Reducing Crash Recoverability to Reachability

Eric Koskinen
Yale University, USA

Junfeng Yang
Columbia University, USA

Abstract
Software applications run on a variety of platforms (filesystems, virtual slices, mobile hardware, etc.) that do not provide 100% uptime. As such, these applications may crash at any unfortunate moment losing volatile data and, when re-launched, they must be able to correctly recover from potentially inconsistent states left on persistent storage. From a verification perspective, crash recovery bugs can be particularly frustrating because, even when it has been formally proved for a program that it satisfies a property, the proof is foiled by these external events that crash and restart the program.

In this paper we first provide a hierarchical formal model of what it means for a program to be crash recoverable. Our model captures the recoverability of many real world programs, including those in our evaluation which use sophisticated recovery algorithms such as shadow paging and write-ahead logging. Next, we introduce a novel technique capable of automatically proving that a program correctly recovers from a crash via a reduction to reachability. Our technique takes an input control-flow automaton and transforms it into an encoding that blends the capture of snapshots of pre-crash states into a symbolic search for a proof that recovery terminates and every recovered execution simulates a crash-free execution. Our encoding is designed to enable one to apply existing abstraction techniques in order to do the work that is necessary to prove recoverability.

We have implemented our technique in a tool called ELEVEN82, capable of analyzing C programs to detect recoverability bugs or prove their absence. We have applied our tool to benchmark examples drawn from industrial file systems and databases, including GDBM, LevelDB, LMDB, PostgreSQL, SQLite, VMware and ZooKeeper. Within minutes, our tool is able to discover bugs or prove that these fragments are crash recoverable.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Software/Program Verification—Model checking; Correctness proofs; Reliability—D.4.5 [Operating Systems]: Reliability—Verification; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—Program analysis

General Terms Verification, Theory, Reliability

Keywords Crash recovery, formal verification, program analysis

1. Introduction
Storage systems such as file systems, databases, and version control store valuable user data, so their correctness is paramount. A major issue that these systems must handle is crash-recovery: the machine may crash at any unfortunate moment, regardless of what operation is in flight and what data is being mutated. A storage system must correctly recover from these potentially very broken states and retain as much useful data as possible. These kinds of bugs can be particularly frustrating because, even when it has been formally proved for a program $P$ that $P = \varphi$, the proof is foiled by these external events that crash and restart the program. Some recent efforts [7, 8, 19, 26] notwithstanding, this space remains largely unexplored: little backbone has been developed for understanding what it means for a program to correctly recover from a crash from a verification perspective.

The first part of this paper provides a hierarchical specification of crash recoverability. The idea is that, after a crash, a re-executed program should behave the “same” as the original uncashed program from an initial state. More precisely, a proof of recoverability means that, after a crash, the program eventually will reach a state that is observationally equivalent to (simulates) some state that was visited before the crash (i.e. in the trace’s prefix). We formalize this notion of recoverability, which is general enough for proving correctness of programs that use real-world durability protocols.

Next, we introduce a novel technique capable of automatically discovering proofs of crash recoverability of unmodified programs. Our core algorithmic technique is to reduce the crash recoverability problem to reachability. For an input program $A$ described as a control-flow automaton [24], we describe a novel transformation into an encoding $A^{\triangledown}$ such that

$$\exists D. A^{\triangledown} \text{ cannot reach error } \Rightarrow A \text{ is crash recoverable}$$

where $D$ is an indexed set of termination arguments. The encoding $A^{\triangledown}$ symbolically tracks snapshots of pre-crash states and, after a crash, attempts to align the crashed program with one of these previously saved uncashed program snapshots in order to show simulation. $A^{\triangledown}$ reaches error if either (i) recovery does not terminate or (ii) recovery does not lead to a state that is observationally equivalent to some state in the pre-crash prefix. In this way, a proof of automaton non-reachability entails a proof of recoverability. Thus, we can leverage existing abstraction techniques (interpolation, abstraction refinement, termination argument refinement, etc.) to automatically reason about the recoverability of a program.

We have implemented this technique in a tool called ELEVEN82 that is able to prove recoverability of C/C++ programs. While our formal definition of recoverability is over arbitrary transition systems, our implementation is based on control-flow automata [24], as they have been successful at verification of industrial programs [5]. Our tool is able to prove recoverability for programs that implement complex filesystem algorithms such as write-ahead locking.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

POPL’16, January 20–22, 2016, St. Petersburg, FL, USA
© 2016 ACM, 978-1-4503-3549-2/16/01...$15.00
http://dx.doi.org/10.1145/2837614.2837648

*11-82 is the police scanner code for “Accident - No Injuries.”
logging [25] and shadow paging [20]. We have applied our implementation to benchmark examples (based on Fig. 4 of [27]) taken from several industrial databases, virtual machine monitors, and distributed consensus servers including GDBM, LevelDB, LMDB, PostgreSQL, SQLite, VMware and ZooKeeper. Our tool is able to automatically discover (known) recoverability bugs or prove recoverability of these examples, and it is typically able to do so within a minute or two.

In summary, our principal contributions are:

- A hierarchical inductive/co-inductive definition of crash recoverability (from $N$- to $\infty$-recoverability) (Sections 3 and 4).
- A novel technique for automatically proving recoverability via a reduction to reachability (Section 5) and a soundness proof thereof (Section 6).
- An implementation (Section 8) called ELEVEN82, the first tool capable of automatically proving recoverability of programs.
- An experimental validation on two recovery paradigms and several examples taken from real systems (Section 9).

**Limitations.** Our automated technique is able to prove that a program correctly recovers from at least one crash. Our theoretical development outlines a stronger hierarchy of recoverability properties, including $N$-recoverability and $\infty$-recoverability. Techniques for proving these higher-order recoverability properties are left for future work. Our model also does not directly capture recovery that leads to states that are outside the trace prefix but it may be possible to model such examples by modifying the program.

2. Recoverability and Storage Systems

The problem of crash-recovery in storage systems, for example, is fundamentally rooted in their hierarchical design. Magnetic disks are durable: data stored on disks persist across power cycles. However, disks are orders of magnitude slower than volatile memory. Thus, storage systems strive to keep data in memory as long as possible for speed, before they have to flush the data to disk. In addition, the disk durability interface is quite limited, typically providing only two primitives: (1) a primitive that guarantees the atomic write of a disk sector at the moment of a power failure and (2) a wait primitive that returns only after a given section is written durably to a sector of a disk at the moment of a power failure and (2) a wait primitive that returns only after a given section is written durably to disk.

Storage applications in turn build more complex functionality on top of them, and must correctly use the abstractions to sync data to disk. These implementations have complex designs so as to carefully write out the modified disk sectors in appropriate orders and, consequently, provide ad hoc or poor crash-recovery guarantees and are ripe with bugs [16, 23, 27, 33–36].

Systems researchers and practitioners have long fought this crash-recovery challenge. They have created techniques such as shadow paging [20], write-ahead logging in ARIES [25], and soft updates [18], for safely persisting data on disk. These techniques have been adapted and used in the Linux ext3/ext4 file systems [32].

