Seagull Verification Framework

yatharth.goswami@epfl.ch

December 2022

1 Overview

This document serves as a report for my work in understanding the Seagull Verification Framework during my time as a Semester Exchange Student at EPFL and as a research assistant at RS3 Lab, EPFL. I aim to write this in a concise and easy-to-understand manner so that anyone new to this framework (or field) can comprehend it.

2 Seagull Approach

The main problem with verifying concurrent programs is to reason about all possible thread interleavings. Recent works have utilised Concurrent Separation Logic [2] for this purpose. Seagull also relies somewhat on this concept, but it uses it differently. Firstly, I will focus on how Seagull can be used to verify use cases where a thread exclusively owns resources. Then, I will incorporate cases where read-sharing is also allowed. An example of a bank application has been provided to understand the concepts better. The bank application consists of series of accounts along with *query* and *transfer* functionalities. I will also like to make it clear beforehand what Seagull can and cannot do. It can be used to verify the functional correctness of the program (i.e. if the function returns the output will follow the given specification). It cannot prove termination of the program, or deadlock freedom, or liveness properties. Now, moving on to the first part involving exclusive ownerships, they propose the following steps for verification

2.1 Specification

Develop a specification for the program. This involves building a trusted high-level state machine which takes atomic steps. For the bank application, a specification could consist of a mathematical dictionary mapping accounts to the amount in them, along with rules for making steps in case of *query* and *transfer* requests.

The main point of verification now becomes to connect the implementation to specification. We want to prove that the implementation satisfies the specification. But in complex programs involving concurrency, it is tough to reason about all possible thread interleavings in the sophisticated implementation that a programmer might have written. Hence, directly proving implementation satisfying the specification is a next-to-impossible task. To handle this, they create an intermediate step first and call it the *Abstract View.*

2.2 Abstract View

We now develop an abstract view of the program in the form of a state machine. This should be complex enough to represent the program's concurrency and implementation logic but abstract away the exact details like threads, memory etc. For the bank example, if the implementation uses a hash map to implement the mathematical dictionary in the specification, then the abstract view will be a sequence of hash entries with query and transition logic. This part of the program is untrusted. Therefore, we must prove that the abstract view refines the specification developed before using frameworks like TLA+.

If you think of the problem now, it still pertains. Trying to prove that implementation matches the abstract view is still tricky as it is also prey to the same problem as specification. Let's think about the exact problem now. In the bank example, the implementation might give accounts named A and B to a thread which wants to transfer money between accounts A and B. In this case, this thread does not know the rest of the accounts. There might be another thread which has taken the lock for other accounts and changed the amounts inside them parallelly. This is not a problem with sequential implementations as, in that case, the whole program state is visible to a thread, and it can reason about facts about the complete program state at any instant.

2.3 Idea of Sharding

Now, we come to the key insight about Seagull. Their idea is that if a thread owns a resource exclusively, it can reason about only its state in an instant. Let's call this state (a part of the overall abstract view state machine) a shard. Concretely, a shard is a part of an abstract view that a thread can reason about with all authority. Now, if we soundly reassemble all the shards, we get the complete picture of the program. We can get the complete abstract view by developing a global state machine by assembling different shards. Once we get the global state machine, we can apply the standard techniques to investigate invariants. One might use state machine refinements to explain the program's interaction with external devices.

Note that since we want the shards of the program state to be combined in order to form global state machine, they naturally get encoded as elements of a Monoid. A commutative monoid is a set with a composition operation $(\cdot)$ which takes two elements of the set and returns a new element from the same set. This operator is called composition operator and is associative and commutative for a commutative monoid. Now, we formalize the above intuition below in the form of LTS (Localized Transition Systems) and GSM(Global State Machine).

2.4 LTS: Localised Transition System

Seagull introduces this to formalize the idea that a transition updates and depends on only a portion of state, while the rest of the state is irrelevant. LTS is a triple *(M, Init, τ_{local})*, where M is the state of states which form a commutative monoid, *Init* is the set of initial states and τ_{local} represents a local transition function, which takes in two elements from M and returns a boolean, as to if the transition between those two states is allowed or not.

In a practical sense, for defining an LTS, one needs to define the set of states (which should be dividable as shards) and the composition operation $(\cdot)$ to assemble two shards to form a bigger shard. Now, try to think about what a transition of the global machine would look like. Since we are in the realm of exclusive ownership of resources, any global transition would be composed of some local transitions inside different shards + some part of the global state remaining unchanged. This is how the global state machine gets defined.

