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 some 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

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

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 $P \vDash \varphi$, 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 *un*crashed 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 $\mathcal{A}$ described as a control-flow automaton [24], we describe a novel transformation into an encoding $\mathcal{A}_{\mathcal{E}}^{(\circ)}$ such that

$$\exists \mathcal{D}.\mathcal{A}_{\mathcal{E}}^{\mathcal{D}} \text{ cannot reach } \texttt{error} \;\Rightarrow\; \mathcal{A} \text{ is crash recoverable}$$

where $\mathcal{D}$ is an indexed set of termination arguments. The encoding $\mathcal{A}_{\mathcal{E}}^{\mathcal{D}}$ symbolically tracks snapshots of pre-crash states and, after a crash, attempts to align the crashed program with one of these previously saved *un*crashed program snapshots in order to show simulation. $\mathcal{A}_{\mathcal{E}}^{\mathcal{D}}$ 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*

[*] 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 ∞-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 disk. Yet atop this limited interface storage systems must provide rich, intuitive abstractions such as files, directories, and transactions. 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/reusability. 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, baring 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 utility `fsck`). 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 } *Initial state*
3. Execute the program
4. Crash mid-execution } *Crash*
5. Re-Boot machine
6. Execute recovery script } *Recovery*
7. Establish program environment
8. Re-execute the program (perhaps from a checkpoint)

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 havocs 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 nondeterminisitically 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 (uncrashed) 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.

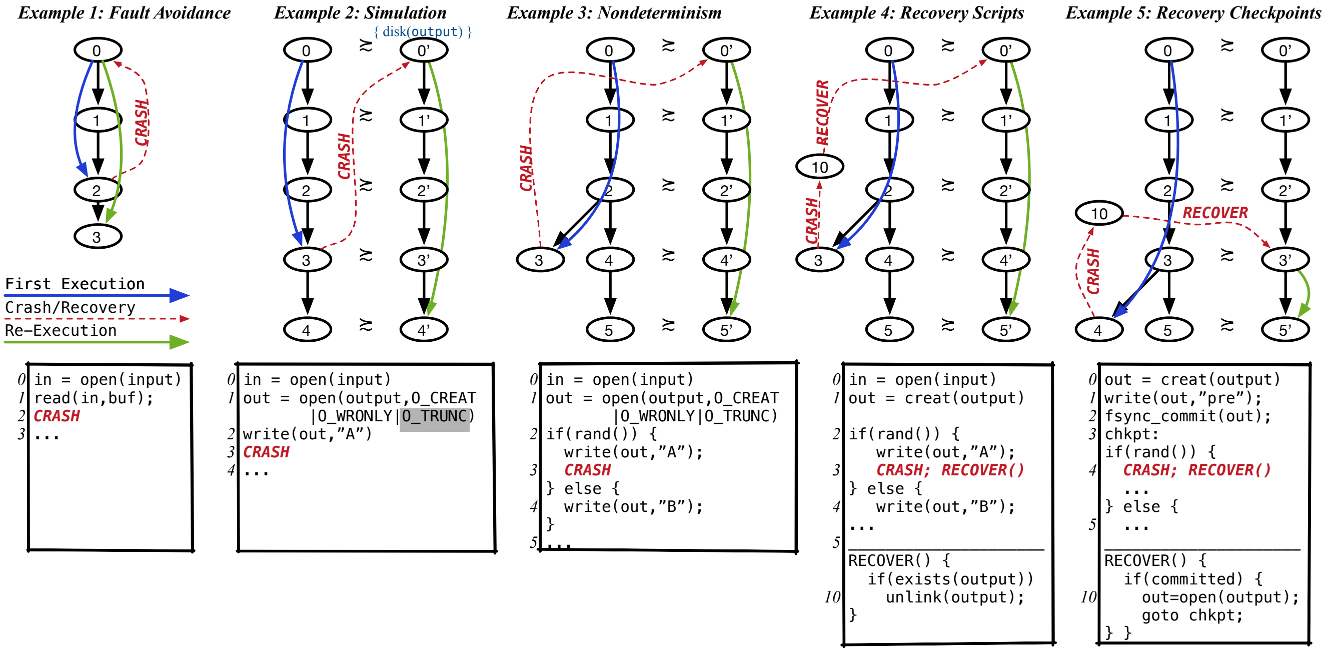
Informally, we might think of the trace semantics $[\![P]\!]$ of a program P, and use $\hat{P}$ to indicate a program that is identical to P except that it may crash (at any point) and be re-executed. We would like to show that $[\![\hat{P}]\!] \subseteq [\![P]\!]$ modulo a simulation relation. In particular, 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



Example 1: Fault Avoidance

```
0 in = open(input)
1 read(in,buf);
2 CRASH
3 ...
```

Example 2: Simulation

```
0 in = open(input)
1 out = open(output,O_CREAT
           |O_WRONLY|O_TRUNC)
2 write(out,"A")
3 CRASH
4 ...
```

Example 3: Nondeterminism

```
0 in = open(input)
1 out = open(output,O_CREAT
           |O_WRONLY|O_TRUNC)
2 if(rand()) {
    write(out,"A");
3   CRASH
  } else {
4   write(out,"B");
  }
5 ...
```

Example 4: Recovery Scripts

```
0 in = open(input)
1 out = creat(output)

2 if(rand()) {
     write(out,"A");
3    CRASH; RECOVER()
  } else {
4    write(out,"B");
  ...
5 ______________________
  RECOVER() {
    if(exists(output))
10    unlink(output);
  }
```

Example 5: Recovery Checkpoints

```
0 out = creat(output)
1 write(out,"pre");
2 fsync_commit(out);
3 chkpt:
  if(rand()) {
4   CRASH; RECOVER()
    ...
  } else {
5   ...
  ______________________
  RECOVER() {
    if(committed) {
10    out=open(output);
      goto chkpt;
  } }
```

Figure 1. Examples of programs that crash and (possibly) recover.

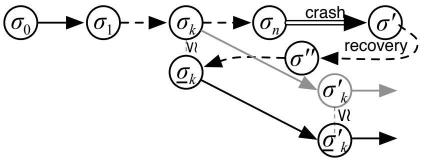
the `creat(output)` operation rather than using `open`. This is acceptable because there is also a recovery script that, after a crash, removes the `output` file if it exists on disk. Without the recovery script, this program would not be crash recoverable.