**Testing.** The systems community has also developed testing tools for crash-recovery. FiSC [34, 35] and eXplode [33, 36] systematically enumerate through many possible crash scenarios and check that a system correctly recovers from each of them. The core ideas in these systems are borrowed from model checking except that they check code directly without requiring a formal specification of a checked system. While this practical design enables FiSC and eXplode to find many serious errors in real-world storage systems, it also limits their rigor and guarantee. SQCk [23] advocates the idea that file system recovery utilities should be implemented using declarative languages. Alice [27] extracts applications’ durability update protocols by running the applications and simulating many different crash scenarios. Results from Alice show that different file systems provide very different durability guarantees and that many serious data-loss errors are considered “unfixable” by developers.

**Formal Methods.** The critical nature of file system warrants formal analysis of their implementations.

Concurrent with our work, Chen et al. [7] describe their efforts developing the first verified filesystem, and a specification logic called Crash Hoare Logic (CHL). Their work, like ours, is concerned with the recoverability problem and, in particular, for filesystems with asynchronous disk operations. From a broad perspective, our works are complementary: Chen et al. focus on verifying the filesystem, while our work is aimed at verifying user-level programs, with the assumption that the underlying filesystem has already been verified. From a technical perspective, the main difference is that we focus on automation while they focus on proof modularity/rewusability. Our work is fully automatic: users do not need to provide specifications nor write any proofs. By contrast, Chen et al. require user-provided CHL specifications and do not have fully-automated proofs. The user needs to be involved in completing the proof obligations. In an earlier short paper from (mostly) the same authors [8] they discuss several ways of specifying crash recoverability properties but, again, do not provide any techniques for automatically verifying recoverability.

Ntzik et al. [26] describe a novel program logic for expressing crash recoverability, in which the volatile and persistent (durable) state are tracked separately. Unlike any prior work, their logic even has support for concurrency. Their work does not discuss automation and, thus, many of the distinctions of our work to Chen et al. mentioned above also distinguish us from Ntzik et al.

Finally, Gardner et al. [19] describe a logic for reasoning about tree structures, such as the POSIX file system and Ridge et al. [30] provide a specification of the POSIX filesystem. These works are orthogonal to ours; while they do not directly pertain to proving crash recovery, they provide a rigorous separation between applications and OS implementations.

In recent years, techniques have emerged for proving temporal properties of imperative programs (or automata derived therefrom) [4, 12–14, 17]. While there is some similarity (i.e. mix of safety and liveness), to our knowledge recoverability cannot be expressed in temporal logic (not even with auxiliary variables). Simulation has been reduced to safety in other contexts such as SMV’s refinement layer proofs [10] and Mocha’s simulation proofs [1]. Our transformation produces an encoding that has two copies of the program in synchrony, barring some similarity to self-composition in information flow security [3, 31].

The fundamental distinction between volatile and persistent storage means that recoverability is a concern even in the absence of concurrency. That is, the recoverability problem here is orthogonal to the atomicity problem in concurrency and the fault tolerance problem [6, 29].

**Challenges.** Storage systems have some of the most complex code for implementing recovery features that are crucial for functionality and performance. These features present many open challenges for formal verification. First, to verify a program, one needs to have the code of the program. This most basic scope question is difficult for a storage system because crash recovery often involves operations done by multiple programs (such as the filesystem recovery tool) in addition. In addition, since crashes are user-visible events, the recovery can involve users and administrators. All of these operations must be modeled.

Second, storage systems typically adopt a multi-layered design that includes (from bottom to top): disks whose firmware can have 400K lines of code [2], device-specific disk drivers in the OS, generic block device drivers to abstract over disks, file systems,
generic virtual file system layers to abstract over file systems, and storage applications such as MySQL. It is already challenging to verify that an individual layer is crash recoverable. For end-to-end crash recoverability, we have to verify recovery of the entire storage stack. Work on formalizing these layers (e.g. CertiKOS [21]) does not currently address the crash recoverability problem.

Third, for performance, storage systems heavily use delayed writes that reach disks asynchronously. For instance, Linux ext3 file system flushes out modified file data every 30 seconds, and its write-ahead log every 5 seconds. If a crash happens before these timers are fired, the file system is permitted to lose data. Thus, the specification language and proof system for crash recoverability must allow reasoning over time.

Finally, storage systems heavily rely on concurrency for performance. While currency is already a notoriously difficult problem for verification, the problem is even more challenging when combined with crash recoverability. Existing concurrency verification approaches are not immediately applicable because invariants that hold without crashes may no longer hold.

3. Examples and Program Model

We now introduce our treatment of the crash recoverability problem with a few examples, given in Figure 1. In later sections we will formalize recoverability and give our reduction to reachability.

3.1 What Is a Crash?

We work with programs that run on some platform which, at some nondeterministic point, may crash the execution. Consider the diagram of program locations in Example 1 in Figure 1. The first execution (in blue) begins at node 0, proceeds through node 1 to node 2. At this point, the environment/platform crashes the program, and the program is restarted for a second execution (in green). To make things more concrete, one might think of a personal computer and the following events:

1.Boot machine  
2. Establish program environment  
3. Execute the program  
4. Crash mid-execution  
5. Re-Boot machine  
6. Execute recovery script  
7. Establish program environment  
8. Re-execute the program  

We will consider the first two events to establish the initial state (as in other verification contexts). The crash is some kind of external event and, for concreteness, we will say that it havoc's volatile storage, but maintains persistent storage (the formalism does not require this definition of a crash). After a crash, there is a recovery process which involves re-establishing a (potentially new) initial state, perhaps with the help of a recovery script. Finally, the program is re-executed. In practice, a storage application may come with a program for normal operation and a recovery utility for recovering from crashes. Moreover, some recovery operations are done manually by administrators. In this paper, we combine these separate elements into one program for clarity.

Consider the code in Figure 1. Example 1. This is a simplified example of a command-line tool that manipulates the filesystem using standard POSIX operations: open, read, rename, etc. Initially we will assume that the file input is on disk. After opening input, it reads data into a buffer. Imagine that, at this point, it crashes and is re-executed from the beginning. In this example there should be no problem re-executing the program, largely because the program is read-only with respect to persistent storage: it does not modify the filesystem. The re-execution behaves exactly the same as the original execution, had it not crashed (we will assume there are no other programs modifying the filesystem).

But what happens if the program does modify persistent storage? Consider Example 2 which, after opening the input file, creates an output file and writes a byte to it. The second open operation has the O_CREAT flag, telling the OS to create the file if it does not exist, and open it for write only (for now, assume that the gray O_TRUNC flag is not used). The first new subtlety here is that, for performance, this program does not use synchronous file system operations: operations open, write, rename, etc. may not immediately make it to disk. These operations return to the user and then later are asynchronously written from the system libraries down to the disk. Although this does improve performance in most cases, it means that the user program is “out of sync” with the file system. We can model this behavior as a cross-product between the program code and the internal filesystem operations which nondeterministically writes data from memory to disk. (We omit illustration of this cross-product to keep these examples concise.)