2.5 GSM: Global State Machine

Suppose you have six accounts in your dummy bank application (say, *A, B, C, D, E* and *F*). Now, say you want to transfer some amount from account A to B. Parallelly, you would also want to transfer from account C to D. Also, your locking strategy has per-account locks. This means that one of the threads will take locks for accounts A and B, and another will take locks for accounts C and D. Therefore, one of the threads will have access to shards of accounts A and B and other threads will have access to shards of accounts C and D.

Say you reassemble these four shards before and after the transfers. Originally, their composition was state d, and after the transfer, their composition led to state d'. Say, the composition of shards for the rest of the bank application (*i.e.* accounts E and F) to be e. We can say the transfer was a valid transition step in global state machine iff

$$\tau_{global}(s, s') = \tau_{local}(d, d') \land (s = d \cdot e) \land (s' = d' \cdot e)$$

In general, the transition is said to be valid in the GSM iff

$$\tau_{global}(s, s') = \exists d, d', e \cdot \tau_{local}(d, d') \land (s = d \cdot e) \land (s' = d' \cdot e)$$

Intuitively, this connects the local transitions to global transitions in the sense that any global transition can be viewed as a series of local transitions (or a transition in LTS), keeping the rest of the state unchanged. Emphasizing again on the fact here that doing this reassembly is sound only when a thread has exclusive access of the shard which it is reading/writing. With this global state machine, we can prove things which would have been impossible seeing just the local view; for example - we can assert that after all transfers are done, the sum of the total money should remain constant. You can also prove various invariants about the program, being given this global view. However, for applying the state machine refinement technique, the authors say you need to connect this to the external interactions. This is because your trusted spec will have claims about the observable behaviour of the program, and from this GSM, we don't distinguish between internal and external actions. So, let's move on to remedying that now.

2.6 Abstracting External Interactions

If we want to make claims about the observable behaviour of our program, we need to establish a connection between our LTS and the observable behaviour. We must do this as we must refine our abstract state machine into the trusted program specification. The authors do this by just making the shards corresponding to the request (Request(id, c)) and response (Response(id, r)) as distinguished ones, different from the regular ones. For our bank example, we will have the request shards of the form of a map having the key as an *id* and value as a command, we want to perform (like $transfer(A, B, x)$ $\equiv$ Transfer x amount from account A to B). Similarly, for the response, we can have shards as a map from *id* to a value, where *id* is the same as request *id* and response can be the final amount in the account A. The *id* is required to confirm that the response is given to which request.

2.7 Connecting with Implementation

The next step is to prove that the implementation follows the abstract view (GSM) that we developed for the program above. Seagull uses a special ghost data type known as *Token* to do this. *Token* represents a shard of the overall state, and therefore, its values are elements from a commutative monoid of the LTS formed above. Intuitively, you can think of Tokens as something that is present in the implementation but is just a representative of the LTS. Since tokens are just representations of shards inside implementation, they will store a program state as well. Our main aim of showing the equivalence of the implementation to the GSM would be to show that the value inside the tokens (which gets updated using the LTS transitions) matches the corresponding values in the physical variables in the implementation at the starting point and ending point of the procedures. This is the planned way to establish equivalence between the implementation and GSM.

Tokens should behave just like shards in nature. For example - When you get a request from a client inside the LTS, you would add a new shard corresponding to this request to your old shard. Similarly, in the implementation, you would inject a new request token when you ask for a command to be executed. As for the response, when the program provides a response to the client, it removes a shard from the overall program state, i.e. the response shard. Similarly, inside the implementation, you would equivalently eject a response token when you return from the function. For performing internal transitions, you need to provide some trusted API for exchanging the old token (say d) with the new token (say d'), given $\tau_{local}(d, d')$ holds. The idea is to make, somehow, a connection between how a transition in LTS would look like and then try performing the same through tokens inside the implementation. This way, if the physical state of the program matches the ghost token state in the end, we can infer that the implementation obeys the abstract view or the global state machine transitions.

2.8 Ownership Concerns

If you think of the above solution, there is still a problem. If the tokens represent shards of the overall program state owned by different threads, then we should be able to reassemble or combine them at any stage to get a global state. For this to hold, it should be true that no token is owned by more than one thread at any instant. Another concern is what if the implementation produces its own tokens (to pretend a request occurred?) or duplicates the existing tokens (to retain the rights of using the tokens again).

