

# Verification of the CRAQ Protocol

Henry Tang  
Aaron Wu  
COS 516  
Professor Zak Kincaid

## 1 Introduction

Our final project was creating a verified implementation of Chain Replication with Apportioned Queries (CRAQ) [10], a fault-tolerant distributed object-storage system. In particular, the safety property we chose to verify was linearizability; informally, this is a guarantee that the set of executions form a sequential history that respects real-time order. Our specification and verification language of choice for this project was Ivy.

Both describing and implementing distributed protocols correctly is notoriously difficult. At the protocol-level, designers may miss certain execution scenarios, which may lead to incorrect protocols. A classic example at this level of abstraction is the Chord protocol. The original work provided a safety proof, yet later model checking and verification efforts discovered several critical errors [4, 13]. Even if a protocol is almost surely correct, translating said protocol to a concrete implementation often requires a significant amount of engineering work and can introduce subtle deviations from the original protocol description. A good example of this difficulty is one of the assignments from the distributed systems class here at Princeton, where the only task is a faithful implementation of Ongario and Ousterhout’s Raft protocol. Many production databases systems have also exhibited serious safety violations.<sup>1</sup>

CRAQ is a well-known distributed protocol that, to the best of our knowledge, has not been formally verified since its publication in 2009. Despite its apparent simplicity, we wanted to obtain convincing evidence that it guarantees linearizability in the absence of failures. Furthermore, there exist a few publicly available implementations but the correctness of these is unclear. As such, we also set out to extract proof-of-concept C++ code that could be developed into a fully functional system.

## 2 Overview

### 2.1 Chain Replication with Apportioned Queries

CRAQ is a linearizable, fault-tolerant, object-storage system. As the name suggests, it is an augmentation of van Renesse et. al’s chain replication protocol, where system servers are arranged linearly in a logical chain [11]. Following the convention of both works, we call the start of the chain as the head and the end as the tail. Assuming the presence of a reliable failure detector and fail-stop servers, both CRAQ and chain replication systems will continue to function correctly in the presence of  $n - 1$  server crashes at arbitrary locations in the chain, where  $n$  is the total number of servers in the system. CRAQ exposes a simple put/get API to clients:

1. **write(key  $k$ , value  $v$ ):** Associate value  $v$  with key  $k$  within this storage system.

---

<sup>1</sup>As explored by Jepsen.

2. **read(key  $k$ ):** Read the most recent value associated with the key  $k$ .

Like chain replication, each write request must start at the head. The write is propagated down the chain until it is acknowledged by the tail, at which point the write is considered committed. Each server locally applies each write upon receiving it. This ensures that when a client receives a response for the write from the tail, it has already been received and replicated by every single non-faulty node in the chain. Once the tail has committed the entry, it propagates the request back through the chain towards the head, in order to inform the other servers that the write has been committed. Since all operations are processed in order relative to the tail, writes satisfy strong consistency.

The key innovation of CRAQ is the ability to serve read requests from *any* server in the chain; in contrast, only the tail is responsible for reads in chain replication. Thus, for read-heavy workloads CRAQ offers lower latency (as replicas can be placed closer to clients) and higher throughput. To maintain linearizability, CRAQ introduces the notion of version numbers and dirty bits. Each write is associated with a monotonically increasing version number, determined by the head of the chain. If the most recent write on a server is known to be committed, then the dirty bit on that server is set to false. Otherwise, the dirty bit is set to true.

To serve a read request, a server checks if the dirty bit is false. If so, the server can safely return the data associated with the most recent version number it has seen. Otherwise, the server must contact the tail for its most recent committed version number. On hearing a response from the tail, a server returns the corresponding data from its local store to the client.

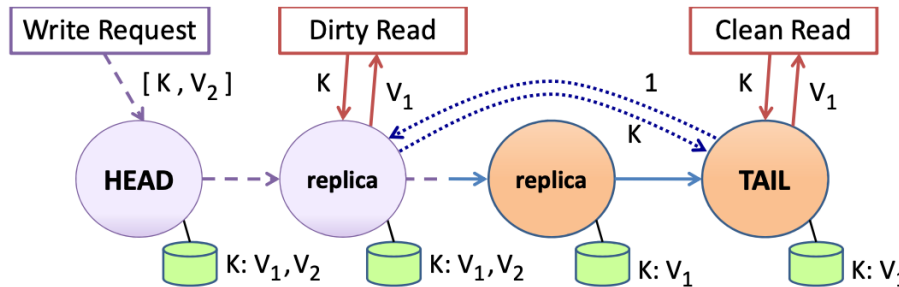


Figure 1: An illustration of the CRAQ protocol.