Returning to Example 2, if the program crashes on Line 3, the disk will be left in one of three possible states: output does not exist, output exists but is empty, or output exists and contains a single byte. Taking the second or third case, we can visualize what happens with the Example 2 diagram: after the crash, the program is re-executed in a state that is not exactly the same as the initial state because output will exist on disk. When re-executed, how can we say that the program behaves correctly? This leads us to a working hypothesis of this paper:

Recovered programs should not introduce behaviors that were not present in the original (unchrashed) program.

With the possibility of crashes comes the possibility that new behaviors will be introduced during re-execution. Programs that are able to properly recover from a crash need to account for these new situations that may arise: when the program is re-executed, some operations will have become persistent and volatile data will be lost. By including the O_TRUNC flag in Example 2, we make the program crash recoverable: when re-executed, the program overwrites the incomplete modifications from the first execution.

Informally, we might think of the trace semantics \( [P] \) of a program \( P \), and use \( \hat{P} \) to indicate a program that is identical to \( P \) except that it may crash (at any point) and be re-executed. We would like to show that \( [P] \subseteq [\hat{P}] \) modulo a simulation relation. In particular, we will search for observational equivalence. The \( \geq \) symbols in Example 2 illustrate the simulation relation that we would like to find. A key benefit of this approach to crash recoverability is that we can use the original program as the specification for how the program should behave in the context of crashes.

Non-determinism. Consider Example 3 in Figure 1. Here, after the output file is opened, the program nondeterministically decides whether to write A or B to output. Perhaps the first execution will go from Line 2 to Line 3, whereas the re-execution will go to Line 4. Certainly this should be acceptable, but adds another subtlety: the re-execution of the program need not simulate the first execution but, rather, may simulate some other feasible execution of the original program.

3.2 Recovery Scripts

Thus far, we have considered examples that merely involved re-execution of the program. The last two examples increase the expressiveness of the recovery process.

While the previous examples were written to carefully tolerate unfinished changes to persistent storage, another route is for the recovery script to restore it to the initial state (or similar thereto). In Example 4, we have changed the program, so that it performs
the `creat(output)` operation rather than using `open`. This is acceptable because there is also a recovery script that, after a crash, removes the output file if it exists on disk. Without the recovery script, this program would not be crash recoverable.

But do we always have to recover to the initial state? No: filesystems and databases would be horribly inefficient if they always returned to a start state after a crash. Instead, we would like to support checkpoints, as illustrated in Example 5. POSIX-like filesystems offer an operation called `fasync` which can be used to force operations to disk. These operations are like barriers: they ensure ordering constraints. `fasync(π)` blocks the program until all operations on file f have been written to persistent storage.

This facility is helpful in establishing a checkpoint, as shown on Line 2. This command `fasync_commit` is pseudo-code for atomically performing the `fasync` and setting a flag `committed` to true. If the program crashes after a checkpoint (e.g. on Line 4), the recovery script can restore to the checkpoint rather than the beginning of the program. In this example, the RECOVER script checks whether committed is true, opens the output, and resumes to Line 3.

### 3.3 General Form

This final Example 5 nearly illustrated the most general form of programs in our setting. Recoverability proofs will have the following form for each trace:

![Diagram](image)

This diagram illustrates a trace of a system that begins at an initial state $σ_0$, steps through $σ_1$ and so on, via $σ_2$ to $σ_n$. At $σ_n$, a special crash transition occurs, taking the system to some $σ'$, from which point recovery is needed. The recovery, if correct, eventually leads to some state $σ''_k$ that is observationally equivalent to (i.e. simulates) a state $σ_k$ that was in the trace prefix before the crash occurred. Hence forward, the system behaves as it would if it had not crashed.

We found that this definition was suitable to cover a wide variety of examples taken from the industrial systems that we considered.

There is yet another complication: what happens if there are multiple crashes? In particular, what happens if the recovery script crashes? In the next section we give a specification of recoverability that encompasses all of these issues.

## 4. Specifying Crash Recoverability

We now establish preliminary definitions and then define a hierarchy of crash recoverability.

### States and transition systems.

A labeled transition system $M = (Σ, Δ, Γ, σ_0)$ is over state space $Σ$, labels $Δ$, transitions $Γ \subseteq Σ \times Δ \times Σ$, and initial state $σ_0$. We denote a single transition as $σ \xrightarrow{λ} σ'$. A trace $π$ of machine $M$ is a sequence $σ_0 \xrightarrow{λ_0} σ_1 \xrightarrow{λ_1} σ_2 \ldots$ such that $∀ i \geq 0, σ_i \xrightarrow{λ_i} σ_{i+1}$. We will use the notations $σ_0 \xrightarrow{λ_0, λ_1, \ldots} \ldots$ and $σ_0 \xrightarrow{λ_0, \ldots} \ldots$ where $Λ_λ$ is a sequence of transitions, when we don’t need to bind names to intermediate states. For convenience and w.l.o.g., we will work only with infinite traces. We use the notation $Π(M, S)$ to mean the set of all traces of $M$ from $S$, and we use $Π(M)$ to mean $Π(M, \{σ_0\})$. We also will use the notation $σ \xrightarrow{π}$ when we want to talk about the prefix $σ \xrightarrow{λ_0, \ldots}$ of a trace $σ \xrightarrow{π}$. We use $π^n$ to denote the first state of $π$, and $π_n$ to be the $n$th suffix of $π$ (the latter binds tighter than the former in $π^n$). As usual, we say that a simulation relation is a relation $⟩$ on states, defined coinductively:

\[
∀ σ_1, σ_1', λ_1 λ_2 σ_2, σ_2' \rightarrow σ_2' ∧ λ_1 \xrightarrow{λ_2} σ_2' ∧ σ_1' \leq σ_2'
\]

\[
σ_1 \leq σ_2
\]
Later in this paper, we will describe our automatic technique in which transitions will correspond to abstract method operations. We will use arc labels $\lambda_1$ and $\lambda_2$ to denote operation executions, and equality between $\lambda_1$ and $\lambda_2$, denoted $\equiv$, when the results of the operations are equivalent. We will work with transition systems that have a special crash label $\lambda_{\text{crash}} \in \Lambda$.