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 `fsync` which can be used to force operations to disk. These operations are like barriers: they ensure ordering constraints. `fsync(f)` 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 `fsync_commit` is pseudo-code for atomically performing the `fsync` 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:



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_k 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 $\underline{\sigma}_k$ 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* $M = (\Sigma, \Lambda, \Gamma, \sigma_0)$ is over state space Σ, labels Λ, transitions $\Gamma \subseteq \Sigma \times \Lambda \times \Sigma$, and initial state σ_0. We denote a single transition as $\sigma \xrightarrow{\lambda} \sigma'$. A *trace* π of machine M is a sequence $\sigma_0 \xrightarrow{\lambda_0} \sigma_1 \xrightarrow{\lambda_1} \sigma_2 \cdots$ such that $\forall i \geq 0.\ \sigma_i \xrightarrow{\lambda_i} \sigma_{i+1}$. We will use the notations $\sigma_0 \xrightarrow{\lambda_0, \lambda_1, \ldots} \cdots$ and $\sigma_0 \xrightarrow{\Lambda_a} \cdots$ where Λ_a 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 $\Pi(M, S)$ to mean the set of all traces of M from S, and we use $\Pi(M)$ to mean $\Pi(M, \{\sigma_0\})$. We also will use the notation $\sigma \xrightarrow{\lambda} \pi$ when we want to talk about the prefix $\sigma \xrightarrow{\lambda}$ of a trace $\sigma \xrightarrow{\lambda} \pi$. We use π^0 to denote the first state of π, and π_n to be the nth *suffix* of π (the latter binds tighter than the former in π_n^0). As usual, we say that a *simulation relation* is a relation $\precsim$ on states, defined coinductively:

$$\frac{\forall \lambda_1.\ \sigma_1 \xrightarrow{\lambda_1} \sigma_1'.\ \exists \lambda_2\, \sigma_2'.\ \sigma_2 \xrightarrow{\lambda_2} \sigma_2' \ \wedge\ \lambda_1 \overset{\Delta}{=} \lambda_2 \ \wedge\ \sigma_1' \precsim \sigma_2'}{\sigma_1 \precsim \sigma_2}$$

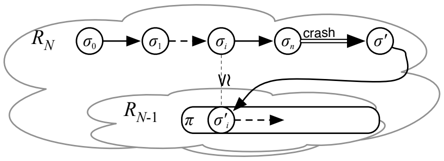
Later in this paper, we will describe our automatic technique in which transitions will correspond to abstract method *operations*. We will use arc labels λ_1 and λ_2 to denote operation executions, and equality between λ_1 and λ_2, denoted $\overset{\Lambda}{=}$, when the results of the operations are equivalent. We will work with transition systems that have a special *crash label* $\lambda_{\texttt{crash}} \in \Lambda$.

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 Λ. Recovery attempts to bring the system back to a "normal state" σ_i. 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 σ'_k that simulates σ_k. For the purposes of this paper we focus on recovery that returns the system to a state σ'_k that simulates a state σ_k in the current trace's prefix before the $\lambda_{\texttt{crash}}$ transition. Formally, *recovery* $\mathcal{R}_N$ of a trace is defined inductively as:

$$\frac{}{\mathcal{R}_0(\pi)} \qquad \frac{\sigma \overset{\lambda}{\to} \pi^0 \quad \mathcal{R}_N(\pi)}{\mathcal{R}_N(\sigma \overset{\lambda}{\to} \pi)}$$

$$\frac{\exists\, k < n,\ j \geq 0.\ \pi^0_j \precsim \sigma_k \ \wedge\ \mathcal{R}_{N-1}(\sigma_0 \xrightarrow{\lambda_0 \cdots \lambda_{k-1}} \pi_j)}{\mathcal{R}_N(\sigma_0 \xrightarrow{\lambda_0 \cdots \lambda_{n-1}} \sigma_n \xrightarrow{\lambda_{\texttt{crash}}} \pi)}$$

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



The crashed trace $\sigma_0, \sigma_1, ..., \sigma_k, \sigma_n, ...$ is recovered, provided that (*i*) the first element of π (*i.e.* σ'_k) simulates σ_k, denoted $\pi^0 \precsim \sigma_k$ and (*ii*) we can say that π 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:

$$\begin{array}{c}\sigma \overset{\lambda}{\to} \pi^0 \quad \mathcal{R}(\pi)\\ \hline\hline \mathcal{R}(\sigma \overset{\lambda}{\to} \pi)\end{array} \quad \begin{array}{c}\exists\, k < n, j \geq 0. \pi^0_j \precsim \sigma_k \wedge \mathcal{R}(\sigma_0 \xrightarrow{\lambda_0 \cdots \lambda_{k-1}} \pi_j)\\ \hline\hline \mathcal{R}(\sigma_0 \xrightarrow{\lambda_0 \cdots \lambda_{n-1}} \sigma_n \xrightarrow{\lambda_{\texttt{crash}}} \pi)\end{array}$$

Crashy programs have a "doomed" piece of computation: some states are visited (above, $\sigma_k, ..., \sigma_n, ..., \pi^0_{j-1}$) 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 ∞-*recoverable* denoted $\mathcal{R}(M)$ provided $\forall \pi \in \Pi(M)$ that $\mathcal{R}(\pi)$. M is N-*recoverable* denoted $\mathcal{R}_N(M)$ provided that $\forall \pi \in \Pi(M)$. $\mathcal{R}_N(\pi)$. 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 $\mathcal{A}$, into a new automaton $\mathcal{A}^{\mathcal{D}}_{\mathcal{E}}$ which symbolically encodes the 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 $\mathcal{A}$ is a tuple $\langle Q, q_0, X, \texttt{Op}, \twoheadrightarrow\rangle$ where Q is a finite set of control locations, q_0 is the initial control location, X is a finite set of typed variables, $\texttt{Op}$ is a set of operations and $\twoheadrightarrow \subseteq Q \times \texttt{Op} \times Q$ 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 $\theta : X \to Val$ to be a mapping from variables names to values. Let Θ be the set of all valuations. An $m \in \texttt{Op}$ is a name of a method call (operation). The notation $\mathsf{exec}(\theta_i, m, \theta_{i+1}, Y)$ means that executing method m, using the values given in θ_i, leads to a new valuation θ_{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 $m \in \texttt{Op}$ that $\mathsf{exec}(\theta_1, m, \theta_2, Y)$ is computed 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 determinizing the input CFA: whenever there is a nondeterministic operation m, we can augment X with a fresh prophecy variable ρ, and replace m with a version that consults ρ to resolve nondeterminism. Similar symbolic determinization 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 $\forall i \geq 0.\ \exists m.\ q_i \overset{m}{\twoheadrightarrow} q_{i+1} \ \wedge\ \mathsf{exec}(\theta_i, m, \theta_{i+1}, Y)$. Again, for convenience, we assume only infinite runs (this is not without loss of generality). We denote as $\Pi(\mathcal{A})$ the set of all runs of $\mathcal{A}$. We say that CFA $\mathcal{A}$ can *reach* automaton state q provided that $\exists q_0, \theta_0, q_1, \theta_1, ... \in \Pi(\mathcal{A})$ such that there is some $i \geq 0$ such that $q_i = q$.

