APIdemic: Verifying Idempotency of REST API Clients

BHAVIK KAMLESH GOPLANI, University of Kansas, USA

Bhavik is an undergraduate advised by Sankha Narayan Guria. His ACM student member ID is 3058380.

1 INTRODUCTION

REST APIs are ubiquitously used to write applications that interact with a huge range of cloud services from social networking to finance. These APIs use HTTP which unfortunately depends on reliable networks for successful completion. However, even the best network setups are prone to disruptions like intermittent outages or timeouts. To mitigate this, developers implement retries for failed REST API requests, but faulty retries in code can lead to duplicate requests, changing the semantics of the application. Ideally, servers would handle such duplicates by tagging unique sequence numbers to requests, but most real-world APIs leave this responsibility to the client, requiring API consumers to ensure *idempotency*. Overall, about 93% of web API developers use REST APIs, and REST API use consumes 83% of all Internet traffic [Simpson 2022]. Given the vast usage, programmers need tools to verify that the retries added in REST API client programs are *idempotent*, i.e., yield the same results under retries.

In this project, we present APIdemic, a push-button tool to verify the idempotency of REST API client programs. Our proposed verification method relies on a pair of REST APIs – one to update the server state and another to query it. We then translate these to an encoding in Dafny [Leino 2010] as safety and liveness lemmas to discharge the verification conditions automatically.

```
1 begin
2     charge_txn(user, credit_card, amount)
3 rescue
4     retry
5 end
```

(a) Buggy program that may cause double charges.

```
1 begin
2     charge_txn(user, credit_card, amount)
3 rescue
4     txn = query_txn(user, credit_card,
5         amount) rescue retry
6     retry if txn.nil?
7 end
```

(b) Correct program (idempotent) charges exactly once.

Fig. 1. Two programs (simplified) that charge the credit card.

2 BACKGROUND

Figure 1 shows a simplified example in the Ruby programming language with two programs that charge a customer's credit card, a typical use case handled by payment processors such as PayPal or Stripe. In Figure 1a, line 2, the program uses charge_txn to attempt a charge on the customer's card. This call has to go over the network to the card processors (Visa/Mastercard). Networks are unreliable and have intermittent errors due to timeouts or response not reaching back from the card network. In case of an error, a simple retry is not enough! For example, if the card network processed the charge but the response failed to be received by the payment processor because of network unreliability, retrying will cause a double charge on the user card. So the `retry` on line 4 triggers this bug in case of a network error.

In contrast, Figure 1b shows an alternate implementation for the same program with this error mitigated. On line 4, in case of a network error, the program first attempts to check the status of the original charge using query_txn. Note, that checking the charge simply reads the data and it is safe to retry (`rescue retry` clause at the end) till it finds the status from the card network. If the original charge was successfully processed by the card network, the program chooses to exit

Author's address: Bhavik Kamlesh Goplani, University of Kansas, Lawrence, Kansas, 66045, USA, bhavik@ku.edu.

the code block without retrying. So the `retry` on line 6 triggers only if the card network did not process the original charge, i.e., the txn value is `nil`. Thus we only retry if the previous attempt did not succeed and the txn is `nil`, i.e., the previous charge did not go through.

Related Work. Ramalingam and Vaswani [2013] have addressed fault tolerance in network retries and failures via idempotency by designing domain-specific languages. Input/output (I/O) dependent idempotence bugs are a critical issue in intermittent systems, where repeated I/O operations can result in inconsistent states [Surbatovich et al. 2019]. In contrast, we look at cases when software services communicate over the network via REST API calls. Testing is a common technique used to identify bugs and security vulnerabilities in APIs by running the program on a variety of inputs. However, the non-determinism and low frequency of idempotency bugs make them an ill-fit for testing [Zhang and Arcuri 2023]. This makes idempotency a great candidate for formal verification.