## 2.2 Ivy

Ivy is a tool that is designed for specifying, implementing, and verifying infinite-state protocols [8]. The research community has already produced a number of automated tools for formal proof. These include powerful SMT solvers such as Z3 that are able to check for satisfiability within a large class of logics and theories. By discharging proof obligations to these SMT solvers, several verification-aware programming languages, such as Dafny, support user-defined specification to guarantee correctness and even some level of invariant discovery. However, these tools can be quite difficult to use, as they generate uninformative error messages when the provided specification is not sufficient to prove the desired properties. For instance, it is often very difficult to understand what, and sometimes even where, invariants need to be added.

As a result, many efforts over the past decade towards verifying complex distributed systems have used significant manual effort and expertise in formal methods [3, 5, 12]. To

enable verification in a more interactive and semi-automated manner, Ivy’s generates graphical and understandable *counterexamples to induction*. These counterexamples either describe an initial state where the invariant is not satisfied, or start out in an initial state where the desired invariants are satisfied and describe a program execution to a state where an invariant is violated. This allows the user to reason about why their system is not correct, and interactively modify either their invariants or their implementation accordingly.

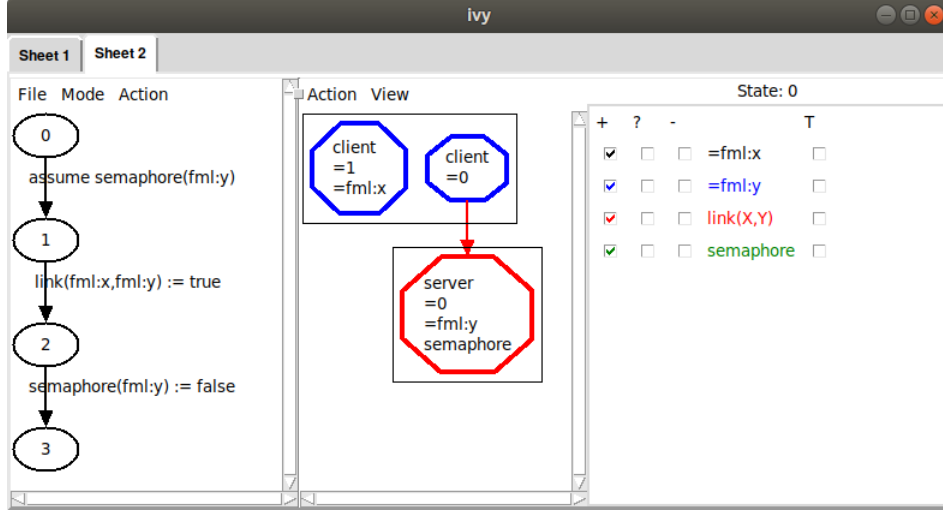


Figure 2: An example of a counterexample found by Ivy. The left panel displays the series of transitions that lead to an invariant violation, while the middle panel displays the state of the system at a particular point in the program. In this case, the error results from the server’s semaphore being up and available even though a client is connected. This should not be a possible initial state in the system, and adding another inductive invariant resolves this.

Ivy’s application is limited to distributed protocols that can be modelled as a program written in RML (relational modeling language), a restricted but decidable logic. This enhances the usability of Ivy, as it ensures that Ivy can always either prove correctness, or generate a counterexample to induction. RML represents program states using first order relations. However, a few important restrictions include the fact that RML does not allow arithmetic operations, requires updates to functions and relations to be quantifier free, and restricts invariants and verification conditions to be expressible in EPR (Effectively Propositional Logic). EPR is the set of sentences in first order logic that contain a  $\exists^*\forall^*$  prefix in prenex normal form, and do not contain any function symbols. Despite these restrictions, however, RML is still Turing complete and can be used to specify a wide variety of distributed protocols.

