Eliminating the Formality Gap: A Simple Set Replication System

Maofan (Ted) Yin
my428@cornell.edu

November 25, 2017

Abstract

This report serves as a survey and practice on the certified distributed systems. Several best-of-time works that strive to provide mechanically-checked proofs for practical distributed systems are reviewed. By comparing the traditional specification verification to the most recent design of proof systems, the formality gap issue is discussed. Finally, the author has built a simple but complete set replication system that is both provably correct and practically deployable, as a proof-of-concept on eliminating the gap.

1 Introduction

As a PhD who mainly works on consensus protocols and distributed systems, I am interested in the latest results from applying mechanical proof-checking systems to traditional network and distributed systems. These systems are notoriously difficult to formalize and rigorously specify, given the fact that many of them imply concurrency in execution. Usually in such a system or software, there are more than one participants which send and receive data and alter their internal states. Thus, the state of the whole system is pretty much like a Cartesian product of possible states for each node. What's worse, the data that are passed around the system also have exponentially large space of temporal interleaving. The real clock time in the world for the system, however, sometimes may not serve as a good measurement for describing a trace of run. Hence, it gives rise to some certain techniques that redefine the meaningful ticks for each event to cut down the vast space of concurrent runs, such as logical clocks, vector clocks, and synchronized physical clocks [1].

The primitives invented to better model and capture the concurrent behavior of systems are sufficiently powerful to help researchers to prove the invariants and implement systems, however, not capable of leading to writing proofs that are more rigorously, or mechanically checked by a proof checker. Moreover, given the fact the human-written proof is actually correct, there exists a noticeable gap between the protocol specifications and the actual system implementations. The former is easier to model, simpler to reason about, and thus has a greater chance of being correct. But the latter is hard to formalize, complex to analyze and always haunted by bugs. A proof checker, or more advanced, a proof system dedicated to distributed systems, could be the right cure to these problems: it automatically verifies the logic of the protocol specification better than human efforts, and additionally, it gives the opportunity to seriously look into the implementation code and mechanically check whether the stated invariants or properties are still preserved.

One well-known effort in formalization of concurrent systems goes to Lamport, who designed a language of temporal logic, TLA^+ [2], to facilitate the specification of concurrent protocols. There have been both model checking (TLC) [3] and proof assistant tools (TLAPS) [4] for TLA^+. The proof assistant tools was used to mechanically prove the correctness of a Byzantine Paxos algorithm [5]. Raft, an easy-to-understand crash failure tolerance state machine replication protocol, is also checked via TLA^+ [6].

Recently, there are many interesting on-going projects which strive to build certified systems that also have practical performance. Verdi [7], a formally verifying distributed systems framework, provides with both verification and implementation environment for developing system applications. Project Everest [8], another ambitious project, aims towards a verified practical implementation of a drop-in replacement for the whole HTTPS (TLS/SSL) stack.

2 Formality Gap

TLA^+ is a temporal logic language that uses *formulas* to describe "actions", namely, the state transitions in a system. The plain first-order logic, as we know, does not capture any temporal semantics, and thus it is cumbersome or nearly impossible to model a step for the system. In TLA$^+$, the steps are modeled by formulas involving both primed and non-primed variables. The primed variables represent the new value for the same variable after this action, and by introducing lambda-term-like functions, TLA$^+$ is expressive enough to describe complex logic and protocols. The following formula stands for a sending action that changes the internal state rdy to $\text{rdy}' = 1 - \text{rdy}$ when the current rdy has the same value as ack. Also the new val$'$ could be any valid value in the constant set Data and ack stays unchanged.

$$\begin{aligned}\text{Send} \triangleq\ & \land\ \text{rdy} = \text{ack} \\ & \land\ \text{val}' \in \text{Data} \\ & \land\ \text{rdy}' = 1 - \text{rdy} \\ & \land\ \textsc{unchanged}\ \text{ack}\end{aligned}$$

From the above example, it is clear that protocols are "declaratively" specified. In other words, each action formula, although could be a complex composition of several other more basic expressions, consists of only two major parts in its semantics: the condition and the new state. The condition predicate part of the action plays a role of selecting potential states in the system's entire state space as the transition points, and working out the next state for each one of them. The power of such expression is an action could simultaneously and selectively generate multiple transitions and thus makes it possible to write a formal but simple and concise specification for a real world protocol. Also, it allows modeling non-deterministic transitions, so Byzantine behavior could be included.