**Recovery.** A robust system will want to prepare for the possibility that a crash may occur, so it will also have recovery labels that are, for convenience, not disjoint from $\Lambda$. Recovery attempts to bring the system back to a “normal state” $\sigma$. However, it may not bring it back to exactly the same state that would have been reached. Instead, it may bring the system back to a state $\sigma_k$ that simulates $\sigma_i$. For the purposes of this paper we focus on recovery that returns the system to a state $\sigma_k$ that simulates a state $\sigma_i$ in the current trace’s prefix before the $\lambda_{\text{crash}}$ transition. Formally, recovery $\mathcal{R}_N$ of a trace is defined inductively as:

$$
\sigma \xrightarrow{\lambda \pi^0} \pi \xrightarrow{\lambda \pi} \mathcal{R}_N(\pi)
$$

$$
\exists k < n, j \geq 0, \pi_j \preceq \sigma_k \land \mathcal{R}_{N-1}(\sigma_0, \lambda_{0\rightarrow\lambda_{k-1}}) \rightarrow \mathcal{R}_N(\sigma_0, \lambda_{0\rightarrow\lambda_{k-1}}) \rightarrow \sigma_n, \lambda_{\text{crash}} \rightarrow \pi
$$

where, among the $\lambda$ labels, only $\lambda_{\text{crash}}$ is a crash arc (a may have crash arcs). The base case $\mathcal{R}_0$ indicates that $\pi$ recovers from at least 0 crashes, and thus has no premises. The second rule says that $\sigma \xrightarrow{\lambda \pi} \pi$ recovers safely from $N$ crashes, provided that $\lambda$ is not a crash, $\sigma \xrightarrow{\lambda \pi} \pi^0$, and the trace $\pi$ recovers from $N$ crashes. The third rule allows us to derive $\mathcal{R}_N$ from $\mathcal{R}_{N-1}$. We splicel out a portion of the trace that causes $\sigma_k$ to “then and” recovers to a similar state $\pi'_0$. This can be visualized as:

![Diagram of recovery process](image)

The crashed trace $\sigma_0, \sigma_1, \ldots, \sigma_k, \sigma_n, \ldots$ is recovered, provided that (i) the first element of $\pi$ (i.e. $\sigma'_1$) simulates $\sigma_k$, denoted $\pi^0 \preceq \sigma_k$ and (ii) we can say that $\pi$ has $(N-1)$ recovered.

This inductive definition applies when there are only finitely many crashes. For example, if there are $N$ crashes, the rule on the right can be used $N$ times and then, with all crash arcs removed, the rule on the left can be used. Note also that this definition permits crashes that occur during recovery. The following co-inductive variant can be used when there are infinitely many crashes:

$$
\sigma \xrightarrow{\lambda \pi^0} \mathcal{R}(\pi)
$$

$$
\exists k < n, j \geq 0, \pi_j \preceq \sigma_k \land \mathcal{R}(\sigma_0, \lambda_{0\rightarrow\lambda_{k-1}}) \rightarrow \mathcal{R}(\sigma_0, \lambda_{0\rightarrow\lambda_{k-1}}) \rightarrow \sigma_n, \lambda_{\text{crash}} \rightarrow \pi
$$

Crashy programs have a “doomed” piece of computation: some states are visited (above, $\sigma_k, \ldots, \sigma_n, \ldots$), and then this computation is erased. This is similar to the notion of zombie transactions in a software or hardware transactional memory (TM). In the TM community, it is recognized that there are cases (e.g. dependent transactions [28]) when it is acceptable for a transaction to observe the effects of a zombie transaction and yet also more rigorous correctness criteria (e.g. opacity [22]) that forbid such things. We believe a similar paradox exists here. Consider systems code that is performing filesystem operations, mutating things such as log files along the way. What happens if this code crashes? It is the authors’ opinion that side-effects such as these log messages should (morally) not be allowed. However, we also acknowledge that often they are harmless.

With the above definitions of $\mathcal{R}$ and $\mathcal{R}_N$, we can define an overall notion of crash recoverability of a transition system: labeled transition system $M$ is $N$-recoverable denoted $\mathcal{R}_N(M)$ provided $\forall \pi \in \Pi(M)$ that $\mathcal{R}_N(\pi)$. $M$ is $N$-recoverable denoted $\mathcal{R}_N(M)$ provided $\forall \pi \in \Pi(M)$. $\mathcal{R}_N(\pi)$. For the rest of this paper we focus on 1-recoverability; techniques for proving higher-order recovery are left to future work.

5. Automation

We now give an automatic technique for discovering proofs of crash recoverability by reducing the problem to automaton reachability. We will describe a novel transformation of a given input program automaton $A$, into a new automaton $A^\mathcal{R}$ which symbolically encodes a search for a proof that recovery terminates and that new behaviors are not introduced.

5.1 Control-Flow Automata

We focus on programs described as control-flow automata (CFA) [24], as they are quite general and have been used as the basis of tools that are able to verify significant industrial examples [5].

**Definition 5.1** (Control flow automaton [24], A (deterministic) control flow automaton $A$ is a tuple $(Q, q_0, X, \delta, \rightarrow)$ where $Q$ is a finite set of control locations, $q_0$ is the initial control location, $X$ is a finite set of typed variables, $\delta$ is a set of operations and $\rightarrow \subseteq Q \times \delta \times Q$ is a finite set of labeled edges.

We work with labels of the form $Y = m(X)$ where $Y \subseteq X$ is a vector of variables returned from the execution of $m$. These are the observations made by an operation, which we will later use in our definition of observational equivalence. We define a valuation of variables $\theta : X \rightarrow Val$ to be a mapping from variables names to values. Let $\Theta$ be the set of all valuations. An $m \in \delta$ is a name of a method call (operation). The notation $\text{exec}(\theta_1, m, \theta_{i+1}, Y)$ means that executing method $m$, using the values given in $\theta_1$, leads to a new valuation $\theta_{i+1}$, mapping variables $X$ to new values, and returning observations via variables $Y$. We assume that for every $\theta_1 \in \Theta$ and every $m \in \delta$ that $\text{exec}(\theta_1, m, \theta_{i+1}, Y)$ is computable in finite time, and that $\theta_2 \in \Theta$. One may discharge this assumption via a termination analysis [15].

Arcs are required to be deterministic. This is not without loss of generality. We support nondeterminism in our examples by symbolically determining the input CFA: whenever there is a non-deterministic operation $m$, we can augment $X$ with a fresh prophecy variable $\rho$, and replace $m$ with a version that consults $\rho$ to resolve nondeterminism. Similar symbolic determination is done elsewhere [12].

A run of a CFA is an alternation of automaton states and valuations: $r = q_0, \theta_0, q_1, \theta_1, q_2, \ldots$ such that $\forall i \geq 0$, $\exists m, q_i \rightarrow r_{i+1} \land \text{exec}(\theta_i, m, \theta_{i+1}, Y)$. Again, for convenience, we assume only infinite runs (this is not without loss of generality). We denote as $\Pi(A)$ the set of all runs of $A$. We say that CFA $A$ can reach automaton state $q$ provided that $3q_{\rho_0, \theta_0, q_1, \theta_1, \ldots} \in \Pi(A)$ such that there is some $i \geq 0$ such that $q_i = q$.

