON serialization, the SDAM state machine, connection string parsing, and other features. Some tests are simply described in English. BSON and connection string tests assert on in-process data structures rather than on API calls or wire messages, so the entity map / operation / `expectEvents` structure does not fit. SDAM tests inject simulated server state directly into a driver's internal state machine, bypassing the public API layer entirely. If our YAML format were capable of describing any possible test, it would have become a Turing-complete programming language with interpreters in a dozen other languages, and its cost and risks would outweigh the benefits. We stopped folding formats into UTF when we reached the point of diminishing returns.

Single schema version: UTF's monotonically increasing schema version led to situations where a driver was blocked from implementing the tests for a new spec, because it had fallen several schema versions behind. If UTF files instead had a list of required test runner features, driver engineers would have had more flexibility about the order in which they implemented specs.

Judicious enforcement: Despite spec authors' efforts, our drivers still have some implementation-specific behavior. For example, a field on a result/response object might be optional or absent in some drivers. We added the `$$unsetOrMatches` operator to UTF so that test authors can permit drivers **not** to return a value in some cases, but require a **specific** return value if there is one.

Manual test authoring: Our YAML tests are written by hand. Automating test generation is difficult for several reasons. Writing correct expected wire-level assertions requires deep knowledge of the protocol. It is hard to distinguish a wrong test from a driver bug without human judgment. Spec authors need to understand why a test passes before they can trust it; an opaque generated test provides little confidence. These barriers have kept manual authoring the status quo. However, property-based testing or LLM-assisted generation—using the prose spec and existing tests as context—are promising directions we intend to explore.

V. Related Work

Our task—implementing the same network protocol and a consistent API in many programming languages—falls within the category of *multilanguage development* [5]. The subcategory of *polyglot programming* describes the development of an individual program with components in multiple languages; testing such programs is a well-studied problem [6], [7]. However, our problem is to test the conformance of *multiple* programs, each in a different language, to each other and to a specification. This subject has not been researched thoroughly, to our knowledge.

A. Domain-Specific Test Languages

Our YAML tests are an example of a *domain-specific test language*, or DSTL [8]–[12]. Many telecommunication protocols' test suites are written in the TTCN family of DSTLs [13]. A well-known DSTL is Gherkin, executed by Cucumber. We rejected Gherkin because it is designed for communication with non-technical stakeholders, whereas our tests are by and for engineers.

The Smithy project includes a DSTL that is more like our YAML tests [12]. Smithy is an interface definition language for services. Engineers write test cases in Smithy's DSTL which specify exactly how client and server implementations should serialize remote procedure calls (RPCs) on the wire as HTTP messages. Similar to our YAML tests, Smithy's DSTL tests interact with both a client's "upper" and "lower" surface—the in-process API and the wire messages. Interpreters are available in at least nine languages.

Ecma International maintains Test262, containing over 50,000 handwritten test cases to check JavaScript implementations' conformance to the ECMA-262 standard [11]. Like MongoDB, Ecma wrote a spec of the spec tests, describing the case file format and how test cases should be run. There are at least six independent implementations of the Test262 spec. COMFORT [11] is an experimental system that uses an LLM to generate JavaScript test cases from ECMA-262. The LLM benefits from the spec's rigidly structured English. COMFORT's test cases do not include expected outputs; instead, COMFORT applies differential testing with many JavaScript implementations.

B. Testing the Upper and Lower Surface

A major strength of our approach is that we test not only the return values of API calls, but also the contents of drivers' messages on the wire. The "ferry-clip" test architecture is the closest antecedent of this idea [14], [15]. It tests an implementation of an OSI protocol layer by controlling the protocols above and below it in the OSI stack, attaching to the SUT's top and bottom like a "clip." The test harness installs a proxy server which "ferries" messages produced by the SUT back to the test harness to be checked against assertions. Our tests' mechanics are quite different—we use direct procedure

calls at the top surface, and command-monitoring callbacks at the bottom—but the architecture is analogous.