To prevent these malpractices, Seagull employs a linear type system [1]. The idea is to make the ghost tokens linear to ensure that there is just one token owner. This also prevents the production as well as duplication of tokens. The only way to obtain tokens will be to go through the LTS transitions through trusted APIs.

2.9 Seagull Memory Management

Since Seagull relies on Linear Dafny [3] for verification and Dafny does not support shared memory, Seagull provides its own memory primitives to handle shared memory. It provides two memory primitives - Atomic Memory and Shared Memory which is data-race-free. It maps this memory model to C++ memory model. Let's now discuss the memory primitives in a bit detail and also how does it achieve the data-race-free shared memory primitive.

The Atomic memory is the usual atomic memory and supports atomic operations (like load, store, Compare-and-Swap etc.) Their Atomic primitive is basically like a lock, in the sense that it allows us to store some ghost state along with physical state and tie these two using some invariant known as atomic invariant. Whenever you do some operation on the atomic memory, you need to check that the atomic invariant holds after doing the operation by modifying the ghost token stored in the atomic memory appropriately. Another thing to note is that Atomics support sequentially consistent ordering only.

The other type of memory is called a *Cell*. Seagull enforces that *Cell* is data-race-free by associating a ghost *Permission* token to it. This is an idea from Separation Logic to enforce ownership. Multiple threads can have the reference to the *Cell* memory at a time, but to read/write to that memory, they need that special ghost *Permission* token. The fact that the memory remains data-race-free relies on the fact that *Permission* token is held by one thread at a time, which they enforce using their linear type system.

2.10 Building a Simple Spin Lock

Another thing to note is what is the semantics for the passing of these tokens, i.e. How would one thread relinquish the right to that memory and pass this right to another thread? According to their model, Atomic allows its tokens to be passed freely, while Cell does not allow this directly. Therefore, generally, the Permission token for the Cell is to be passed using an Atomic. This also allows us to build a simple spin lock using the Atomic and Cell memory primitives. The idea is to store the resource inside the Cell and keep its Permission token inside some Atomic. The Atomic will physically store the exclusive flag boolean as to whether the lock is free. The threads trying to get the resource will have the reference to the Cell but not the permission token. To get that, they will check the Atomic memory for if the lock is free. If it is, the atomic will pass the ghost Permission token to that thread and update its physical value to correspond to the lock taken. While releasing the lock, the thread has to deposit the ghost Permission token back into the Atomic.

2.11 Verifying Custom Read/Write Tools

Seagull authors claim to have developed a new logic called Burrow [4], which allows them to separate the verification of application logic and synchronization primitives. I have not completely grasped this part as to how they can achieve this. Still, I will continue explaining how they have approached the verification for any synchronization primitive as such in isolation to application logic.

From the previous discussions, it would be clear that the way we do concurrent verification is that we will have some ghost state. We need to tie this ghost state to the physical state, with the ghost state transitioning according to a trusted transition system. Now, many applications would require the physical state to be held in Reader Writer Lock manner, and thus it becomes evident to build a system to allow sharing the ghost state in such a manner. For this purpose, Seagull also has introduced a concept of *shared* variable along with *linear* variables. Unlike **linear** variables, these **shared** variables allow us to alias the same object (or ghost variable) in a read-only manner. There are rules that let you turn a linear (exclusively owned) object into a shared (shared borrow) object, and they make sure that these shared objects expire before exclusive access is regained to the original.

The main API for the reader-writer lock then becomes something like this.

1. On doing a shared acquire, a thread, in turn, gets a linear *handle*, which allows it to get a shared version of the data inside the lock.
2. There is a function *borrow_shared* which allows you to get the value inside the lock in a shared variable on being given a handle (as a shared type, so that you can still call it afterwards) which represents a sort of permission to get the resource.
3. On calling release, the handle is required to be provided so that you can't retain the rights of obtaining the value inside the lock even after calling release.

The correct usage of the above API is also ensured by the linear type system. Notice, since we get, in return, a linear handle on calling acquire shared and the only way to consume this handle fully will be to call release on it. Hence, a linear type system ensures the correct usage by requiring the release to be called after an acquire and the release not before an acquire. Another exciting thing is that once the linear handle is destroyed, the data inside the lock held by the thread will also be destroyed, as it was a shared reference.