However, it should be noticed that such expression is too "mathematical" (as emphasized by Lamport himself in the book) to encode the operational description of the protocol. In other words, this system only specifies what a protocol does, but specifies little on how it achieves that. It is still a great formal system that is heavily used in system industry and practically plays an important role in verifying protocol specifications. But as just discussed, it usually carries no information on the implementation of the algorithm behind the protocol, so in reality, the developers are still required to write the code according to their own understanding of the verified protocol specification. In the end, TLA$^+$ only provides with correctness of the specification, but not the implementation.

Therefore, the formality gap emerges: the possible discrepancy between the protocol specification (usually theoretical) and the actual operations (practical). The gap is caused fundamentally, according to the author's opinion, by the different requirements from theory and practice and the lack of an appropriate language that can be both formal and convenient to encode the operational logic.

The theory part of a distributed system usually assumes some ideal network or failure model, and also describes behavior of the system using a mixture of formal mathematical expressions and informal operational descriptions. Prior to (or even after) TLA$^+$, systems are mostly described in pseudo-code. Although pseudo-code may be enough for getting the system behavior understood, it is not formal enough to be mechanically checked. The nature of temporal relation between transitions in the system and internal states that a system holds make it difficult to create a clean and powerful language or system to formally describe them all. Luckily, as the programming language community keeps developing more powerful functional programming languages and formal methods, it has been made possible. The most recent works, such as Verdi and Project Everest, mainly use a unified language to both model the theory and code up the implementation.

The Verdi core system is built upon Coq, a proof system that supports multiple coding paradigms. The operational implementation could be directly implemented using Gallina, a simple but expressive functional programming language. Interestingly, the type system of Gallina also corresponds to the underlying set theory and constructive logic of vernacular or tactics languages used to write proofs. The correspondence, established by Curry-Howard isomorphism, is the crux of solving the formality gap in Coq.

The pipeline in Verdi has three stages. The first stage is the protocol designer implements the system using Coq (Gallina) under an ideal network or failure model purely defined in Coq. Therefore, it then becomes possible to prove desired invariants and the correctness of the written core protocol. When this is done in Coq, the Coq code is naturally divided into two parts: the part that is written in Gallina serving as a callable function (e.g. an event handler), and the part that serves as a formal proof for the theorems based on the implemented functions. Then, the two diverge their paths, so the implementation undergoes an extraction process by Coq to yield the runnable code in some other languages, and the proofs are mechanically checked in separation, using a minimal, trusted proof checker. In the end, some shims to the extracted core code are generated or hand written (usually small, trivial and less-prone to error) to direct I/O flows to the core. The novelty of Verdi is that it introduces a secondary stage where a written protocol could be transformed into another one with the same functionality and correctness, but more realistic (usually weaker) assumptions.

The idea of transformers makes sense in two aspects. On the one hand, in system design, the implementation usually uses several layers of abstraction to improve modularity and re-usability, which means entirely different systems are able to share the same ideal network assumption, which is guaranteed by some lower-level code that masks the packet duplication and loss in the real world. Such masking code is highly reusable because it does not try to follow any specific protocol, but just creates a generic abstraction layer to hide the actual network condition from the higher-level application. The Verdi transformer is just like it.

On the other hand, more importantly, it is the saver to developers using Verdi, because proving the same system with different assumptions could end up with very different choices of invariants and proof strategies. In some sense, the formality gap has not been exterminated but just turned into such pain of writing proofs using real-world, non-ideal network or failure models. Transformers, however, offer the remedy to alleviate the problem, and thus largely close up the gap.

Another system that is worth mentioning is Project Everest. The project aims at building a drop-in replacement of the entire HTTPS protocol stack using mechanically-checked implementation. Instead of Coq, a new language, F* is used to write the implementation and check the proofs. It is an ML-like functional programming language that is targeted to program verification. The type system provides with dependent types and weakest precondition calculus. The proof system in F* verifies the specification using a combination strategy of checking manual proofs and using SMT solvers. To achieve the ambitious goal, a language Low* is design to be a subset of F* to provide C-like imperative programming while preserving some high-level verification support. A compiler, KreMLin is built to translate Low* to C. The TLS-1.3 record layer is implemented in Low* and then proved for cryptographic security. Vale, a new domain-specific language for writing verifiable assembly is used to write cryptographic primitives that are provably correct. The early implementation is tested in `libcurl`, a widely used multi-protocol file transfer library. This project has multiple dedicated languages and compilers because of the performance consideration. It may not be so hard to solely eliminate the formality gap, but it is extremely difficult when the gap needs to be reduced without sacrificing much performance, especially for cryptographic algorithms that are usually time-critical.