C. Test Case Generation

MongoDB driver specifications are written in English, and we wrote our YAML tests by hand. But there are many examples of specifications that use formal logic or a finite state machine (FSM), from which one can generate test cases mechanically [16]–[22]. We could model MongoDB's client-server system as an FSM and check that the FSM's transitions match the drivers'. We could also use an LLM to generate more test files from the prose spec and existing tests. The LLM could autonomously check its work: if every driver fails an LLM-made test, the test is probably wrong, but if only some fail, the test may have uncovered a bug.

VI. Conclusion

We described a specification-based testing method developed over eleven years to ensure consistent behavior among libraries in a dozen languages, serving millions of users. We found that declarative, machine-interpretable test specs are effective at enforcing consistency. Our Unified Test Format minimizes the cost of maintaining tests: each driver implements a single UTF test runner; tests are written once and run everywhere. Our approach is not as advanced as some prior research, but its continuous use and evolution at scale offers an extended case study. We hope the lessons we learned prove useful to other teams enforcing consistent behavior across many implementations.

References

[1] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[2] S. S. Brilliant, J. C. Knight, and N. G. Leveson, "The consistent comparison problem in N-version software," *SIGSOFT Softw. Eng. Notes*, vol. 12, no. 1, pp. 29–34, Jan. 1987. [Online]. Available: https://dl.acm.org/doi/10.1145/24574.24575

[3] W. Schultz, S. Zhou, I. Dardik, and S. Tripakis, "Design and Analysis of a Logless Dynamic Reconfiguration Protocol," *LIPIcs, Volume 217, OPODIS 2021*, vol. 217, pp. 26:1–26:16, 2022, artwork Size: 16 pages, 922157 bytes ISBN: 9783959772198 Medium: application/pdf. [Online]. Available: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.OPODIS.2021.26

[4] M. Wynne, A. Hellesøy, and S. Tooke, *The Cucumber Book, Second Edition: Behaviour-Driven Development for Testers and Developers*. Raleigh, North Carolina: Pragmatic Bookshelf, 2017.

[5] G. Mussbacher, B. Combemale, J. Kienzle, L. Burgueño, A. Garcia-Dominguez, J.-M. Jézéquel, G. Jouneaux, D.-E. Khelladi, S. Mosser, C. Pulgar, H. Sahraoui, M. Schiedermeier, and T. van der Storm, "Polyglot Software Development: Wait, What?" *IEEE Software*, vol. 41, no. 4, pp. 124–133, Jul. 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10380193

[6] P. Houdaille, D. E. Khelladi, B. Combemale, and G. Mussbacher, "On Polyglot Program Testing," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, Jul. 2024, pp. 507–511. [Online]. Available: https://dl.acm.org/doi/10.1145/3663529.3663787

[7] H. Yang, Y. Nong, S. Wang, and H. Cai, "Multi-Language Software Development: Issues, Challenges, and Solutions," *IEEE Transactions on Software Engineering*, vol. 50, no. 3, pp. 512–533, Mar. 2024. [Online]. Available: https://ieeexplore.ieee.org/document/10413900/

[8] D. Felício, J. Simão, and N. Datia, "RapiTest: Continuous Black-Box Testing of RESTful Web APIs," *Procedia Computer Science*, vol. 219, pp. 537–545, 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1877050923003319

[9] A. R. da Silva, A. C. R. Paiva, and V. E. R. da Silva, "A Test Specification Language for Information Systems Based on Data Entities, Use Cases and State Machines," *Communications in Computer and Information Science,Model-Driven Engineering and Software Development*, vol. 991, pp. 455–474, 2019. [Online]. Available: https://doi.org/10.1007/978-3-030-11030-7_20

[10] C. Dragoi, C. Enea, S. Nagendra, and M. Srivas, "A Domain Specific Language for Testing Consensus Implementations," Apr. 2023, arXiv:2303.05893 [cs]. [Online]. Available: http://arxiv.org/abs/2303.05893