For CFA $\mathcal{A} = \langle Q, q_0, \texttt{Op}, X, \texttt{Op}, \twoheadrightarrow\rangle$, with valuation space Θ, we define the *induced trans. system* $M^q_{\mathcal{A}} = \langle \Sigma_{\mathcal{A}}, \Lambda_{\mathcal{A}}, \Gamma_{\mathcal{A}}, \sigma^0_{\mathcal{A}}\rangle$ as:

$$\begin{array}{rl}
\Sigma_{\mathcal{A}} = & Q \times \Theta, \\
\Lambda_{\mathcal{A}} = & \texttt{Op}, \\
\Gamma_{\mathcal{A}} = & \{\langle q_1, \theta_1\rangle, m, \langle q_2, \theta_2\rangle \mid q_1 \overset{m}{\twoheadrightarrow} q_2 \ \wedge\ \mathsf{exec}(\theta_1, m, \theta_2, Y)\}, \\
\sigma^0_{\mathcal{A}} = & \langle q, \theta_0\rangle
\end{array}$$

We will simply write $M_{\mathcal{A}}$ when we mean $M^{q_0}_{\mathcal{A}}$. For a run $r = q_0, \theta_0, q_1, \theta_1, \ldots$ of $\mathcal{A}$, we define the *induced trace* to be $\pi_r \equiv$

For an input CFA $\mathcal{A} = \langle Q, q_0, \texttt{Op}, X, \texttt{Op}, \twoheadrightarrow\rangle$ with $Q_{\text{rcv}} \subseteq Q$, and crash script `crash`, let CFA $\mathcal{A}_{\mathcal{E}}^{\mathcal{D}} \equiv \langle Q_{\mathcal{E}}, q_{\mathcal{E}}^0, X_{\mathcal{E}}, \texttt{Op}_{\mathcal{E}}, \underset{\mathcal{E}}{\twoheadrightarrow}\rangle$, where

$$\begin{array}{rcl}
Q_{\mathcal{E}} & = & Q \cup \{q_{\text{err}}\} \bigcup_{(q_i,\_,q_j)\in\twoheadrightarrow} \{q_{ij}\} \\
q_{\mathcal{E}}^0 & = & q_0 \\
X_{\mathcal{E}} & = & \{\texttt{\_CR}\} \cup {`X} \cup X \cup \bigcup_i {`X_i} \bigcup \tilde{X} \\
\texttt{Op}_{\mathcal{E}} & = & \textit{(see } \underset{\mathcal{E}}{\twoheadrightarrow} \textit{ below)} \\
\underset{\mathcal{E}}{\twoheadrightarrow} & = & \{q_{\text{err}} \xrightarrow[\mathcal{E}]{\textsf{true}} q_{\text{err}}\} \cup
\end{array}$$

$$\left\{\begin{array}{lll}
\bigcup_{(q_i,m,q_j)\in\twoheadrightarrow} . & q_i \xrightarrow{\{\neg\texttt{\_CR}\}Y:=m(X);`X_i:=X} q_j & (L1) \\
 & q_i \xrightarrow{\{\texttt{\_CR}\}Y:=m(X);`Y:=m(`X);} q_{ij} & (L2) \\
 & q_{ij} \xrightarrow{\{Y=`Y\}} q_j \;\text{ and }\; q_{ij} \xrightarrow{\{Y\neq`Y\}} q_{\text{err}} & (L3) \\
\bigcup_{q_i\in Q,\hat{q}_k\in Q_{\text{rcv}}} . & q_i \xrightarrow{\texttt{crash}(X);\texttt{\_CR}:=\textsf{true};`X:=`X_k;\tilde{X}:=X} \hat{q}_k & (L4) \\
\bigcup_{\hat{q}_k,\hat{q}_{k'}\in Q_{\text{rcv}}} . & \hat{q}_k \xrightarrow{\{(\tilde{X},X)\notin\mathcal{D}\}} q_{\text{err}} \;\text{ and }\; \hat{q}_k \xrightarrow{\tilde{X}:=X;Y:=m(X)} \hat{q}_{k'} \;\text{ and }\; \hat{q}_k \xrightarrow{Y:=m(X)} \hat{q}_{k'} & (L5)
\end{array}\right.$$

Figure 2. The formal transformation from input automaton $\mathcal{A}$ to $\mathcal{A}_{\mathcal{E}}^{\mathcal{D}}$. Non-reachability of q_{err} entails crash recoverability of $\mathcal{A}$.

$\langle q_0, \theta_0\rangle, m_0, \langle q_1, \theta_1\rangle, m_1, \ldots$ where $\forall i \geq 0.\ \exists m_i.\ q_i \xrightarrow{m_i} q_{i+1} \wedge \textsf{exec}(\theta_i, m_i, \theta_{i+1}, Y)$. Finally, for a run r we define N-recovery of the run $\mathcal{R}_N(r)$ to be N-recovery of the induced trace of r. That is: $\mathcal{R}_N(\pi_r)$. Likewise for $\mathcal{R}$.

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 $X_v \uplus X_p$ where: $X_v = \{\texttt{mem}, x, y, z, ...\}$ and $X_p = \{\texttt{disk}\}$ where x, y, z are program variables.

Recovery 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 q_k, then the crash arc will lead to a special automaton state $\hat{q}_k$. (In implementations this is tantamount to marking checkpoints.) From $\hat{q}_k$, recovery will then, if it operates correctly, lead throughly recovery-oriented states $Q_{\text{rcv}} \subseteq Q$ and eventually reach state $q_k \in (Q \setminus Q_{\text{rcv}})$. 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 $\hat{q}_0$ to q_0.