For CFA $A = (Q, q_0, \delta, \rightarrow)$, with valuation space $\Theta$, we define the induced trans. system $M^\mathcal{R}_\Lambda = (\Sigma, A, \Lambda, \Gamma, A, \sigma^\mathcal{R}_\Lambda)$ as:

$$
\Sigma_A = Q \times \Theta,
$$

$$
\Lambda_A = \delta_{\mathcal{R}},
$$

$$
\Gamma = \{ (q_1, \theta_1, m, \{q_2, \theta_2\} | q_1 \rightarrow m \land \text{exec}(\theta_1, m, \theta_2, Y) \})
$$

$$
\sigma^\mathcal{R}_\Lambda = (q, \theta_0)
$$

We will simply write $M_A$ when we mean $M^\mathcal{R}_\Lambda$. For a run $r = q_0, \theta_0, q_1, \theta_1, \ldots$ of $A$, we define the induced trace to be $\pi_r = \{ (q_i, \theta_i) | q_i \rightarrow q_{i+1} \}$. For a run $r = q_0, \theta_0, q_1, \theta_1, \ldots$ of $A$, we define the induced trace to be $\pi_r = \{ (q_i, \theta_i) | q_i \rightarrow q_{i+1} \}$.
(q₀, θ₁), m₀, (q₁, θ₂), m₁, ..., where ∀i ≥ 0, ∃mᵢ, mᵢ → mᵢ₊₁ ∧ exec(θᵢ, mᵢ, θᵢ₊₁, Y). Finally, for a run r we define N-recovery of the run Rᴺ(r) to be N-recovery of the induced trace of r. That is: Rᴺ(τᵢ). Likewise for Rᵢ.

As discussed earlier, our implementation works with programs that perform file operations (open, read, etc.) on an asynchronous file system that may lazily shuffle these operations from mem to disk. To this end, we construct a cross-product CFA between an input program and the filesystem's transition system. This cross-product has built into it an implicit distinction between volatile and persistent components of the state. That is, the variables in X can be partitioned into Xᵦ ⊔ Xₚ where: Xᵦ = {mem, x, y, z, ...} and Xₚ = {disk} where x, y, z are program variables.

Reciprocity scripts. We will work with CFAs that have a certain structure pertaining to recovery. When the CFA has recovery code that is intended to recover to automaton state qᵢ, then the crash arc will lead to a special automaton state q̂₀. (In implementations this is tantamount to marking checkpoints.) From q̂₀, recovery will then, if it operates correctly, lead throughly recovery-oriented states Qᵦ ⊆ Q and eventually reach state qᵢ ∈ (Q \ Qᵦ). When we wish to analyze an automaton that does not have this structure, we can impose a trivial version of this structure that does no recovery by adding a skip arc from q̂₀ to q₀.

5.2 The Algorithmic Reduction

Transformation of an input control-flow automaton A into a new CFA A⁺ = E(A, crash, D) is given in Figure 2. It is designed to reach q̂₀ if it is possible for a post-crash recovery to diverge or lead to a state that is not observationally equivalent to some prefix state. Consequently, a (non)reachability proof for A⁺ entails crash recoverability of A. Qᵤ is given by Q but augmented with an error state qᵦ and intermediate states qᵢ, q̂₀ is simply q₀, and there are several duplicate sets of variables for X: variables for per-node snapshots Xᵦ, variables for a loaded snapshot X̂ and variables for the termination check X̂. Q₀ can be derived from the definition of E which, intuitively, involves the following:

1. Symptomatic snapshots (L1): The transformation introduces a command that captures a symptomatic snapshot X̂, of X on each arc entering each state qᵢ (L1). These X̂ variables record the state that was visited when a run passed through qᵢ. Later, if recovery happens to return to qᵢ, the values of X̂ can be loaded into X which will then be used as part of the mechanism for ensuring observational equivalence.

2. Non-deterministic crash (LA): Crash arcs are introduced from each qᵢ to q̂₀. A run of the CFA may nondeterministically choose to take these arcs which execute the crash and begin recovery from q̂₀. At this point, flag CR is set and snapshot X̂ is loaded into X. The unindexed variables X are duplicated to X̂ for use in (L5).

3. Simulation check arcs (L2): On a run in which a crash has occurred, we must make sure that the run of the automaton over these crashed/recovered variables X̂ simulates a crash-free run of the automaton from qᵢ whose saved state is recalled into the X variables. To this end, we create duplicate arcs, guarded by {CR}, in which the op is performed on both the crashed/recovered variables X and the loaded snapshot X̂. An arc is then added to a waypoint qᵢ.

4. Error states for simulation violation (L3): We introduce arcs from these waypoints qᵢ to qᵦ whenever the return value Y of performing m(X) is not equivalent to the return value Y of performing m(X) arcs to qᵢ otherwise.

5. Eventuality (L5): We ensure that recovery code eventually leads to the destination with a known transformation [15] that introduces another set of variables X which may, at any nondeterministic moment in time, be duplicated from X. An arc is added to qᵦ whenever (X̂, X) is not in well-founded D.

The valuations arising from runs of A⁺ involve multiple components (arising from the symbolic snapshots). We will use the notation: [B, θ, φ, θ, θ₀, θ₁, ...] where the first component is a boolean value for CR, the second component is the valuation θ for the loaded snapshot X̂, the third component is the valuation θ used for the X̂ variables, the fourth component is the valuation θ used for the X̂ terminal variables, the fourth component is the valuation θ used for the main X variables, and the remaining components are the valuations of snapshot X̂ variables. Thus, a run rₑ of the encoded machine will be of form

rₑ = (q₀, [false, 1, 1, θ₀, 1, 1, ...], (initial config.)
q₁, [false, 1, 1, θ₁, θ₀, 1, ...],
q₂, [false, 1, 1, θ₂, θ₀, θ₁, ...],
...)

6. Soundness

We now give some helper lemmas along with our soundness result. We will abbreviate post-crash valuations, as [true, θ, θ], focusing

Figure 2. The formal transformation from input automaton A to A⁺. Non-reachability of qᵦ entails crash recoverability of A.
only on the restored snapshot \( \theta \) and the crashed/recovered \( \theta \), omitting the other snapshots (note that the valuation \( \theta \) is needed for Lemma 6.3 but, for simplicity, not displayed here).

**Lemma 6.1.** Every configuration of \( \mathcal{A}_D^P \) has a successor.

**Proof.** By the axiom that every configuration of \( \mathcal{A} \) has a successor and by construction of \( \mathcal{A}_D^P \). □

**Lemma 6.2 (Simulation).** For all \( A \) and corresponding encoding \( \mathcal{A}_D^P \), if \( r \in r_e \) beginning at config. \( (q_0, \{ \text{true}, \theta_0, \theta \}) \) that \( r \) cannot reach \( q_{\text{err}} \), then \( (q_0, \theta) \not\sim (q_0, \theta_0) \) where these induced states are w.r.t. respective transition systems \( ^M = \langle Q \times \theta, \delta_{e}, \Gamma_{A}; (q_0, \theta_0) \rangle \) and \( M = \langle Q \times \theta, \delta_{e}, \Gamma_{A}; (q_0, \theta) \rangle \) both induced by \( A \).

