Argosy: Verifying Layered Storage Systems with Recovery Refinement

Tej Chajed
MIT CSAIL, USA
tchajed@mit.edu

Joseph Tassarotti
MIT CSAIL, USA
jtassaro@andrew.cmu.edu

M. Frans Kaashoek
MIT CSAIL, USA
kaashoek@mit.edu

Nickolai Zeldovich
MIT CSAIL, USA
nickolai@csail.mit.edu

Abstract

Storage systems make persistence guarantees even if the system crashes at any time, which they achieve using recovery procedures that run after a crash. We present Argosy, a framework for machine-checked proofs of storage systems that supports layered recovery implementations with modular proofs. Reasoning about layered recovery procedures is especially challenging because the system can crash in the middle of a more abstract layer's recovery procedure and must start over with the lowest-level recovery procedure.

This paper introduces *recovery refinement*, a set of conditions that ensure proper implementation of an interface with a recovery procedure. Argosy includes a proof that recovery refinements compose, using Kleene algebra for concise definitions and metatheory. We implemented Crash Hoare Logic, the program logic used by FSCQ [8], to prove recovery refinement, and demonstrated the whole system by verifying an example of layered recovery featuring a write-ahead log running on top of a disk replication system. The metatheory of the framework, the soundness of the program logic, and these examples are all verified in the Coq proof assistant.

CCS Concepts • **Theory of computation → Program verification**; • **Hardware →** *System-level fault tolerance.*

Keywords Kleene Algebra, Refinement

ACM Reference Format:
Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Argosy: Verifying Layered Storage Systems with Recovery Refinement. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3314221.3314585

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06.
https://doi.org/10.1145/3314221.3314585

general background on the storage systems

1 Introduction

Storage systems, including file systems, databases, and persistent key-value stores, must protect data from loss even when the computer crashes (e.g., due to a power failure). These systems provide crash-safety guarantees about what data persists if such crashes occur. To achieve these guarantees, many systems perform some form of repair in a recovery procedure that runs after a reboot.

Storage systems are typically structured into several layered abstractions. For example, a storage system might use several physical disks for redundancy. By replicating writes across these disks, the storage system can implement an interface presenting a single synchronous disk, and then use write-ahead logging to implement a transactional API for atomically writing multiple disk blocks (see Figure 1).

an extended example illustrating layering and recovery

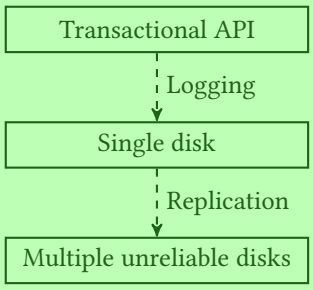

Figure 1. A simple storage system that uses recovery at multiple layers of abstraction.

If the computer crashes, write operations may have occurred on only some of the physical disks. To repair its state, the storage system runs a recovery procedure after reboot. First, it propagates missing writes to the remaining disks to restore replication. Then, it reads the transaction log to determine if transactions need to be aborted or applied, based on whether the system crashed before or after they were committed. The storage system may have to run the recovery procedure several times, because the system can crash again during recovery.

Secondary contribution: not essential to solving the problem, but an interesting technique

Problem statement

Because a storage system needs to handle crashes at any time,[1] implementing and testing them is difficult. Storage systems in practice have had bugs that resulted in data loss and data leaks [16, 17, 23]. Since these bugs are costly, formal verification is attractive because it can rule out large classes of bugs. For verification to scale to modern, complex storage systems, the proofs for the implementations in each layer should be independent. This independence is hard to achieve because crashes during one layer of abstraction's recovery procedure requires re-running all of the recovery procedures of lower levels. For example, with the system in Figure 1, a crash in the middle of the write-ahead log's recovery procedure may leave disks out-of-sync, which requires starting over with the replicated disk's recovery.

This paper presents Argosy, a framework for verifying storage systems that supports layered recovery procedures with modular proofs. Argosy introduces the notion of *recovery refinement*, a set of proof obligations for an implementation and recovery procedure. These obligations are sufficient to guarantee clients observe the specification behavior, including with multiple crashes followed by recovery. Furthermore, recovery refinement *composes* between two implementations: this allows the developer to prove each implementation separately and then obtain a proof about the whole system with a general composition theorem. We describe the metatheory behind recovery refinement in section 4. The framework is encoded in the Coq proof assistant, with machine-checked proofs of soundness.

Main contribution

There are several existing systems that support reasoning about crashes and recovery, particularly in the context of file-system verification [7, 8, 11, 26, 28]. Most have no support for layered recovery, since they consider only a single recovery procedure at a time. The Flashix modular crash refinement work [11] does consider layered recovery, but to simplify proofs recovery procedures cannot rely on being able to write to disk. Argosy supports *active recovery* procedures which write to persistent storage; both the replicated disk and write-ahead log implementations rely on active recovery. Furthermore, the metatheory for a number of existing systems is based on pen & paper proofs, whereas Argosy has machine-checked proofs for both the metatheory and example programs.

To prove recovery refinement within a single layer, Argosy supports a variant of Crash Hoare Logic (CHL), the logic used in the FSCQ verified file system [7, 8]. Argosy generalizes FSCQ's CHL by supporting non-deterministic crash behavior, whereas FSCQ modeled only persistent state and assumed it was unaffected by a crash. The main benefit of using CHL is that as long as recovery's specification satisfies an *idempotence* condition, the developer can reason about recovery using only its specification and ignore crashes during recovery.

Secondary contribution: was introduced in prior work

To simplify the definition of recovery execution as well as facilitate proofs of Argosy's metatheory for recovery refinement, we formulated the execution semantics and recovery refinement using the combinators of Kleene algebra [18]. Kleene algebra is well-suited for this purpose because it models sequencing, non-determinism, and unbounded iteration, which arise naturally when reasoning about crashes and recovery.

As a demonstration of Argosy, we implemented and verified the storage system of Figure 1. The disk replication and write-ahead log are separately verified using CHL, each with its own recovery procedure; section 6 details how this proof works in CHL within Argosy. We then compose them together to obtain a verified transactional disk API implemented on top of two unreliable disks. The composed implementation extracts and runs, using an interpreter in Haskell to implement the physical disk operations at the lowest level.

The paper's contributions are as follows:

1. Argosy, a framework for proving crash-safety properties of storage systems that introduces *recovery refinement* to support modular proofs with layered recovery procedures.
2. Machine-checked proofs in Coq of the metatheory behind recovery refinement that are simplified by appealing to properties of Kleene algebra.
3. An implementation of Crash Hoare Logic (CHL) for proving a single layer of recovery refinement, which we use to verify an example of a storage system with layered recovery.

Summarize contributions in order of importance

This paper's introduction orients the reader, even though this requires a lot of space. It introduces general background before going into any details on the solution. It first orients the reader (assumed to be in programming languages) to the general problem of writing storage systems that tolerate crashes. Next, it gives an extended example that leads to the core problem: "...the proofs for the implementations in each layer should be independent. This independence is hard to achieve because crashes during one layer of abstraction's recovery procedure requires re-running all of the recovery procedures of lower levels."

[1] In this work we use "crash" to refer to the entire storage system halting and requiring restart, such as due to a power failure or kernel panic.