A particularly helpful feature of Ivy is its support of an object-oriented style of specification. Users can describe their system as a set of isolates, which can interact and update their internal state through actions. To check correctness, Ivy adopts a rely-guarantee style approach. If all isolates can satisfy their specifications and invariants under the *assuming* action preconditions, and guarantee that inputs to actions in other isolates satisfy the necessary preconditions, then the system collectively satisfies the specification. Further detail on the soundness of this approach can be found in [9].

Ivy proves the correctness of programs using weakest liberal preconditions. At a high level, if  $T$  is a line of code and  $R$  is condition corresponding to the program state, then  $wlp(T, R)$  is the weakest condition that must hold before  $T$  so that  $R$  holds after  $T$ . Solving the weakest precondition problem in general, however, is undecidable due to the presence of loops. To

ensure this can be done in a decidable manner, users must provide invariants that hold at the beginning and end of each loop iteration. In Ivy, preconditions correspond to an *assume* at the beginning of an action, and postconditions correspond to an *assert* at the end of an action. In addition, Ivy does not have support for loops. Instead, an invariant  $I$  for an isolate is inductive if initialization satisfies  $I$  and if each exported action maintains  $I$ .

## 2.3 Sift

Sift is a general process for verifying distributed systems, based on the classical notion of refinement [3, 6]. Instead of specifying the entire system in a monolithic proof, Sift (and Ironfleet) opts for a layered approach. Starting at a high-level specification, the methodology iteratively lowers the abstraction down to a complete, implementation-level specification. Each refinement layer encodes the externally visible state of the layer above within its specification. In particular, lower layers initiate state transitions in the upper layer through upcalls. In Ivy, this corresponds to calls from lower layer isolates to higher layer actions. Figure 3 illustrates this concept.

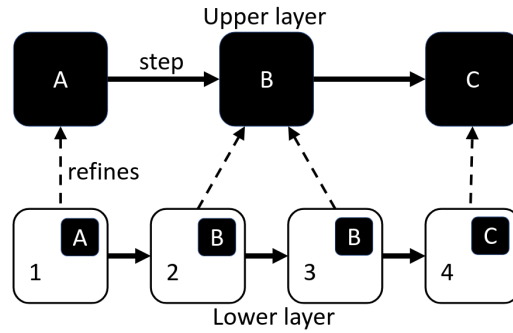


Figure 3: Illustration of refinement and encapsulation.

Correctness of the system is then guaranteed through the verification of two conditions. First, the externally visible state of lower layers must agree with that of the upper layers. For example, if the implementation layer network has registered a response, then the abstract layer must also have “responded” to the request. Second, we need to ensure that the state transitions initiated by lower layers are valid with respect to our system semantics. For example, we would not want to commit a write if it has not yet been replicated on the tail node. Thus, we must guarantee that various pre- and postconditions on isolate upcalls are never violated.

These two types of verification conditions are expressed in Ivy in two ways: either conditional blocks or invariants. Any condition that requires a check on proof-based state, such as that exposed by an abstract upper layer must be expressed as an invariant. This ensures preservation of correctness during Ivy’s C++ code extraction.

One of the chief advantages of the Sift approach is the ability to leverage automated invariant search tools. Sift elects to use IC3PO to make its invariant inductive. If IC3PO times out or is otherwise unable to perform the search, additional refinement layers are introduced. Refer to Figure 4 for an illustration of the Sift methodology.

## 2.4 IC3PO

The process of proving that an unbounded distributed protocol satisfies a provided safety property involves incrementally strengthening invariants until they are inductive, assuming