[11] G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, X. Sun, L. Bian, H. Wang, and Z. Wang, "Automated Conformance Testing for JavaScript Engines via Deep Compiler Fuzzing," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. Virtual Event, Canada: Association for Computing Machinery, Jun. 2021, pp. 435–450, arXiv:2104.07460 [cs]. [Online]. Available: http://arxiv.org/abs/2104.07460

[12] Smithy authors, "HTTP Protocol Compliance Tests - Smithy 2.0." [Online]. Available: https://smithy.io/2.0/additional-specs/http-protocol-compliance-tests.html

[13] J. Grabowski, A. Wiles, C. Willcock, and D. Hogrefe, "On the design of the new testing language TTCN-3," in *Testing of communicating systems: Tools and techniques. IFIP TC6/WG6.1 13th international conference on testing of communicating systems (TestCom 2000), august 29–september 1, 2000, ottawa, canada*, H. Ural, R. L. Probert, and G. v. Bochmann, Eds. Boston, MA: Springer US, 2000, pp. 161–176. [Online]. Available: https://doi.org/10.1007/978-0-387-35516-0_10

[14] G. V. Bochmann and A. Petrenko, "Protocol testing: review of methods and relevance for software testing," in *Proceedings of the 1994 international symposium on Software testing and analysis - ISSTA '94*. Seattle, Washington, United States: ACM Press, 1994, pp. 109–124. [Online]. Available: http://portal.acm.org/citation.cfm?doid=186258.187153

[15] S. T. Chanson, B. P. Lee, N. J. Parakh, and H.-X. Zeng, "Design and Implementation of a Ferry Clip Test System." *PSTV*, pp. 101–118, 1989, version Number: 1. [Online]. Available: https://dblp.org/rec/conf/pstv/ChansonLPZ89

[16] D. Richardson, O. O'Malley, and C. Tittle, "Approaches to Specification-Based Testing," *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 86–96, 1989. [Online]. Available: https://www.researchgate.net/publication/2722274_Approaches_to_Specification-Based_Testing

[17] E. Hoque, "Ensuring specification compliance, robustness, and security of wireless network protocols," Ph.D. dissertation, 2015. [Online]. Available: https://docs.lib.purdue.edu/dissertations/AAI10099173

[18] K. L. McMillan and L. D. Zuck, "Formal specification and testing of QUIC," in *Proceedings of the ACM Special Interest Group on Data Communication*. Beijing China: ACM, Aug. 2019, pp. 227–240. [Online]. Available: https://dl.acm.org/doi/10.1145/3341302.3342087

[19] S. Bishop, M. Fairbairn, M. Norrish, P. Sewell, M. Smith, and K. Wansbrough, "Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations," in *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Charleston South Carolina USA: ACM, Jan. 2006, pp. 55–66. [Online]. Available: https://dl.acm.org/doi/10.1145/1111037.1111043

[20] R. Singha, R. Mondal, R. Beckett, S. K. R. Kakarla, T. D. Millstein, and G. Varghese, "MESSI: Behavioral Testing of BGP Implementations." in *21st USENIX Symposium on Network Systems Design and Implementation*, Santa Clara, CA, USA, 2024, pp. 1009–1023. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/singha

[21] B. Neelakantan and S. V. Raghavan, "Protocol Conformance Testing — A Survey," in *Computer Networks, Architecture and Applications*, S. V. Raghavan and B. N. Jain, Eds. Boston, MA: Springer US, 1995, pp. 175–191. [Online]. Available: http://link.springer.com/10.1007/978-0-387-34887-2_11

[22] R. Dssouli, K. Saleh, E. Aboulhamid, A. En-Nouaary, and C. Bourhfir, "Test development for communication protocols: towards automation," *Computer Networks*, vol. 31, no. 17, pp. 1835–1872, Jun. 1999.