**Proof.** Let notation \( (q_0, \ldots, \theta) \Rightarrow q_{\text{err}} \) mean that \( q_{\text{err}} \) cannot be reached in any run from \( (q_0, \ldots, \theta) \). We prove the lemma by coinduction with the following hypothesis:

\[
\forall q_0' \, q_1', \theta_0, m, q_0 \xrightarrow{m} q_1,\quad q_0', q_1', \theta_0 \not\sim q_{\text{err}} \Rightarrow (q_0', \theta_0) \not\sim (q_0, \theta_0)
\]

First we specialize \( \forall \) for CFA-induced transition systems. Consider an arc \( q_0 \xrightarrow{m} q_1 \) in \( A \). For this \( q_0, m, q_1 \) we know by construction (L2, L3, L4) that \( \mathcal{A}_D^P \) has the arcs:

\[
q_0 \xrightarrow{(Y \cdot m)(X); \cdot Y \cdot m(X)} q_{00}, q_{01} \xrightarrow{Y \cdot \theta} q_{10}, q_{11} \xrightarrow{Y \cdot \theta} q_{1\text{err}}.
\]

Since \( (q_0', \theta_0, \theta_0) \not\sim q_{\text{err}} \) and by Lemma 6.1, configuration \( (q_0', \theta_0, \theta_0) \) must have a successor (and a successor’s successor). We now know state \( \theta_0 \) and \( \theta_0 \) such that \((\theta_0, m, Y, \theta_0)\) and \((\theta_0, m, Y, \theta_0)\) and \( Y = \theta_0 \). Moreover, \( \mathcal{A}_D^P \) can reach \( (q_1, \text{true}, \theta_1) \) and \((\theta_0, \theta_0, \theta_0)\) can’t reach \( \text{err} \), neither \( \text{err} \) can be reached from \( (q_1, \text{true}, \theta_1) \).

We now let \( j = 1 \) and apply the inductive hypothesis to this \( q_1, \theta_1, \theta_1 \), obtaining that \( (q_1, \theta_1) \not\sim (q_1, \theta_1) \). Applying the definition of \( \not\sim \), we obtain that \( (q_0', \theta_0) \not\sim (q_0, \theta_0) \). □

**Lemma 6.3 (Eventuality).** If there exists \( D \) such that encoding \( \mathcal{A}_D^P = \mathcal{E}(A, \text{crash}, D) \Rightarrow q_{\text{err}} \), then for every \( q_k \in Q \), \( M^M \) eventually reaches some \( (q, \theta) \) where \( q \in Q_{\text{crash}} \).

**Proof.** Straight-forward application of Thm 3 in [15]. □

**Theorem 6.4 (Soundness).** For CFA \( A \) and crash script \( \text{crash} \), if there exists \( D \) such that encoding \( \mathcal{A}_D^P = \mathcal{E}(A, \text{crash}, D) \) cannot reach \( q_{\text{err}} \), then \( A \) is 1-recoverable.

**Proof.** We must show that for all \( A \) and crash script \( \text{crash} \), \( \mathcal{A}_D^P = (Q, q_0, X, \delta, \epsilon, \cdot) \) and the transition system \( M_A \) induced by \( A \), that

\[
\begin{align*}
&\text{(C1) ~} A_D^P \text{ cannot reach } q_{\text{err}} \quad \\
&\text{(C2) ~ } \forall q_k \in Q, M^M_{q_k} \text{ eventually reaches some } (q, \theta) \quad \\
&\quad \text{on Run } \mathcal{R}_k(M_A) \quad \\
\end{align*}
\]

(C2) is given by Lemma 6.3. Unfolding \( \mathcal{R}_r \), we must show that for every trace \( \pi_r \in \Pi(M_A) \), that \( \mathcal{R}_r(\pi_r) \), where the trace \( \pi_r \) is induced by run \( r \) of \( A \). Pick such a given run \( r \). If \( r \) does not involve a crash, then the proof is trivial. Otherwise, the induced trace of a run of \( A \) will be of the form:

\[
(q_0, \theta_0, \ldots, (q_i, \theta_i), \lambda_{\text{crash}}, (q_{i+1}, \theta_{i+1}, \ldots, (q_j, \theta_j), \ldots)
\]

To derive \( \mathcal{R}_r \), we must show that:

1. There exists \( K \leq n \) such that ...
2. ... there exists a \( J > n \) such that \( (q_j, \theta_j) \not\sim (q_k, \theta_k) \).

(The requirement that \( \mathcal{R}_r \) hold of the "snapped" trace is trivially satisfied.) By construction of \( \mathcal{A}_D^P \), there is a corresponding run of \( \mathcal{A}_D^P \) with the following form, using the left column to indicate the index number of the run:

\[
\begin{align*}
0. & \quad (q_0, \{ \text{false}, 1, \theta_0, \} ), m_0, \quad \text{(By L1)} \\
1. & \quad (q_1, \{ \text{false}, 1, \theta_1, \} ), m_1, \quad \text{(By L2)} \\
2. & \quad \ldots \\
M. & \quad (q_m, \{ \text{false}, 1, \theta_m, \} ), m_M \quad \text{(By L5)} \\
N+1. & \quad (q_n, \{ \text{true}, \theta_n, \} ), m_{N+1} \quad \text{(By L4)} \\
N+2. & \quad \ldots \\
N+M. & \quad (q_{N+M}, \{ \theta_{N+M}, \} ) \quad \text{(By L4)}
\end{align*}
\]

We are given the value of \( K \) by the crash arc between \( n \) and \( n + 1 \), which indicates the state we hope to recover to. So we can let \( K = k \). Since this run \( r \) contains a crash, there are finitely many steps to this \( K \) in \( r \) and, by construction, finitely many steps to \( K \) in the run of the encoding \( r_e \).

Now we must find \( J \). In \( r \), from \( (q_n, \theta_n) \) we go to \( (q_k, \theta_n) \). Condition C2 ensures that from \( (q_k, \theta_n) \), run \( r \) reaches some \( (q_k, \theta_n, \ldots) \) after some \( m \) (finitely many) steps. By construction \( r_e \) also reaches some configuration \( (q_k, \{ \text{true}, k, \theta_n, \ldots, \} ) \) after \( m \) steps.

We let \( J = n + 1 + m \). What remains to show is that \( (q_j, \theta_j) \not\sim (q_k, \theta_k) \). By condition C1, \( r_e \) cannot reach \( q_{\text{err}} \), and so from \( (q_k, \{ \text{true}, k, \theta_n, \ldots, \} ) \) we cannot reach \( q_{\text{err}} \). By Lemma 6.2, \( q_{\text{err}} \) also reaches some configuration \( (q_k, \{ \text{true}, \theta_ne, \ldots, \} ) \) after \( m \) steps.

Note that \( q_1 = q_k \). Rewriting we obtain \( (q_1, \theta_1) \not\sim (q_1, \theta_1) \). □

7. A Complete Example

We will now describe our technique on a more realistic example that is based on the useradd utility which adds a new user to a UNIX-like system:

```c
int fd = open("/pw"); if(fd<0) return -1;
char *buf = read(fd);
close(fd);
if(!strstr(buf,"joe",3)) { 
d = readdir("/u");
int pw2 = creat("pw2");
append(pw2,buf);
append(pw2,"joe");
ffcntl(pw2,D);
close(pw2);
unlink("pw");
rename("pw2","pw");
mkdir("/u/joe");
} 
```

103
This utility modifies an important user/group configuration file: `pw` (i.e. `/etc/passwd`), whose corruption can be extremely serious and prevent users from logging in. This example checks whether a new user `joe` in `pw` (Lines 1–4) and, if not, creates a copy of the file (Line 6), appends a new entry (Lines 6–8), flushes the changes to disk (Line 9) and then snaps the new file into place (Lines 11–12). Finally, a directory is created for `joe` (Line 13). We elide details about checking return values for errors and simplify some operations. A control-flow automaton [24] representation of this example is shown in Figure 3. Version 1. We refer to nodes as $\ell_i$, for example, we will refer to nodes as $\ell_0$, $\ell_1$, etc. For now, disregard the large bold dashed arrow. Remember that the file system operations may happen asynchronously. In the above example, `fsync` before $\ell_2$ ensures that all of the data has been copied (read/written) from `pw` into `pw2` and that `joe` has been written to `pw2`.