Verification techniques have also been applied to stateful serverless applications built on cloud platforms such as Amazon Web Services (AWS), Google Cloud, and Microsoft Azure, where automated tools ensure idempotent behavior despite the distributed nature of serverless architectures [Ding et al. 2023]. Building on this, our work generalizes these verification techniques applicable to any API-driven system. Additionally, our tool draws on methods used in systems like IronFleet [Hawblitzel et al. 2015] and LVR [Yao et al. 2024], where safety and liveness properties are verified using formal methods to ensure reliability in distributed systems. Typically, such tools deal with a distributed system and need some *manual effort* to formally verify system correctness.

3 OUR APPROACH

We extended a Ruby-type system, RDL [Kazerounian et al. 2019] to add effect annotations to high-level library methods that perform REST API calls to reason about server state. These effects denote Read or Write operations over some abstract server states, similar to effects in RbSyn [Guria et al. 2021]. In practice, most REST API endpoints come in pairs that allow high-level API to update or read server state, and the corresponding Ruby library methods closely mirror the same. For example, library methods like query_txn and charge_txn are annotated effects such as Read<Charge> and Write<Charge>. The parameter in the polymorphic effect label allows APIdemic to know if the reads and writes are happening to the same abstracted server side resource. These library method annotations are the only user effort required for our work and the annotations are reusable across projects.

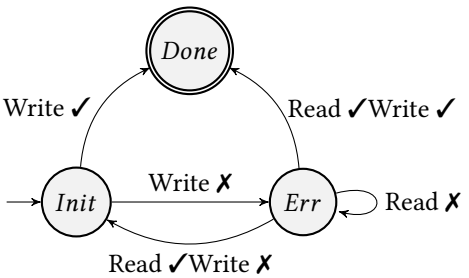

Fig. 2. State machine modeling transitions from the correct program in Figure 1b

We use control flow analysis [Shivers 1991] to translate the code into state machine transitions by abstracting a program only to the control flow jumps decorated with the effect labels. Figure 2 shows the translation to the state machine for the program in Figure 1b. The program start (*Init*) can only enter completion (*Done*). All other transitions are control flows derived from exception handling via `begin`, `rescue`, and `retry`. Write ✓denotes the program can successfully complete if the charge_txn request runs successfully. If the request fails (Write ✗), the program transitions to the *Err* state and checks the status via the query_txn call. However, query_txn may also experience network issues, requiring retries (Read ✗), and potentially looping back to itself. If query_txn confirms that the initial write succeeded, the program can complete (Read ✓Write ✓ transition to

Done). However, if the query_txn reports the previous charge_txn failed, the program has to start from scratch again (Read ✓Write ✗ transition to *Init*).

These transitions are translated as logical predicates in Dafny, with key actions like reads, writes, and success API calls modeled as natural numbers. The success counter is incremented only on the control jumps that end in *Done*. Note, that these states are demonstrated as per the example in Figure 1b. More generally, our algorithm generates a state machine with one *Init* and *Done* state and multiple *Err* states depending on the number of `rescue` blocks in a method being analyzed.

```
datatype State = Init | Err | Done
datatype Variables = Variables(read:nat, write:nat, success:nat, state:State)
predicate Valid(v:Variables) {
    ∧ (v.state == Init ∨ v.state == Err ⇒ v.success == 0)
    ∧ (v.state == Done ⇒ v.success == 1)
    ∧ (v.write > v.read ⇒ v.write - v.read ≤ 1) }
```

The Valid predicate above gives the safety condition for an idempotent program. If the program is in the *Init* or the *Err* state, the number of successful writes is 0, but it is 1 (denoting the write has happened exactly once) if the program is in the *Done* state. Additionally, it enforces bounds on the read and write counters, ensuring that if write operations exceed read, the difference is at most 1, as each read and write operation pair aligns to update and read the state. The transitions in Figure 2 are modeled as a relation between two program states in Dafny. A Next predicate (definition omitted) states that there exists a step in the program such that it can take one transition as per Figure 2. Finally, we decompose the verification problem into two lemmas: *safety* and *liveness*.