5.2 The Algorithmic Reduction

Transformation of an input control-flow automaton $\mathcal{A}$ into a new CFA $\mathcal{A}_{\mathcal{E}}^{\mathcal{D}} = \mathcal{E}(\mathcal{A}, \texttt{crash}, \mathcal{D})$ is given in Figure 2. It is designed to reach q_{err} 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 $\mathcal{A}_{\mathcal{E}}^{\mathcal{D}}$ entails crash recoverability of $\mathcal{A}$. $Q_{\mathcal{E}}$ is given by Q but augmented with an error state q_{err} and intermediate states q_{ij}, $q_{\mathcal{E}}^0$ is simply q_0, and there are several duplicate sets of variables for X: variables for per-node snapshots $`X_i$, variables for a loaded snapshot $`X$ and variables for the termination check $\tilde{X}$. $\texttt{Op}_{\mathcal{E}}$ can be derived from the definition of $\underset{\mathcal{E}}{\twoheadrightarrow}$ which, intuitively, involves the following:

1. *Symbolic snapshots* $(L1)$: The transformation introduces a command that captures a symbolic snapshot $`X_j$ of $X$ on each arc entering each state $q_j$ $(L1)$. These $`X_j$ variables record the state that was visited when a run passed through q_i. Later, if recovery happens to return to q_k, the values of $`X_k$ can be loaded into $`X$ which will then be used as part of the mechanism for ensuring observational equivalence.
2. *Non-deterministic crash* $(L4)$: Crash arcs are introduced from each q_j to $\hat{q}_k$. A run of the CFA may nondeterministically choose to take these arcs which execute the `crash` and begin recovery from $\hat{q}_k$. At this point, flag `_CR` is set and snapshot $`X_k$ from recovery destination location $q_k$ is loaded into $`X$. The unindexed variables X are duplicated to $\tilde{X}$ 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 X simulates a crash-free run of the automaton from q_i whose saved state is recalled into the $`X$ variables. To this end, we create duplicate arcs, guarded by $\{\texttt{\_CR}\}$, in which the op is performed on both the crashed/recovered variables $X$ and the loaded snapshot $`X$. An arc is then added to a waypoint q_{ij}.
4. *Error states for simulation violation* $(L3)$: We introduce arcs from these waypoints q_{ij} to q_{err} whenever the return value Y of performing $m(X)$ is not equivalent to the return value $`Y$ of performing $m(`X)$ and arcs to q_j 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 $\tilde{X}$ which may, at any non-deterministic moment in time, be duplicated from X. An arc is added to q_{err} whenever $(\tilde{X}, X)$ is not in well-founded $\mathcal{D}$.

The valuations arising from runs of $\mathcal{A}_{\mathcal{E}}^{\mathcal{D}}$ involve multiple components (arising from the symbolic snapsots). We will use the notation: $[\mathbb{B}, `\theta, \tilde{\theta}, \theta, `\theta_0, `\theta_1, \ldots]$ where the first component is a boolean value for `_CR`, the second component is the valuation $`\theta$ is used for the loaded snapshot $`X$ variables, the third component is the valuation $\tilde{\theta}$ used for the $\tilde{X}$ termination variables, the fourth component is the valuation $\theta$ used for the main $X$ variables, and the remaining components are the valuations of snapshot $`X_j$ variables. Thus, a run $r_{\mathcal{E}}$ of the encoded machine will be of form

$$\begin{array}{rll}
r_{\mathcal{E}} = & q_0, [\textsf{false}, \bot, \bot, \theta_0, \bot, \bot, \ldots], & \text{(initial config.)} \\
 & q_1, [\textsf{false}, \bot, \bot, \theta_1, `\theta_0, \bot, \ldots], & \\
 & q_2, [\textsf{false}, \bot, \bot, \theta_2, `\theta_0, `\theta_1, \ldots], & \\
 & \ldots &
\end{array}$$

6. Soundness

We now give some helper lemmas along with our soundness result. We will abbreviate post-crash valuations, as $[\textsf{true}, `\theta, \theta]$, focusing only on the restored snapshot $‘\theta$ and the crashed/recovered θ, omitting the other snapshots (note that the valuation $\tilde{\theta}$ is needed for Lemma 6.3 but, for simplicity, not displayed here).

Lemma 6.1. *Every configuration of $\mathcal{A}^{\mathcal{D}}_{\mathcal{E}}$ has a successor.*

Proof. By the axiom that every configuration of $\mathcal{A}$ has a successor and by construction of $\mathcal{A}^{\mathcal{D}}_{\mathcal{E}}$. □

Lemma 6.2 (Simulation)**.** *For all $\mathcal{A}$ and corresponding encoding $\mathcal{A}^{\mathcal{D}}_{\mathcal{E}}$, if $\forall r_{\mathcal{E}}$ beginning at config. $\langle q_0, [\mathit{true}, ‘\theta_0, \theta_0]\rangle$ that $r_{\mathcal{E}}$ cannot reach q_{err}, then $\langle q_0, ‘\theta_0\rangle \precsim \langle q_0, \theta_0\rangle$ where these induced states are w.r.t. respective transition systems $‘M = \langle Q \times \theta, \mathit{Op}, \Gamma_{\mathcal{A}}, \langle q_0, ‘\theta_0\rangle\rangle$ and $M = \langle Q \times \theta, \mathit{Op}, \Gamma_{\mathcal{A}}, \langle q_0, \theta_0\rangle\rangle$ both induced by $\mathcal{A}$.*

Proof. Let notation $\langle q, [\_, \_, \theta]\rangle \not\twoheadrightarrow q_{\text{err}}$ mean that q_{err} cannot be reached in any run from $\langle q, [\_, \_, \theta]\rangle$. We prove the lemma by coinduction with the following hypothesis:

$$\frac{\begin{array}{l}\forall q_1\ \theta_1\ ‘\theta_1\ m.\ q_0 \xrightarrow{m}\!\!\!\!\rightarrow q_1\ \wedge \\ \langle q_1, [\mathsf{true}, ‘\theta_1, \theta_1]\rangle \not\twoheadrightarrow q_{\text{err}} \Rightarrow \langle q_1, \theta_1\rangle \precsim \langle q_1, ‘\theta_1\rangle\end{array}}{\forall q_0\ ‘\theta_0\ \theta_0.\ \langle q_0, [\mathsf{true}, ‘\theta_0, \theta_0]\rangle \not\twoheadrightarrow q_{\text{err}} \Rightarrow \langle q_0, \theta_0\rangle \precsim \langle q_0, ‘\theta_0\rangle}$$

First we specialize $\precsim$ for CFA-induced trans. systems:

$$\frac{\begin{array}{l}\forall m\ j\ \theta_j.\ q_i \xrightarrow{m}\!\!\!\!\rightarrow q_j \wedge \mathsf{exec}(\theta_i, m, \theta_j, Y). \\ \exists ‘\theta_j.\ \mathsf{exec}(‘\theta_i, m, ‘\theta_j, ‘Y)\ \wedge\ Y = ‘Y\ \wedge\ \langle q_j, \theta_j\rangle \precsim \langle q_j, ‘\theta_j\rangle\end{array}}{\langle q_i, \theta_i\rangle \precsim \langle q_i, ‘\theta_i\rangle}$$

To see that this definition is guarded, recall our axioms that a configuration $\langle q, \theta\rangle$ always has a successor and that exec always terminates and produces a valid θ'.