**Transformation.** The transformation, when applied to the example results in the automaton $A_D^D$, which is displayed in Figure 4. New code is highlighted in rounded-corner boxes and new nodes/ars are given by double lines. The transformation consists of the following steps, where the enumeration numbers correspond to the numbers in clouds in Figure 4:

1. At the entry arcs to each node, we create a symbolic snapshot copy of the state by introducing a new set of duplicate variables for that particular node. For the arc entering $\ell_0$, for example, we have added code that creates the 0th copy. This copy includes all variables and, in fact, a deep snapshot of `disk` and `mem` which we will refer to as disk0 and mem0.

2. We transform the recovery, denoted $[\text{RECOVER}, ()]_D$ using an existing technique [15] that ensures $[\text{RECOVER}, ()]$ will terminate by checking its inclusion in a given well-founded relation $D$ and, if the inclusion does not hold, takes an arc to error.

3. Our encoding introduces the possibility of a crash by adding a nondeterministic outbound crash() arc from each program node such as the double-line arrow from $\ell_1$ to $\ell_0$. crash() havocs the unindexed variables fd, pw, buf, and mem, sets a flag $\text{CR}$ to true, and runs the recovery RECOVER() that is appropriate for recovering to $\ell_i$.

4. These crash/recovery arcs also load symbolic snapshots, by duplicating the appropriate snapshot into a set of pre-primed variables. The 0th snapshot is loaded by the commands `pw=pw0; `disk=disk0; `mem=mem0;`.

5. To encode the search for a simulation, operations are performed on the original program variables (which represent traces that have crashed and recovered) and, since $\text{CR}$ is true, the pre-primed variables as well (which represent a previously-saved execution that did not involve a crash). Note that the command `open`, for example, manipulates `disk` and `mem` rather than `disk` and `mem`. Arcs to the error state are added and taken whenever these operations return inequivalent values (e.g. if `fd=f\overline{d}`).

There is a run $r$ of this encoding $A_D^D$ that can reach $q_{\text{run}}$. The run $r$ passes through $\ell_8$, follows the $\text{CR}$ arc, and then leads to a state at $\ell_0$ in which `pw` is not on disk. All initial states of the original program have `pw` on disk. Therefore in the next arc, as shown in Figure 4, `fd` becomes -1, while `fd` becomes a non-negative integer and so the run goes to $q_{\text{run}}$ since $\overline{fd} \neq fd$.

**Subsequent versions.** In Version 2 of Figure 3, we have attempted to correct the bug by removing the `unlink` before the `rename` operation. In this case the program still may not recover correctly from a crash. The `rename` operation switches a directory entry from an old file to a new file. The problem is that, while `rename` happens atomically with respect to all concurrent file operations, it is not...
atomic with respect to system crash. Consequently, trouble lurks in subsequent code that may assume that the new file is guaranteed to be on disk. In Version 2, a crash at $\ell_1$, may leave the $\mathit{pw}$ file unchanged but then, surprisingly, $\ell_0$ may be in the directory entry $d$ returned by $\text{readdir}$. This sublety of bugs such as this one in Version 2 is not explained in the POSIX specification, and becomes a source for many serious data loss bugs [27, 33].

Version 3 illustrates the fix for this bug: the solution is to add a $\text{sync}$ operation that forces the $\text{parent}$ directory (and any modified directories along the path of the file) to be written to disk at $\ell_8$, thereby ensuring that the directory entry for $\mathit{pw}$ has been written to disk. For clarity, we use $\mathit{sync}$ to represent a sequence of operations that open a file’s parent directory, call $\text{sync}$ on the directory, and close the version. This version is crash recoverable.

8. Implementation

We have developed ELEVEN82, which is capable of automatically proving crash recoverability of C programs. ELEVEN82 is built upon CPAchecker [5], which constructs a control-flow automaton [24] from input programs written in C or (soon) Java. Our prototype implementation transforms input programs via C preprocessor macro replacements for $\text{libc}$ operations. These macros instantiate a model file system (discussed next), create the cross product with the asynchronous operations, and perform the transformation described in Section 5.

Model of the Filesystem. Our specification and recoverability reduction theoretical work (which is over transition systems and control-flow automata, respectively) has no problem supporting complex structures such as trees, lists, etc. However, our implementation relies on an underlying solver and, while some of them have support for trees with a fixed maximum depth, trees of arbitrary recursion depth present many difficulties. CPAchecker, for example, does not currently understand tree structures.

We now describe how we permit reasoning over trees by approximating them with the theories of arrays and linear arithmetic — this was sufficient for the examples that we saw. See Figure 5. First, we track files as the indexes of the $\text{mem}$ and $\text{disk}$ arrays. That is, if integer variable $\mathit{fn}$ is non-negative, then it corresponds to a file. The in-memory contents of the file $\mathit{fn}$ are given by the value $\mathit{mem}[\mathit{fn}]$, and likewise for disk. One can think of these values as version numbers. For example, $\text{create}(f\mathit{n})$ establishes a new file, whose contents is represented by an initial version INIT,VER. We also track whether or not the parent directory of file $\mathit{fn}$ has been $\text{syncd}$ to disk, via the boolean flag $\mathit{fn}.\text{sync}$. This abstraction gives us a reasonably expressive filesystem model so CPAchecker can generate the needed invariants.

Currently, due to the way we have implemented ELEVEN82 with compiler macros, it does not support nested directory structures. However, since directory structures can be approximated with arrays, one can include support for nested directories as follows. First, represent filenames with their fully qualified pathname (e.g. $\mathit{foo/bar/file}$). Then use a series of arrays to track the parents of a given directory, such as:

- PARENT[:$\mathit{foo/bar/file}$] = /foo/bar
- PARENT[:$\mathit{foo/bar/file}$] = /foo
- PARENT[:$\mathit{foo/bar/file}$] = ROOT
- PARENT[:$\mathit{foo/bar}$] = /foo

The model implementation of $\text{create}$, $\text{open}$, etc. will then manipulate these arrays as appropriate.