Safety. The safety lemma ensures that all the possible transitions in the program state will leave the program to be in a valid state, i.e., successfully complete the REST API call exactly once:

```
lemma SafetyProof() ensures forall v | Init(v) :: Valid(v)
ensures forall v, v' | Valid(v) ∧ Next(v, v') :: Valid(v')
```

This states *Init* is a valid state and any state after, from a potential Next transition, is also valid.

Liveness. The liveness lemma guarantees that a trace of program transitions that begins in the *Init* state will eventually reach the *Done* state:

```
lemma LivenessProof(trace: Trace, n: nat) returns (n': nat)
requires IsTrace(trace) ∧ Init(trace(n)) ∧ FairNetwork(trace)
requires forall i: nat :: i ≥ n ⇒ (Valid(trace(i)) ∧ Next(trace(i), trace(i+1)))
ensures n ≤ n' ∧ trace(n').state == Done ∧ trace(n').success == 1
```

The above lemma states that given some trace in *Init* state, eventually the final returned state is a *Done* state. This requires fair network assumption, i.e., if no network errors happen, the *Done* state is reachable in the trace. Proving this amounts to proving termination for the abstracted version of our program. We elide the proofs here for brevity.

We have proved the lemmas once manually. APIdemic uses the CFG from RDL to automatically translate the program to only the necessary predicates. Dafny then verifies if our proofs hold for the automatically generated predicates from the source program. This works on some handcrafted toy examples at the moment. We discuss planned evaluation in § 4.

4 FUTURE WORK

We have already developed a benchmark suite of 18 programs in Ruby comprising buggy and correct examples of idempotent behavior, curated from StackOverflow. We have annotated the relevant library and API methods with the necessary side effects. We plan to evaluate the benchmark programs on APIdemic to verify their idempotency with the expected behavior.

REFERENCES

Haoran Ding, Zhaoguo Wang, Zhuohao Shen, Rong Chen, and Haibo Chen. 2023. Automated Verification of Idempotence for Stateful Serverless Applications. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 887–910. https://www.usenix.org/conference/osdi23/presentation/ding

Sankha Narayan Guria, Jeffrey S. Foster, and David Van Horn. 2021. RbSyn: type- and effect-guided program synthesis. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3453483.3454048

Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. 2015. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/2815400.2815428

Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S. Foster, and David Van Horn. 2019. Type-level computations for Ruby libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 966–979. https://doi.org/10.1145/3314221.3314630

K. Rustan M. Leino. 2010. Dafny: an automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning* (Dakar, Senegal) *(LPAR'10)*. Springer-Verlag, Berlin, Heidelberg, 348–370.

Ganesan Ramalingam and Kapil Vaswani. 2013. Fault tolerance via idempotence. *SIGPLAN Not.* 48, 1 (Jan. 2013), 249–262. https://doi.org/10.1145/2480359.2429100

Olin Shivers. 1991. The Semantics of Scheme Control-Flow Analysis. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91, Yale University, New Haven, Connecticut, USA, June 17-19, 1991*, Charles Consel and Olivier Danvy (Eds.). ACM, 190–198. https://doi.org/10.1145/115865.115884

J Simpson. 2022. *20 Impressive API Economy Statistics.* https://nordicapis.com/20-impressive-api-economy-statistics/

Milijana Surbatovich, Limin Jia, and Brandon Lucia. 2019. I/O dependent idempotence bugs in intermittent systems. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 183 (Oct. 2019), 31 pages. https://doi.org/10.1145/3360609

Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. 2024. Mostly Automated Verification of Liveness Properties for Distributed Protocols with Ranking Functions. *Proc. ACM Program. Lang.* 8, POPL, Article 35 (Jan. 2024), 32 pages. https://doi.org/10.1145/3632877

Man Zhang and Andrea Arcuri. 2023. Open Problems in Fuzzing RESTful APIs: A Comparison of Tools. *ACM Trans. Softw. Eng. Methodol.* 32, 6, Article 144 (Sept. 2023), 45 pages. https://doi.org/10.1145/3597205

Received 2022-11-10; accepted 2023-03-31