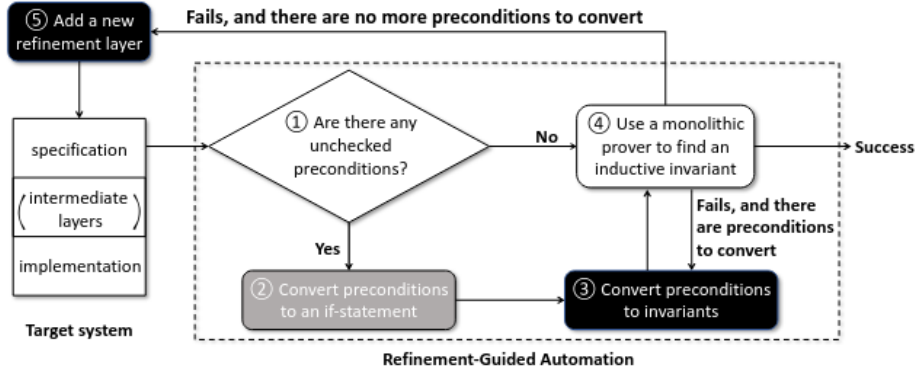


Figure 4: Overview of Sift workflow

the implementation is correct to begin with. This is a very time consuming process, even with the counterexample guidance of a tool such as Ivy. IC3PO is a tool that expedites this process and aims at reducing the amount of manual labour required by automating the inductive invariant search for unbounded distributed protocols.

At a high level, IC3PO works by first finding inductive invariants for a finite instance of the distributed protocol: for instance, it may restrict the number of servers or the size of the values that can be passed around. To find inductive invariants, IC3PO largely follows the PDR/IC3 algorithm, which computes safe inductive invariants given a transition system and a safety property [1, 7]. IC3PO makes this process more efficient for distributed systems by aggressively exploiting the symmetry of distributed systems, allowing it to use a single learned clause to generate a set of symmetrically-equivalent clauses. This in turn is then converted into a compact quantified predicate for the finite model size [2].

To extend this to an infinite instance, IC3PO performs a finite convergence check. It increments each of the finite parameters of the instance, and checks to see if the invariant is still inductive and sufficient to prove the safety property. This can be summarized as follows.

1.  $Init(|s_1|, \dots, |s_i| + 1, \dots, |s_n|) \rightarrow Inv_{|s_1|, \dots, |s_n|}(|s_1|, \dots, |s_i| + 1, \dots, |s_n|)$
2.  $Inv_{|s_1|, \dots, |s_n|}(|s_1|, \dots, |s_i| + 1, \dots, |s_n|) \wedge T(|s_1|, \dots, |s_i| + 1, \dots, |s_n|) \rightarrow Inv'_{|s_1|, \dots, |s_n|}(|s_1|, \dots, |s_i| + 1, \dots, |s_n|)$

The  $s_i$  are the various finite parameters of the system, and the subscripts refer to the inductive invariants found for a system with those specific finite parameters.  $Inv_{|s_1|, \dots, |s_n|}(|t_1|, \dots, |t_n|)$  refers to the quantified invariant found on the finite instance with parameters  $s_1, \dots, s_n$ , but used in the context of the finite instance with parameters  $t_1, \dots, t_n$ . Intuitively, if these two properties holds for each  $i$ , then a size increase does not introduce previously unseen behaviour. However, to actual verify these invariants hold, IC3PO still needs to perform the following checks on an unbounded system.

1.  $Init \rightarrow Inv_{|s_1|, \dots, |s_n|}$
2.  $Inv_{|s_1|, \dots, |s_n|} \wedge T \rightarrow Inv'_{|s_1|, \dots, |s_n|}$

Since these checks may lie beyond the decidable fragment of first order logic, they may not terminate. If either the bounded or unbounded invariant checks are not valid, then IC3PO repeats the procedure on a new finite instance with increased parameters, as seen in Figure 5.

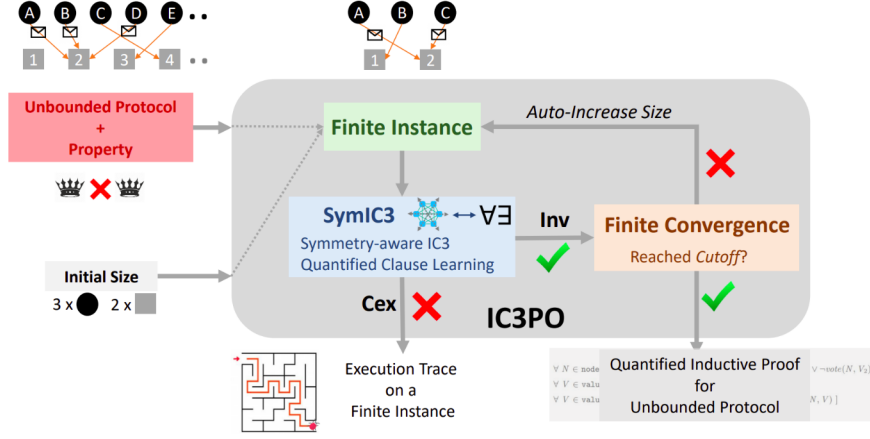


Figure 5: A flowchart describing the execution of the IC3PO tool.