3 A Set Replication System

The rest of the report presents the design and implementation of a small but complete distributed system that is mechanically checked and practically deployed, with little formality gap.

This little project is inspired by a blog post [9] by James Wilcoxone, the author of Verdi. The original post is about modeling and proving a replication system where each node maintains a local counter which could be monotonically increased. This project, instead, tries to explore the replication of a more complex state: a set of natural numbers. Despite the network semantics and the handler monad are borrowed from the counter tutorial as-is, the rest of the proof is entirely rewritten due to the change of the protocol. More lemmas are introduced to support the proof of the safety property. Notice unlike increasing a simple counter, adding the same element to a set is idempotent, which is allowed in our set replication system. Moreover, Coq code is finally extracted to Haskell code and made into a real world system where nodes communicate via TCP sockets.

3.1 System Description

The system implements a simple primary-backup replication scheme. There are two nodes: primary and backup. The environment could trigger an input event only on the primary which will make the state transition and replicate the change to the backup. In the application, both nodes maintain the local state of a set of natural numbers. The external input events could either be `add x` or `read`. The add event will insert the natural number `x` to the set maintained by the primary, and the read will trigger an output log entry on primary's trace. Once the primary gets an add event, it will add `x` to its local state and send a message to the backup to inform the change. When backup completes the insertion, it will acknowledge the replication by an `ack`, causing the primary to write a corresponding entry to its log.

The system assumes perfect network links: there is no corruption, duplication or loss in passing messages, and all sent messages are eventually delivered. However, the network is totally asynchronous, meaning the scheduling order of each node as well as the message interleaving could be arbitrary. Moreover, there is no guaranteed order of message delivery. This is usually referred as a "reliable network" model without FIFO in distributed systems literature.

Initially, both nodes start with the state of an empty set. The safety property proved in the project is that at any possible instant (world) of the protocol's asynchronous execution, the set maintained by the backup is always a subset of the one maintained by the primary.

The project finishes the definition and proof of the entire system in a single self-contained Coq source file `set_replication.v`, including the model of the network semantics, the protocol operations and the proof of the safety property. The handler code in Coq is also extracted into Haskell, and used as the core logic of the Haskell demo application which runs on an actual machine and uses TCP sockets to send and receive messages. The final demo is executable and gives an intuitive proof of concept that the specified and proved protocol will work in action, thus concludes the project.

3.2 Network Semantics

Due to asynchrony, the system is modeled in an event-driven style. The state of the entire system (world) is modeled as a set of states for each node (primary or backup), the in-flight messages that haven't been processed, and the written trace (log). The transition is triggered either by an external input event, or an internal message delivery. To be specific, the global state, is defined in Coq as follow:

```
Record world : Type :=
    { localState : node -> state;
      inFlightMsgs : list packet;
      trace : list externalEvent }.
```

Here, we use a function to represent the mapping from a node identity to its state. Therefore, the update to a node's state could be defined as follow:

```
Definition update (f : node -> state)
                  (n : node)
                  (st : state)
                    : node -> state :=
  fun m => if node_eq_dec n m then st else f m.
```

which uses a common technique that builds up nested if-then expressions to encode the lookup table. Then the transition could be defined as an inductive type in Coq:

```
Inductive reliable_step :
      world -> world -> Prop :=
| step_input :
  forall w i n st' ms outs,
    (processInput n i (localState w n) =
      (st', ms, outs)) ->

    reliable_step w (Build_world
      (update (localState w) n st')
      (ms ++ inFlightMsgs w)
      (trace w ++ [(n, inl i)]
              ++ record_outputs n outs))
| step_msg :
  forall w p st' ms outs,
    In p (inFlightMsgs w) ->
    (processMsg (dest p) (payload p)
               (localState w (dest p)) =
      (st', ms, outs)) ->

    reliable_step w (Build_world
      (update (localState w) (dest p) st')
      (ms ++ remove_one p (inFlightMsgs w))
      (trace w ++ record_outputs (dest p) outs)).
```

It is a small-step semantics for the network. Each step could either handle an input or message event. The function `processInput` and `processMsg` uniquely define the behavior of the protocol. The handlers take in the node identity (`h:node`), the current event (either `i:input` or `m:msg`) and return a dismonadic (isomorphic to the handler monad taking a unit type) version of a handler closure (`state -> state * list packet * list output`). Finally the small-step semantics use the new state, generated messages, and outputs to update the original world.