Now, letting $i = 0$, we have

$$\frac{\begin{array}{l}\forall m\ \theta_j.\ q_0 \xrightarrow{m}\!\!\!\!\rightarrow q_j \wedge \mathsf{exec}(\theta_0, m, \theta_j, Y). \\ \exists ‘\theta_j.\ \mathsf{exec}(‘\theta_0, m, ‘\theta_j, ‘Y)\ \wedge\ Y = ‘Y\ \wedge\ \langle q_j, \theta_j\rangle \precsim \langle q_j, ‘\theta_j\rangle\end{array}}{\langle q_0, \theta_0\rangle \precsim \langle q_0, ‘\theta_0\rangle}$$

Consider an arc $q_0 \xrightarrow{m}\!\!\!\!\rightarrow q_1$ in $\mathcal{A}$. For this q_0, m, q_1 we know by construction $(L2, L3, L4)$ that $\mathcal{A}^{\mathcal{D}}_{\mathcal{E}}$ has the arcs:

$$q_0 \xrightarrow{\{\texttt{\_CR}\}Y=m(X);‘Y=m(‘X);}\!\!\!\!\rightarrow q_{01}, q_{01} \xrightarrow{Y=‘Y}\!\!\!\!\rightarrow q_1, q_{01} \xrightarrow{Y\neq‘Y}\!\!\!\!\rightarrow q_{\text{err}}.$$

Since $\langle q_0, [\mathsf{true}, ‘\theta_0, \theta_0]\rangle \not\twoheadrightarrow q_{\text{err}}$, and by Lemma 6.1, configuration $\langle q_0, [\mathsf{true}, ‘\theta_0, \theta_0]\rangle$ must have a successor (and a successor's successor), there must exist some θ_1 and $‘\theta_1$ such that $\mathsf{exec}(\theta_0, m, Y, \theta_1)$ and $\mathsf{exec}(‘\theta_0, m, ‘Y, ‘\theta_1)$ and $Y = ‘Y$. Moreover, $\mathcal{A}^{\mathcal{D}}_{\mathcal{E}}$ can reach $\langle q_1, [\mathsf{true}, ‘\theta_1, \theta_1]\rangle$ and since $\langle q_0, [\mathsf{true}, ‘\theta_0, \theta_0]\rangle$ can't reach q_{err}, neither q_{err} be reached from $\langle q_1, [\mathsf{true}, ‘\theta_1, \theta_1]\rangle$.

We now let $j = 1$ and apply the inductive hypothesis to this $q_1, \theta_1, ‘\theta_1$, obtaining that $\langle q_1, ‘\theta_1\rangle \precsim \langle q_1, \theta_1\rangle$. Applying the definition of $\precsim$, we obtain that $\langle q_0, ‘\theta_0\rangle \precsim \langle q_0, \theta_0\rangle$. □

Lemma 6.3 (Eventuality)**.** *If there exists $\mathcal{D}$ such that encoding $\mathcal{A}^{\mathcal{D}}_{\mathcal{E}} = \mathcal{E}(\mathcal{A}, \mathit{crash}, \mathcal{D}) \not\twoheadrightarrow q_{err}$, then for every $\hat{q}_k \in Q$. $M^{\hat{q}_k}_{\mathcal{A}}$ eventually reaches some $\langle q, \theta\rangle$ where $q \notin Q_{rcv}$.*

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

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

Proof. We must show that for all $\mathcal{A}$ and crash script $\mathtt{crash}$, $\mathcal{A}^{\mathcal{D}}_{\mathcal{E}} = \langle Q_{\mathcal{E}}, q^0_{\mathcal{E}}, X_{\mathcal{E}}, \mathtt{Op}_{\mathcal{E}}, \twoheadrightarrow_{\mathcal{E}}\rangle$ and the transition system $M_{\mathcal{A}}$ induced by $\mathcal{A}$, that

$$\frac{\begin{array}{ll}\text{(C1)} & \mathcal{A}^{\mathcal{D}}_{\mathcal{E}} \text{ cannot reach } q_{\text{err}} \\ \text{(C2)} & \forall \hat{q}_k \in Q.\ M^{\hat{q}_k}_{\mathcal{A}} \text{ eventually reaches some } \langle q, \theta\rangle\end{array}}{\mathcal{R}_1(M_{\mathcal{A}})}$$

(C2) is given by Lemma 6.3. Unfolding $\mathcal{R}_1$, we must show that for every trace $\pi_r \in \Pi(M_{\mathcal{A}})$, that $\mathcal{R}_1(\pi_r)$, where the trace π_r is induced by run r of $\mathcal{A}$. 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 $\mathcal{A}$ will be of the form:

$$\begin{array}{c}\langle q_0, \theta_0\rangle, m_0, \ldots, \langle q_n, \theta_n\rangle, \lambda_{\mathtt{crash}}, \\ \langle q_{n+1}, \theta_{n+1}\rangle, m_{n+1}, \ldots, \langle q_j, \theta_j\rangle, \ldots\end{array}$$

To derive $\mathcal{R}_1$, we must show that:
1. There exists $K \leq n$ such that ...
2. ... there exists a $J > n$ such that $\langle q_J, \theta_J\rangle \precsim \langle q_K, \theta_K\rangle$.

(The requirement that $\mathcal{R}_0$ hold of the "snipped" trace is trivially satisfied.) By construction of $\mathcal{A}^{\mathcal{D}}_{\mathcal{E}}$, there is a corresponding run of $\mathcal{A}^{\mathcal{D}}_{\mathcal{E}}$ with the following form, using the left column to indicate the index number of the run:

$$\begin{array}{l|ll}
0. & \langle q_0, [\mathsf{false}, \bot, \theta_0, \{\}]\rangle, m_0, & \text{(By L1)} \\
1. & \langle q_1, [\mathsf{false}, \bot, \theta_1, \{‘\theta_0\}]\rangle, m_1, & \\
 & \ldots, & \\
n. & \langle q_n, [\mathsf{false}, \bot, \theta_i, \{‘\theta_0, \ldots, ‘\theta_{n-1}\}]\rangle, & \\
 & \quad \mathtt{crash}(X); & \\
 & \quad \mathtt{\_CR\!:=}\mathsf{true};\ ‘X := ‘X_k, & \text{(By L5)} \\
n+1. & \langle \hat{q}_k, [\mathsf{true}, ‘\theta_k, \theta_{n+1}, \{\}]\rangle, & \\
 & \ldots, & (Q_{\text{rcv}}) \\
n+1+m. & \langle q_k, [\mathsf{true}, ‘\theta_k, \theta_{n+1+m}, \{\}]\rangle, &
\end{array}$$

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 $r_{\mathcal{E}}$.