## 2.5 Deliverables

Our project deliverables are publicly available and can be found [here](#). This repository contains both a specification of the protocol, as well as a C++ implementation, extracted through Ivy’s built-in translation tool. Instructions for checking the specification and locally running the implementation can be found in the repository’s README. We found that the sharded hash table example from Sift contained several useful components, primarily the specification of TCP and the abstract linearizable map, found in spec.ivy. Most of the core specification unique to CRAQ can be found in the trans.ivy and craq\_system.ivy files. Type definitions we use throughout the project can be found in common.ivy. Users can interact with the C++ system either through the command line or by running a benchmarking client, similar to the one provided in Sift’s sharded hash table example.

## 3 Project Tasks

Before we started on verifying CRAQ, we acquainted ourselves with Ivy’s functionality, IC3PO, and devoted time to understanding the Sift methodology. To do this, we read through the corresponding papers [8, 2, 6]. We also worked through the Ivy [tutorial](#) and spent time deciphering the syntax/semantics of the language.

Our implementation of CRAQ utilizes the Sift methodology, and is inspired by the sharded hash table example. Our protocol is broken up into a specification and an implementation layer. Our specification layer is an key-value map. It guarantees the following properties:

1. When CRAQ responds to a read request, it will return the most recently committed value in the key-value map.
2. When CRAQ responds to a write request, it will first commit the associated key-value pair into the key-value map.
3. CRAQ never performs a write request twice.

Together, Property 1 and 2 imply that any read reply to the client reflects the data from the most recent write. Property 3 is critical for linearizability in the presence of client retries.

Our network specification allows for 5 types of messages for inter-server communication: request, reply, inquire, inform, and acknowledge-commit. Between every pair of servers, the

communication channels are specified as 5 different first-in-first-out queues, one for each type of message. Sending a message involves pushing to the appropriate queue, while receiving a message involves popping the first entry in the queue. We can specify our network in this manner because our actual implementation of inter-server communication follows the Transmission Control Protocol (TCP) standard, which is provided by the Ivy TCP standard library. In particular, TCP guarantees packets are sent and received in the same order, and that no packets are dropped in the network. The invariants in `trans.ivy` are used to enforce these constraints in our specification.

Our implementation API exposes `get(k)` and `set(k, v)` actions to the client, which correspond to read and write requests. In addition, our implementation layer contains the logic used for committing requests and handling inter-service communication. Any call to `set(k, v)` from the client is forwarded directly through a request message to the head, unless the contacted server is already the head. Afterward, the request message is continuously passed along the chain until it reaches the tail, at which point the write is committed and the tail responds to the client using a reply message. The tail then propagates a acknowledge-commit message backwards through the chain until it reaches the head.

Upon receiving a call to `get(k)` from the client on a server  $s$ ,  $s$  will first check if its dirty bit for this key is true. If not, it immediately commits and sends a reply message to the client. Otherwise, it sends a inquire message to the tail, which responds using an inform message with the highest committed version corresponding to key  $k$ .

Notice that right before we send a reply message to the client, we always commit the entry by calling up into the specification layer. This ensures that each transition in the specification layer corresponds to one or more transitions in the implementation layer, which forms the basis of Sift’s refinement technique.

Unfortunately, IC3PO was unable to generate inductive invariants for our implementation of CRAQ. The cause of this problem was that we index into our multi-versioned map using a key-version pair i.e. a struct type. As far as we are aware, Ivy type inference does not support expressions involving struct members within an invariant, which we found to be necessary component in proving safety. Consequently, we had to come up with the inductive invariants ourselves, adding features to our implementation to enable more precise expressions along the way. Further discussion surrounding this issue can be found in section 5.