Asyncronous disk. The macros for each filesystem operation (e.g. $\text{read}$) in Figure 5 additionally perform the cross-product with the asynchronous disk, execute the read operation on the unindexed variables and (if $\_\text{CR}$) on the snapshot variables and check equivalence. The macros insert code that runs MAY$\_\text{ASYNC}$. This method iterates through the file system, nondeterministically shuffling files from $\text{mem}$ to $\text{disk}$, by setting $\text{disk}[i]$ to be a version number that is greater than its current value, but at most the version in $\text{mem}[i]$. Additionally, the method iterates through each file, nondeterministically deciding to swap the $\text{sync}$ flag from false to true. Finally, to simulate a crash, each file system macro involves the script MAY$\_\text{CRASH}()$, which havocs the local variables, values in $\text{mem}$, and eliminates any $\mathit{fn}$ for which the $\text{sync}$ flag has not been set ($\mathit{fn}:=\mathit{-1}$). Since our model of the file system distinguishes between file data in memory versus disk, we can verify assertions unrelated to crashes such as “is the contents of file $X$ the same in memory as on disk.”

Optimized cross product. We employ a few optimizations in our encoding. First, we only collect snapshots at program locations that are the targets of the recovery. Second, we only need instances of MAY$\_\text{ASYNC}$ just after a filesystem operation. Intermediate control-flow and local variable statements are not affected by transitions in the asynchronous file system. The trickiest part of our implementation were the macros for creating snapshot versions of variables, that are assigned before operations and loaded when recovery returns to the relevant program location. After running the C processor, our macros expand to a form that is fit to be analyzed by CPAchecker.

Quantified invariants. Our model of the filesystem is built on the theory of arrays and so our implementation depends crucially on tool support for reasoning in this theory. Usually, arrays require support for quantified invariants which are difficult and, for example, not part of CPAchecker. As such, filesystem recoverability provides another compelling example to motivate the need for better
support for array reasoning. Our current implementation avoids the need for quantified invariants by working with statically defined arrays (being careful with cpa.predicate.maxArrayLength) and creating snapshots by iterating over the disk memoria, copying elements to the corresponding snapshot version. Thus, we end up with predicates for each array slot (e.g. snapA mem[0] == mem[0], snapA mem[1] == mem[1], etc.). This was sufficient for the examples we worked with.

9. Evaluation

Benchmarks. We evaluated our technique and implementation ELEVEN82 by applying it to 32 benchmarks: 8 are taken from the examples in Figure 1 and 24 benchmarks are drawn from a variety real database systems. The benchmarks can be found at http://www.cs.yale.edu/homes/ejk/tools/eleven82/

Most of these systems are widely used and built by well-respected engineers/companies. For instance, LevelDB was built by Google fellows who built parts of Google’s platform, and is the back-end database for today’s the most popular browser Chrome. Most of these systems use one of two widely used recovery paradigms:

Write-ahead logging is used in filesystems and databases and provides atomicity on general operations by writing a record of the operation to a log and then, once the log is committed, performing the operation directly on the storage system. If a crash occurs before the log is committed, the system recovers to the state before the operation; otherwise, the recovery utility scans the log to find the operation record, and redoes the operation. Real storage systems, however, may return from an operation before committing the record, trading durability for speed [32]. The following is an example from the LevelDB database server (Section 9).

```c
int fd = creat(new_log); 1
for (int i = 0; i < N; ++i) 2
append(new_log, buf[i]);
fsync(new_log); 4
psync(new_log); 5
```

Shadow paging is the generalization of the example in Section 7 and provides atomicity on general operations. Conceptually, for each operation, it copies the entire system state, applies the operation on the copy, and then atomically switches the system to use the copy as the state. Regardless of when a crash occurs, the system is in either the old or the new state. For performance, instead of copying the whole state, systems employ copy-on-write to copy pages of the state on demand [20]. The VMware benchmark (# 28-30), whose recoverability we have proved, uses shadow paging.

Experiments. We ran our experiments on Fedora Linux version 3.11.10 x86_64. This Linux guest machine was within a VirtualBox that was running on a host machine with an Intel 3.5 GHz Core i5 processor, 16GB of RAM and 3TB of hard disk space. ELEVEN82 is based on CPAchecker for which we used the PredicateAbstraction-RefinementStrategy configuration. The results of our evaluation are given below. For each system (GBDM, Level DB, etc.) we consider several variants (Var) of an extracted portion of the system. The variants explore different ways in which the algorithm could be broken. For example, the recovery script might jump to the wrong location or incorrectly reconstruct a file from disk to memory. The final variant for each benchmark is the correct program and recovery script. Many of the failed variants came about because we thought we had a crash recoverable program, but then the tool revealed otherwise.

The remaining columns indicate the following. LOC is the number of lines of code of the input example program. XLOC indicates the number of lines of the encoding we generate, that includes our optimized cross product with the file system model. (All LOC counts are based on the output after indent is applied.) The SMT column indicates which solver was used. We used SMTInterpol [9] (IPOL), except for benchmark #s 7,8 (Ex 5) and #s 18,19,20,21 (LMDB) for which we used MathSAT [11] (MSAT) because SMTInterpol crashed. The Q column indicates the number of CPA automaton states and Time indicates the total time to either verify or find a counterexample. Finally, Exp is what we expect the result of our tool to generate (either a proof ✓ or a counterexample X), and Res is the output of our tool. In all but two cases, we were able to prove crash recoverability (find a counterexample) in a minute or two. The PostgreSQL variant C and GBDM variant E timed out after the 900s CPAchecker default limit. Some benchmarks included complex recovery scripts. In the LevelDB example, the recovery script examines the disk’s entry for a file new_log and, depending on what version was last written, recovers to the appropriate location in the program, which may be in the middle of a loop. Our benchmarks helped us realize that the file descriptor _COUNTER needs to be included in the snapshots in addition to the disk and memory. We also realized that when a file is created, the disk file version should initially be the same as the memory initial version. We found the counterexamples generated by ELEVEN82 to be very helpful, as they allow one to trace back and discover why the simulation broke down.

The examples we examined all had recovery methods that iterated over a constant-length array so we provided manual termination arguments such as $i > 0 \land i' < i$. We plan to automate this in future versions of ELEVEN82.
10. Conclusions

Our work is the first automated tool for proving that programs correctly recover to their original behaviors in the event of externally-induced crashes, via a novel reduction of termination and simulation to reachability. We have proved that our work is sound and have experimentally validated it on several examples drawn from industrial systems.

We believe that our work provides a specification of a rich new domain where new (and adaptations of existing) automatic verification techniques can be applied. We have also demonstrated that there is practical viability for verification in this domain, and establishes an immediate recoverability hierarchy that can be explored in future work. Going forward, we hope to investigate abstractions that yield techniques for proving $N$- or $\infty$-recoverability. Finally, while recoverability for general-purpose programs over infinite state spaces is undecidable, there may be decidability/complexity results for interesting fragments.

Acknowledgments

We would like to thank Morgan Deters, Pietro Ferrara, Tachio Terauchi, and the blind reviewers for their helpful suggestions. Yang’s contribution to this paper was supported in part by AFRL FA8650-11-C-7190 and FA8750-10-2-0253; ONR N00014-12-1-0166; NSF CCF-1162021, CNS-1054906; an NSF CAREER award; an AFOSR YIP award; and a Sloan Research Fellowship.

References


Figure 6. Evaluation of ELEVEN82 proving crash recoverability on examples from Figure 1 and a variety of industrial systems.