Now we must find J. In r, from $\langle q_n, \theta_n\rangle$ we go to $\langle \hat{q}_k, \theta_n\rangle$. Condition C2 ensures that from $\langle \hat{q}_k, \theta_n\rangle$, run r reaches some $\langle q_k, \theta_{n+1+m}\rangle$ after some m (finitely many) steps. By construction $r_{\mathcal{E}}$ also reaches some configuration $\langle q_k, [\mathsf{true}, k, ‘\theta_k, \theta_{n+1+m}, \{\}]\rangle$ after m steps.

We let $J = n + 1 + m$. What remains to show is that $\langle q_J, \theta_J\rangle \precsim \langle q_k, \theta_k\rangle$. By condition C1, $r_{\mathcal{E}}$ cannot reach q_{err}, and so from $\langle q_k, [\mathsf{true}, k, ‘\theta_k, \theta_{n+1+m}, \{\}]\rangle$ we cannot reach q_{err}. By Lemma 6.2, $\langle q_k, \theta_{n+1+m}\rangle \precsim \langle q_k, ‘\theta_k\rangle$. Note that $q_J = q_k$. Rewriting we obtain $\langle q_J, \theta_J\rangle \precsim \langle q_K, \theta_K\rangle$. □

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:

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

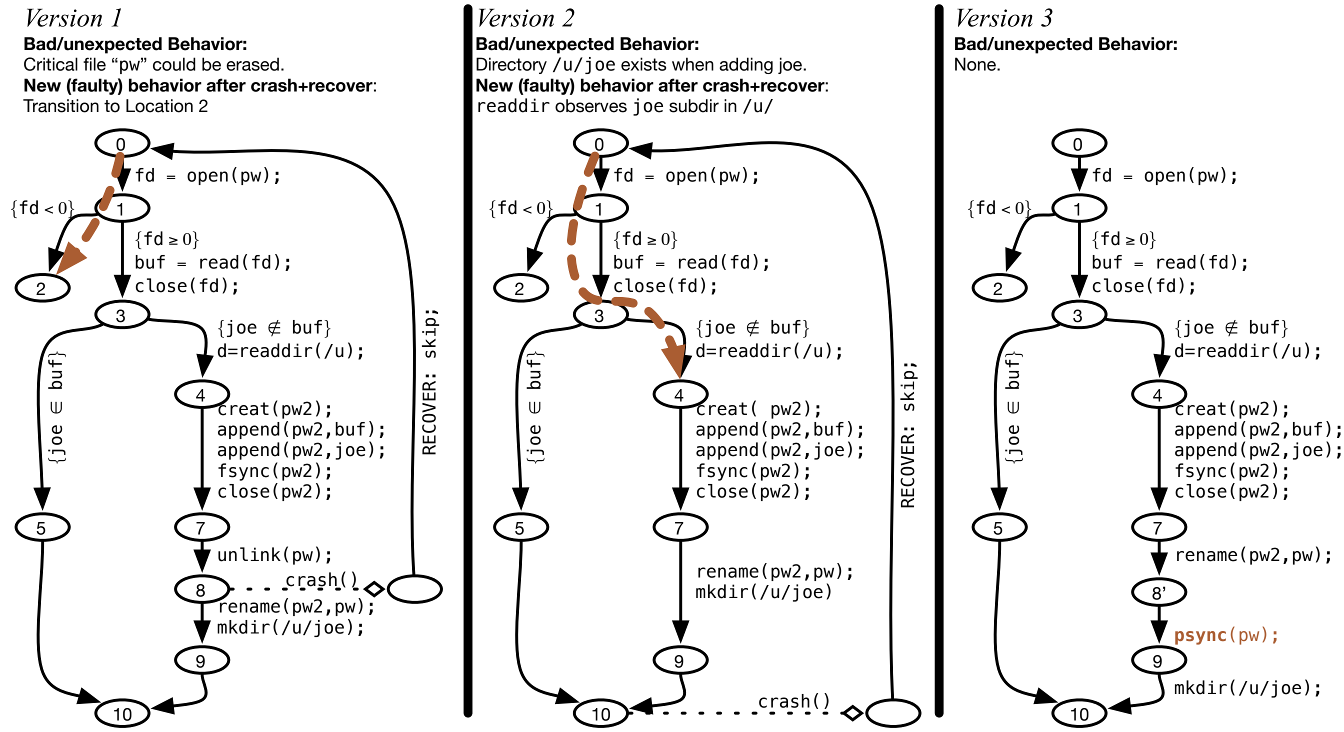


Figure 3. A control-flow automaton for three versions of the example. In all versions, file pw should initially exist. Crash arcs are everywhere, but only displayed when they are used to illustrate faulty behavior.

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 new user joe is 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 code is given in Figure 3, *Version 1*. We will refer to nodes as ℓ_0, ℓ_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 ℓ_7 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 $\mathcal{A}_{\mathcal{E}}^{\mathcal{D}}$, which is displayed in Figure 4. New code is highlighted in rounded-corner boxes and new nodes/arcs 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 ℓ_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 $\texttt{disk}_0$ and $\texttt{mem}_0$.
2. We transform the recovery, denoted $\lfloor \texttt{RECOVER}_i\texttt{()} \rfloor_{\mathcal{D}}$ using an existing technique [15] that *ensures* $\texttt{RECOVER}_i\texttt{()}$ *will terminate* by checking its inclusion in a given well-founded relation $\mathcal{D}$ 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 ℓ_1 to ℓ_0. crash() havocs the unindexed variables fd,pw,buf, and mem, sets a flag _CR to true, and runs the recovery $\texttt{RECOVER}_i\texttt{()}$ that is appropriate for recovering to ℓ_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 ‘$\texttt{pw=pw}_0$;‘$\texttt{disk=disk}_0$;‘$\texttt{mem=mem}_0$;.
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 _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≠fd).