3.3 Protocol Specification

The reason of using a twisted return type for handlers is because they are written in a monadic style. In our set replication protocol, the handlers are as follow:

```
Definition processInput (h : node)
                        (i : input)
                        : handler :=
do
  match h with
  | primary =>
      match i with
      | request_add h => do_add h ;;
                         send backup (add h)
      | request_read => do_read
      end
  | backup =>
      match i with
      | request_read => do_read
      | _ => nop
    end
  end.

Definition processMsg (h : node)
                      (m : msg)
                      : handler :=
do
  match h with
  | backup =>
```

```
      match m with
      | add h => do_add h ;;
                 send primary ack
      | _ => nop
      end
  | primary => match m with
      | ack => out add_response
      | _ => nop
      end
  end.
```

The monadic handler specification allows an imperative-style protocol writing. This is very handy because a protocol is mainly a series of local state mutations based on their current values, which goes against the pure and functional programming style where values are always immutable. By wrapping up the mutations into monadic functions and introducing a Haskell-like monad notation, our set replication core protocol is cleanly specified as above. Of course, some efforts are required for defining such monadic handlers from scratch:

```
(* the monadic handler type *)
Definition handler_monad A :=
  state -> A * state
           * list packet * list output.
(* isomorphic to handler_monad unit *)
Definition handler :=
  state -> state * list packet * list output.
Definition do {A : Type}
            (m : handler_monad A) : handler :=
  fun s => let '(a, s', ps, os) :=
    m s in (s', ps, os).

(* the monad definition *)
Definition ret {A : Type} (x : A)
                : handler_monad A :=
  fun s => (x, s, [], []).
Definition bind {A B : Type}
           (ma : handler_monad A)
           (f : A -> handler_monad B)
           : handler_monad B :=
  fun s => let '(a, s', ps, os) := ma s in
        let '(b, s'', ps', os') := f a s' in
        (b, s'', ps ++ ps', os ++ os').

(* Haskell-style notation *)
Notation "x <- c1 ;; c2" :=
  (@bind _ _ c1 (fun x => c2))
    (at level 100, c1 at next level,
    right associativity).

Notation "e1 ;; e2" :=
  (_ <- e1 ;; e2)
    (at level 100, right associativity).

(* some basic operations *)
Definition nop := ret tt.
Definition send to msg : handler_monad unit :=
  fun s => (tt, s,
            [model.Build_packet to msg], []).
Definition out o : handler_monad unit :=
  fun s => (tt, s, [], [o]).
Definition get : handler_monad state :=
  fun s => (s, s, [], []).
Definition set s : handler_monad unit :=
  fun _ => (tt, s, [], []).

Definition do_add (h : nat): handler_monad unit :=
  x <- get ;; set (h::x).
Definition do_read: handler_monad unit :=
  x <- get ;; out (read_response x).
```

3.4 Utility Lemmas for States

It is still unclear how the set is represented in our formal development. For exercise purpose, the author decides to use Coq lists to represent sets in the replication system. Because a list could contain duplicates, it is not isomorphic to the represented set. For example, a list `1::2::3::nil` represents the same set as list `1::1::2::2::3::nil`. Interestingly, this seemingly meaningless artifact has its meaning in our set replication setting. The list exactly captures the log of updates (added numbers) to the empty set, but on the other hand, represents the updated set itself. Due to the non-isomorphism, we have to define the properties for our set representation:

```
Definition subset (l1 l2: list nat) : Prop :=
    forall (x: nat), In x l1 -> In x l2.
Definition eqset (l1 l2: list nat) : Prop :=
    (subset l1 l2) /\ (subset l2 l1).
```

Unfortunately, unlike natural numbers, Coq is unaware of the properties implied by our artificial definition of sets. The final Coq file spends some time proving a bunch of utility lemmas about the useful properties of the set representation. Nevertheless, this indicates an important stage of developing a mechanically-checked system where the simple, clean lemmas only relevant to the state representation could be proved to facilitate the final proof of any theorem, so that in the subsequent proofs, we do not have to repetitively prove some properties in a more convoluted environment.

3.5 Code Extraction

After finishing the proof of safety, according to our goal, the operational logic defined in Coq should be taken out to some practical languages and compiled into an executable. Coq provides with a simple-to-use command to do that:

```
Require Extraction.
Extraction Language Haskell.
Extraction "SetReplicationCore.hs"
            processMsg processInput.
```

This will let Coq generate a Haskell version of the selected functions and the minimal related code base that the functions depend upon. As a proof-of-concept, the author refrained from tampering the generated code, and decided to use it as-is. However, this means some additional conversion and wrapping are required. It is worth noticing that Coq uses Peano definition for natural numbers, in which the `nat` is represented by a nested `S` constructors terminated by an `O`. In order to show our final Haskell system in action, a conversion is required:

```
convert_nat :: SR.Nat -> Int
convert_nat SR.O = 0
convert_nat (SR.S x) = 1 + convert_nat x
```