2.12 Need of Sharded State Machine

One might wonder, if we already have memory primitives to enable shared and linear variables, why would one not verify the Reader Writer Lock with a similar strategy as a spinlock? The authors say that Rust also has this type-system in it, but still, their Reader Writer Lock has memory-unsafe code present, and they also run into similar problems if they try to verify the reader-writer lock in such a manner. They have also provided an unverified reader-writer lock file in rust inside their codebase [5], where the use of unsafe memory cells is evident.

This leads the authors to develop a new strategy similar to the above-explained strategy. In short, we would like to develop a sound LTS for this first. Let's think of how this LTS should look like. Say you want to store some ghost state into a Reader-Writer lock, and you would want the data to be stored inside a Cell memory; This means that a thread calling acquire exclusive should get the Permission token, and the thread calling release exclusive should return back this Permission Token, the thread calling acquire shared will lead to physically increment a reader count inside the library and in return should get some linear handle which acts as a token to allow it to call *borrow_shared* and obtain the data inside the cell in a shared variable. The release shared caller should provide the linear handle it obtained through acquire to complete the process and should lead to a decrement in reader count physically.

In conclusion, you can notice that an LTS for this should support the deposition and withdrawal of ghost tokens (like the permission token above) and should allow transitions to produce a token/handle shard on calling acquire shared and consuming it on calling release shared. This is just what the guard protocols are all about. We're going to define another ghost state (a Sharded State Machine, in particular), S, with some special rules: it is possible to "deposit" a ghost state into S and "withdraw" that state back out. Furthermore, it is possible to "borrow" a ghost state from the S-state without actually withdrawing it. The methodology requires the programmer to prove (using invariants of the Sharded State Machine) that borrowed state always corresponds to some deposited state/token.

So, for example, we could implement the above interface by creating a Sharded State Machine S, with certain shards enabling borrowing. The shared handle objects could be defined as that state or tokens corresponding to these states. This way, we can build an LTS corresponding to the functionalities we want and work on its verification, just like in the first half. This summarises the section on Guard Protocols in the paper written in a very tough-to-comprehend cryptic style. Enthusiastic readers can try to map the summary with the section in the original Seagull Paper.

3 Conclusion

This project was an attempt to understand the Seagull Verification Framework. The original paper was not precisely to the point and might be challenging to comprehend for anyone new to this field. After reading this report, the main ideas of their work should be better understood. Seagull, through its sharding strategy and linear type system, gives an optimistic technique which could be used in a vast range of concurrent systems. It is a general framework, and many concurrent systems could fit into their verification strategy in a not-so-hard manner. I tried, through this report, to develop an intuition for their technique and why it should work. One might want to dig deeper into the field of Concurrent Separation Logic and their prior work on Burrow Logic to understand their methodology more clearly.

4 Acknowledgements

I want to thank Yugesh Kothari, for helping me throughout this project, providing valuable suggestions and lending his precious time to weekly fruitful discussions. Being completely new to the field of verification, I would like to emphasize that there were times when I could not see any future for myself in understanding this framework and its large codebase in Dafny. Thanks to Yugesh, I kept the motivation to struggle with the framework and finally get something out of it. Next, I would like to thank Vishal Gupta and the people from RS3 Lab, EPFL for being so welcoming and sharing their input on my curiosity related to systems and research (in general). Finally, I would like to thank Prof. Sanidhya Kashyap for providing me with the opportunity of working on this project at RS3 Lab, EPFL, even though I had no prior introduction to this particular field and for providing with the necessary means for the smooth running of the project.

References

[1] Philip Wadler. “Linear Types can Change the World!” In: *Programming Concepts and Methods.* 1990.

[2] Peter W. O’Hearn. “Resources, concurrency, and local reasoning”. In: *Theoretical Computer Science* 375.1 (2007). Festschrift for John C. Reynolds’s 70th birthday, pp. 271–307. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/j.tcs.2006.12.035`. URL: `https://www.sciencedirect.com/science/article/pii/S030439750600925X`.

[3] Travis Hance et al. “Storage Systems Are Distributed Systems (so Verify Them That Way!)” In: *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation.* OSDI’20. USA: USENIX Association, 2020. ISBN: 978-1-939133-19-9.

[4] Travis Hance et al. *Burrow: Custom Read/Write Permissions for Custom Ghost State in Concurrent Separation Logic.* URL: `https://www.cylab.cmu.edu/_files/pdfs/tech_reports/CMUCyLab21002.pdf`.

[5] *Unsafe Reader-Writer lock implementation in Rust.* URL: `https://github.com/rs3lab/concurrency-verification/blob/seagull-stable/concurrency/rwlock/rwlock_unverified.rs`.