There is a run r of this encoding $\mathcal{A}_{\mathcal{E}}^{\mathcal{D}}$ that can reach q_{err}. The run r passes through ℓ_8, follows the _CR arc, and then leads to a state at ℓ_0 in which pw is not 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_{err} since fd ≠ ‘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

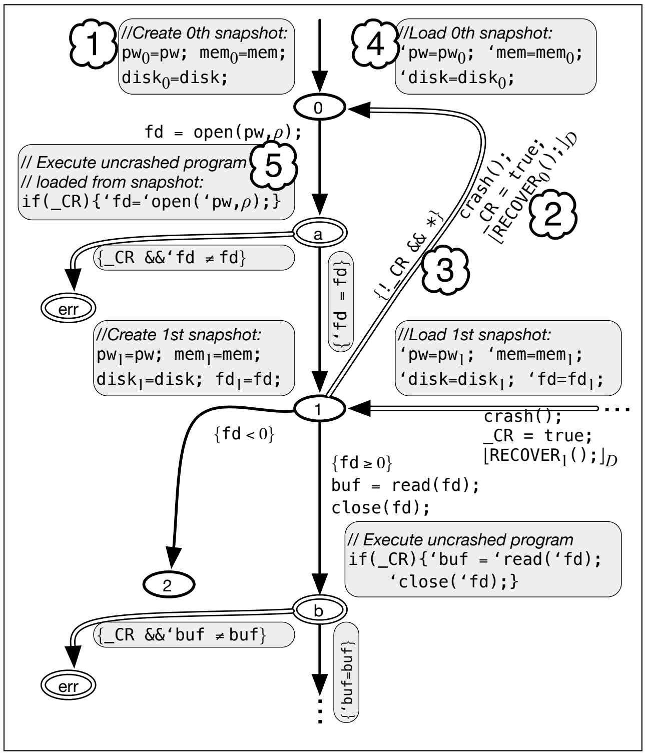


Figure 4. An illustration of the output $\mathcal{A}_{\mathcal{E}}^{\mathcal{D}}$, which is the result of applying our encoding to the example in Figure 3.

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 ℓ_{10} may leave the `pw` file unchanged but then, surprisingly, `joe` may be in the directory entry `d` returned by `readdir`. This subtlety 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 `sync` operation that forces the *parent* directory (and any modified directories along the path of the file) to be written to disk at ℓ_8, thereby ensuring that the directory entry for `pw` has been written to disk. For clarity, we use `psync` to represent the sequence of operations that open a file's parent directory, call `fsync` on the directory, and close the directory. 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 `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 theorical 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 `mem` and `disk` arrays. That is, if integer variable `fn` is non-negative, then it corresponds to a file. The in-memory *contents* of the file `fn` are given by the value `mem[fn]`, and likewise for disk. One can think of these values as version numbers. For example, `creat(fn)` 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 `fn` has been `sync`'d to disk, via the boolean flag `fn_psync`. 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. `/foo/bar/file`). Then use a series of arrays to track the parents of a given directory, such as:

$$\begin{array}{lcl} \texttt{PARENT}_1[\texttt{/foo/bar/file}] & = & \texttt{/foo/bar} \\ \texttt{PARENT}_2[\texttt{/foo/bar/file}] & = & \texttt{/foo} \\ \texttt{PARENT}_3[\texttt{/foo/bar/file}] & = & \texttt{ROOT} \\ \texttt{PARENT}_1[\texttt{/foo/bar}] & = & \texttt{/foo} \\ \ldots & & \end{array}$$

The model implementation of `creat`, `open`, etc. will then manipulate these arrays as appropriate.

Asynchronous disk. The macros for each filesystem operation (*e.g.* `read`) in Figure 5 additionally perform the cross-product with the asynchronous disk, execute the read operation on the unindexed variables and (if `_CR`) on the snapshot variables and check equivalence. The macros insert code that runs `MAY_ASYNC`. This method iterates through the file system, nondeterministically shuffling files from `mem` to `disk`, by setting `disk[i]` to be a version number that is greater than its current value, but at most the version in `mem[i]`. Additionally, the method iterates through each file, nondeterministically deciding to swap the `psync` flag from `false` to `true`. Finally, to simulate a crash, each file system macro involves the script `MAY_CRASH()`, which havocs the local variables, values in `mem`, and eliminates any `fn` for which the psync flag has not been set (`fn:=-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_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 preprocessor, 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

```
int disk[MAX_FILES];
int mem[MAX_FILES];
int _COUNTER = 0;
int creat_OTRUNC(fn) {
  if (fn<0)
    fn = _COUNTER++;
  mem[fn] = INIT_VER;
  disk[fn] = INIT_VER;
  fn_psync = FALSE;
}
```

```
void append(fn) { mem[fn] = mem[fn] + 1; }
void write(fn)  { mem[fn] = mem[fn] + 1; }
void fsync(fn)  { disk[fn] = mem[fn]; }
void psync(fn)  { fn_psync = TRUE; }
void unlink(fn) { fn = -1; }
bool exists(f) { return (f >= 0); }
void rename(f1,f2) { f1 = f2; f2 = -1; }
int read(fn) {
  if (fn < 0) assert(FALSE);
  x = mem[fn];
  return x;
}
```

```
void MAY_ASYNC() {
  for (i = 0; i < counter; i++) {
    if(* && disk[i] < mem[i]) {
      int t = disk[i];
      disk[i] = nondet();
assume(disk[i]<=mem[i]&&disk[i]>=t);
    }
  }
  foreach (file : ALL_FILES)
   if(nondet() && ¬ file_psync)
     file_psync = true;
}
```

Figure 5. A model of an asynchronous filesystem, designed to enable tools such as CPAchecker, which don't support trees, to reason about filesystems using arrays and integers.

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`/`mem` arrays, 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 examples 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).

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

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** (GDBM, 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 χ), and **Res** is the output of our tool. In all but two cases, we were able to prove crash recoverability (or find a counterexample) in a minute or two. The PostgreSQL variant C and GDBM 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 `creat`ed, 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 $[\![\texttt{i} > 0 \wedge \texttt{i}' < \texttt{i}]\!]$. We plan to automate this in future versions of ELEVEN82.