Of course, the handlers generated by Coq are wrapped to fit into the real network environment. Here the author uses Cloud Haskell packages to provide with asynchronous TCP communication. The following code is a wrapper that direct I/O to the Coq generated handlers.

```
handlerWrapper :: (Show a) =>
    Node -> MVar State ->
    (SR.Node -> b -> SR.Handler) -> (a -> b) ->
    a -> Process [(Node, Msg)]
handlerWrapper node state handler
               builder feed = liftIO $ do
    s <- takeMVar state
    let SR.Pair (SR.Pair s' msgs') outs' =
            handler (build_node node)
                    (builder feed)
                    (build_state s)
    let ss' = convert_state s'
    let nmsgs = convert_packets msgs'
    putMVar state ss'
    printf "got: %s\n" $ show feed
    printf "new state: %s\n" $ show ss'
    printf "send: %s\n" $ show nmsgs
    printf "output: %s\n" $
            show (convert_outputs outs')
    return nmsgs
```

Finall the last code snippet shows how the actual events are dispatched from the network (`SR.*` is the imported module that is generated by Coq):

```
step :: Node -> MVar State -> Process ()
step node state = do
    outgoing <- receiveWait [
        match $ handlerWrapper node state
                  SR.processInput build_input,
        match $ handlerWrapper node state
                  SR.processMsg build_msg]
    forM_ outgoing $ \packet -> do
        let (node, msg) = packet
        case node of
            P -> nsendRemote primary_addr
                             pname msg
            B -> mapM_ (\b -> do
                          nsendRemote b pname msg)
                         backups_addr
```

3.6 Possible Enhancement and Issues

The current system only consists of two nodes: primary and backup. It is obvious that using the same protocol (with little modification), we could run a system with a single primary node but multiple backup nodes. The formal specification of node identity could be modified from:

```
Inductive node := primary | backup.
```

to

```
Inductive node := primary | backup: nat -> node.
```

To allow infinite distinctive backup nodes. The author attempted to do that, but it turned out that the generalization is non-trivial. There are several considerations: first, how to encode the membership of backup nodes into the monadic handler of protocol? If the primary does not know exactly how many backups are potential in the network, it won't be able to use a loop to replication the update to each one of them. We cannot assume an infinite membership because that would lose liveness (although safety is still preserved) by getting stuck in the infinite replication of changes. Then, if we use a finite, constant membership setting, how do we elegantly encode this extra constraint into the proof? Some proofs need to rewritten and carefully generalized in order to prove safety. Anyhow, due to the limited time, the author made some attempts to the modification, but failed to obtain a simple solution due to the above reason.

Another issue is about the performance. The memory consumption of the Coq generated handlers is much higher than a non-certified implementation directly in Haskell. This is because it takes constant space to store an integer (natural number) but linear space for Coq. In reality, because of the assumption that natural numbers are within 32-bit (or 64-bit) integer limit, it only takes 4 bytes or 8 bytes to represent each integer, even for $2^32 - 1$. However, to represent the same number using Peano system in Coq will incur constructor chain of length $2^32 - 1$, possibly eating up the entire memory. This issue might be solved by modeling the efficient representation in Coq.

Finally, the author could explore the possibility of using transformer from Verdi to change the network model into a less-ideal one, which will make this demo even more realistic.

4 Conclusion

The formality gap is a long-standing problem in developing mechanically-proved practical systems. However the gap has been rapidly closing up by the development of some promising programming languages and proof systems that are capable of efficiently defining the system operations in a formal way and then deriving proofs directly using the same language that defines them. Inspired by a blog post from Verdi, the author made a complete set replication system consists of two nodes, which could be both formally proved about its correctness and practically deployed.

References

[1] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[2] Leslie Lamport. Specifying concurrent systems with tla. *Preliminary draft, December*, 2000.

[3] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *CHARME*, volume 99, pages 54–66. Springer, 1999.

[4] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. Tla+ proofs. *FM*, 7436:147–154, 2012.

[5] L Lamport. Mechanically checked safety proof of a byzantine paxos algorithm, 2015, 2015.

[6] Diego Ongaro. *Consensus: Bridging theory and practice*. PhD thesis, Stanford University, 2014.

[7] 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. In *ACM SIGPLAN Notices*, volume 50, pages 357–368. ACM, 2015.

[8] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, et al. Everest: Towards a verified, drop-in replacement of https. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 71. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[9] James R Wilcox.