Below we provide two example invariants, which describe some interesting properties of the system.

$$\begin{aligned} & trans.acked(R, D) \wedge server(D).highestVersion(query.qkey(R)) = query.qvnum(R) \\ \implies & query.qvalue(R) = server(node.max).viewMap(query.qkey(R)) \end{aligned} \quad (1)$$

This invariant says that if an entry is acknowledged and remains the highest version number, then the corresponding value is the most recent committed write to the key

$$\begin{aligned} & (trans.requested(R1, D1) \wedge trans.requested(R2, D2) \\ & \wedge (D1 = 0) \wedge (D2 = 0) \wedge (query.qsrc(R1) = query.qsrc(R2) \\ & \wedge query.qid(R1) = query.qid(R2))) \implies (query.qvnum(R1) = query.qvnum(R2)) \end{aligned} \quad (2)$$

This invariant corresponds to the property that the system cannot have two request message in the network with the same version number.

### 3.1 Individual Contributions

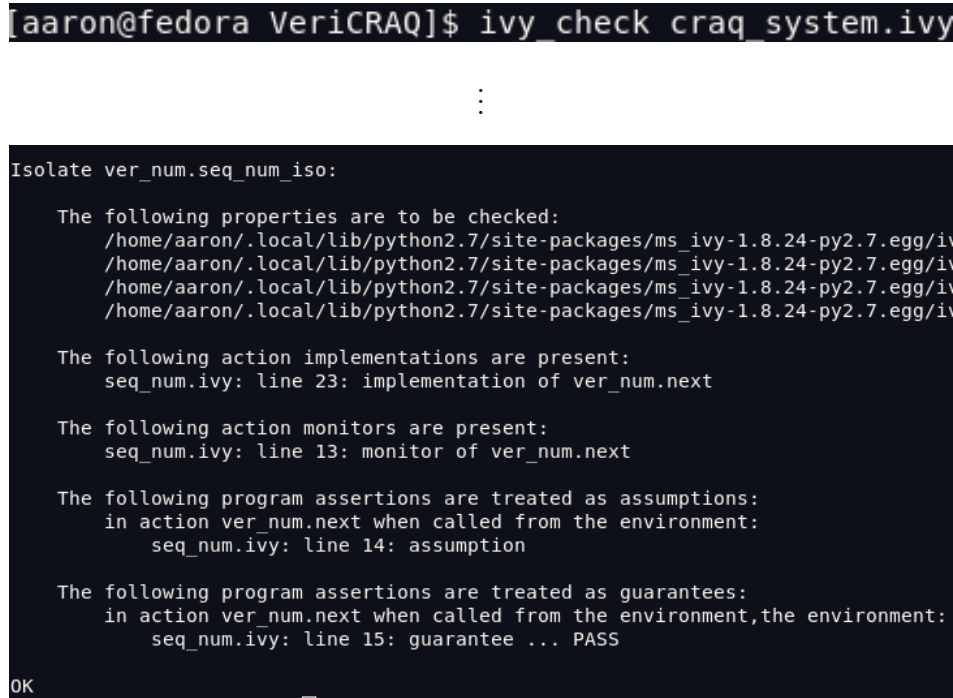
Aaron did most of the initial work for the project such as finding relevant papers, environment setup, experimenting with Ivy examples, and the initial framework for the CRAQ specification

effort. He contributed to parts of the implementation layer specification, finished the invariant search, and extracted the C++ implementation.

Henry implemented most of the network specification, parts of the implementation layer specification, and manually found most of the invariants after we deemed IC3PO to be unhelpful. He also did some work experimenting with Ivy examples, and wrote a large portion of this report.

## 4 Results

Unfortunately, due to several mitigating factors, we were not able to benchmark performance of the C++ implementation. Our main result is the verification of linearizability of the “implementation” layer. See Figure 6 for the results of running `ivy_check`.



```
[aaron@fedora VeriCRAQ]$ ivy_check craq_system.ivy
:
Isolate ver_num.seq_num_iso:

The following properties are to be checked:
/home/aaron/.local/lib/python2.7/site-packages/ms_ivy-1.8.24-py2.7.egg/iv
/home/aaron/.local/lib/python2.7/site-packages/ms_ivy-1.8.24-py2.7.egg/iv
/home/aaron/.local/lib/python2.7/site-packages/ms_ivy-1.8.24-py2.7.egg/iv
/home/aaron/.local/lib/python2.7/site-packages/ms_ivy-1.8.24-py2.7.egg/iv

The following action implementations are present:
seq_num.ivy: line 23: implementation of ver_num.next

The following action monitors are present:
seq_num.ivy: line 13: monitor of ver_num.next

The following program assertions are treated as assumptions:
in action ver_num.next when called from the environment:
seq_num.ivy: line 14: assumption

The following program assertions are treated as guarantees:
in action ver_num.next when called from the environment,the environment:
seq_num.ivy: line 15: guarantee ... PASS

OK
```

Figure 6: Running `ivy_check`. Majority of check output omitted for clarity.

## 5 Discussion/Conclusion

Documentation on Ivy is rather lacking and mostly based around a collection of incomplete and sometimes incorrect toy examples. We spent a significant portion of time reasoning about odd behavior we encountered.

Additionally, while Sift provided a publicly available repository with scripts for reproducing their experiments, the author’s version management was not great. Initially, we assumed that we could use the most recent versions of IC3PO and Ivy to verify the protocols implemented by the authors of Sift. This was not the case. Their results are only reproducible when using a separate branch of Ivy, and an older version of IC3PO. Not using the former results in missing Python files needed for VMT generation, and not using the latter greatly slows down the performance of IC3PO, often from a few minutes needed for termination to a several hours.



We also ran into some difficulties with regards to representing a multi-versioned map in Ivy. The naive method is to instantiate an ordered map type for each key; however, this is not possible in Ivy, as it does not support parameterized typing. We worked around this limitation by flattening the representation, creating a map from lexicographically ordered (key, version number) pairs to values. Unfortunately, IC3PO’s VMT-LIB translation utility could not handle ordering over pairs. This precludes the ability to garbage collect no longer needed versions in our system. As mentioned above, this also meant we were not able to use IC3PO in the process of finding inductive invariants, due to limitations in Ivy’s type system.

During our process of manually finding inductive invariants, we had to augment the “implmentation” layer specification with an auxiliary function from keys to values, denoted `viewMap`. It turns out that the addition of this function is enough for IC3PO to successfully generate invariants.

Finally, with regards to the C++ implementation, Ivy does not support network interaction with external clients. As such, we had to modify the Ivy generated C++, following Sift’s approach. At this point, the client does work and can interact with the system, but system performance severely degrades in doing so. The reason for this is unclear and is a good direction for future work. Another possible area for further research is proving liveness. In this project, we proved that the system guarantees linearizability but this does not discount the possibility of the system never completing certain requests.

## References

- [1] Aaron R Bradley. Sat-based model checking without unrolling. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 70–87. Springer, 2011.
- [2] Aman Goel and Karem Sakallah. On symmetry and quantification: A new approach to verify distributed protocols. In *NASA Formal Methods Symposium*, pages 131–150. Springer, 2021.
- [3] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R Lorch, Bryan Parno, Michael L Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 1–17, 2015.
- [4] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI’07, page 18, USA, 2007. USENIX Association.
- [5] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6), 2010.
- [6] Haojun Ma, Hammad Ahmad, Aman Goel, Eli Goldweber, Jean-Baptiste Jeannin, Manos Kapritsos, and Baris Kasikci. Sift: Using refinement-guided automation to verify complex distributed systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 151–166, 2022.
- [7] Matteo Marescotti, Arie Gurfinkel, Antti EJ Hyvärinen, and Natasha Sharygina. Designing parallel pdr. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 156–163. IEEE, 2017.

- [8] Kenneth McMillan and Oded Padon. *Ivy: A Multi-modal Verification Tool for Distributed Algorithms*, pages 190–202. 07 2020.
- [9] Marcelo Taube, Giuliano Losa, Kenneth L. McMillan, Oded Padon, Mooly Sagiv, Sharon Shoham, James R. Wilcox, and Doug Woos. Modularity for decidability of deductive verification with applications to distributed systems. *SIGPLAN Not.*, 53(4):662–677, jun 2018.
- [10] Jeff Terrace and Michael J Freedman. Object storage on craq: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference*, 2009.
- [11] Robbert Van Renesse and Fred B Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, 2004.
- [12] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. *SIGPLAN Not.*, 50(6):357–368, jun 2015.
- [13] Pamela Zave. Using lightweight modeling to understand chord. *SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, mar 2012.