Benchmarks				ELEVEN82					
#	**System**	**Var.**	**LOC**	**XLOC**	Q	**SMT**	**Time**	**Exp**	**Res**
1	Fig 1, Ex 2	A	53	256	167	IPOL	7.750s	χ	χ
2	Fig 1, Ex 2	A	53	248	166	IPOL	15.988s	✓	✓
3	Fig 1, Ex 3	A	61	449	317	IPOL	7.583s	χ	χ
4	Fig 1, Ex 3	A	61	443	318	IPOL	18.129s	✓	✓
5	Fig 1, Ex 4	A	66	452	319	IPOL	20.496s	χ	χ
6	Fig 1, Ex 4	A	66	482	331	IPOL	24.943s	✓	✓
7	Fig 1, Ex 5	A	88	759	515	MSAT	34.333s	χ	χ
8	Fig 1, Ex 5	A	96	767	518	MSAT	50.033s	✓	✓
9	GDBM create	A	84	846	557	IPOL	18.105s	χ	χ
10	GDBM create	B	85	332	218	IPOL	21.320s	✓	✓
11	GDBM create	C	84	842	554	IPOL	19.060s	χ	χ
12	GDBM create	D	83	971	642	IPOL	32.916s	χ	χ
13	GDBM create	E	87	937	612	IPOL	>900s	✓	**T/O**
14	Level DB	A	58	369	232	IPOL	46.321s	✓	✓
15	Level DB	B	52	283	197	IPOL	10.528s	χ	χ
16	Level DB	C	58	403	269	IPOL	25.229s	χ	χ
17	Level DB	D	60	460	297	IPOL	77.090s	✓	✓
18	LMDB	A	50	410	302	MSAT	16.304s	χ	χ
19	LMDB	B	51	473	349	MSAT	77.843s	✓	✓
20	LMDB	C	57	606	427	MSAT	24.718s	χ	χ
21	LMDB	D	51	473	349	MSAT	77.430s	✓	✓
22	PostgreSQL	A	59	655	564	IPOL	15.272s	χ	χ
23	PostgreSQL	B	60	716	624	IPOL	16.218s	χ	χ
24	PostgreSQL	C	68	895	706	IPOL	>900s	✓	**T/O**
25	PostgreSQL	D	68	658	505	IPOL	315.109s	✓	✓
26	SQLite	A	69	956	770	IPOL	20.900s	χ	χ
27	SQLite	B	65	875	700	IPOL	71.006s	✓	✓
28	VMware	A	65	540	450	IPOL	10.235s	χ	χ
29	VMware	B	68	715	558	IPOL	7.605s	χ	χ
30	VMware	C	68	715	558	IPOL	63.055s	✓	✓
31	ZooKeeper	A	56	594	453	IPOL	26.422s	χ	χ
32	ZooKeeper	B	55	524	397	IPOL	43.490s	✓	✓

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

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 recoverabiltiy hierarchy that can be explored in future work. Going forward, we hope to investigate abstractions that yield techniques for proving N- or ∞-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

[1] R. Alur, T. A. Henzinger, F. Y. C. Mang, S. Qadeer, S. K. Rajamani, and S. Tasiran. MOCHA: modularity in model checking. In *Computer Aided Verification, 10th International Conference, CAV '98, Proceedings*, pages 521–525, 1998.

[2] L. N. Bairavasundaram. PhD thesis, Characteristics, Impact, and Tolerance of Partial Disk Failures, 2008.

[3] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011.

[4] T. Beyene, C. Popeea, and A. Rybalchenko. Solving existentially quantified horn clauses. In *CAV'11*, 2013.

[5] D. Beyer and M. E. Keremoglu. Cpachecker: A tool for configurable software verification. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806, pages 184–190. Springer, 2011.

[6] M. Castro, B. Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[7] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015*, pages 18–37. ACM, 2015.

[8] H. Chen, D. Ziegler, A. Chlipala, M. F. Kaashoek, E. Kohler, and N. Zeldovich. Specifying crash safety for storage systems. In *15th Workshop on Hot Topics in Operating Systems*, 2015.

[9] J. Christ, J. Hoenicke, and A. Nutz. Smtinterpol: An interpolating SMT solver. In A. F. Donaldson and D. Parker, editors, *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*, volume 7385 of *Lecture Notes in Computer Science*, pages 248–254. Springer, 2012.

[10] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model checker. *STTT*, 2(4):410–425, 2000.

[11] A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT Solver. In N. Piterman and S. Smolka, editors, *Proceedings of TACAS*, volume 7795 of *LNCS*. Springer, 2013.

[12] B. Cook and E. Koskinen. Making prophecies with decision predicates. In T. Ball and M. Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 399–410. ACM, 2011.

[13] B. Cook and E. Koskinen. Reasoning about nondeterminism in programs. In *PLDI'13*. ACM, 2013.

[14] B. Cook, E. Koskinen, and M. Y. Vardi. Temporal property verification as a program analysis task. In *CAV'11*, pages 333–348, 2011.

[15] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI'06*, pages 415–426, 2006.

[16] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. In *Eighteenth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS '13)*, 2013.

[17] D. Dietsch, M. Heizmann, V. Langenfeld, and A. Podelski. Fairness modulo theory: A new approach to LTL software model checking. In *CAV*, 2015.

[18] G. R. Ganger and Y. N. Patt. Soft updates: A solution to the metadata update problem in file systems. Technical report, University of Michigan, 1995.

[19] P. Gardner, G. Ntzik, and A. Wright. Local reasoning for the POSIX file system. In Z. Shao, editor, *Proceedings of the 23rd European Symposium on Programming, ESOP 2014*, volume 8410 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2014.

[20] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, and I. Traiger. The recovery manager of the system r database manager. *ACM Comput. Surv.*, 13(2):223–242, June 1981.

[21] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo. Deep specifications and certified abstraction layers. In *PRoceedings of the42nd ACM Symposium on Principles of Programming Languages (POPL'15)*, 2015.

[22] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In S. Chatterjee and M. L. Scott, editors, *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2008, Salt Lake City, UT, USA, February 20-23, 2008*, pages 175–184. ACM, 2008.

[23] H. S. Gunawi, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Sqck: A declarative file system checker. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 131–146, 2008.

[24] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the 14th International Conference on Computer Aided Verification (CAV'02)*, volume 2404, pages 526–538. Springer, 2002.

[25] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, Mar. 1992.

[26] G. Ntzik, P. da Rocha Pinto, and P. Gardner. Fault-tolerant resource reasoning. *Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS), Pohang, South Korea*, 2015.

[27] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[28] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. In D. A. Reed and V. Sarkar, editors, *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2009, Raleigh, NC, USA, February 14-18, 2009*, pages 163–172. ACM, 2009.

[29] Y. Raz. The principle of commitment ordering, or guaranteeing serializability in a heterogeneous environment of multiple autonomous resource mangers using atomic commitment. In *Proceedings of the 18th International Conference on Very Large Data Bases*, pages 292–312. Morgan Kaufmann Publishers Inc., 1992.

[30] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell. Sibylfs: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th Symp. on Operating Systems Principles, SOSP 2015*, pg. 38–53, 2015.

[31] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, pages 352–367, 2005.

[32] S. C. Tweedie. Journaling the Linux ext2fs File System. In *the Fourth Annual Linux Expo*, May 1998.

[33] J. Yang, C. Sar, and D. Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 131–146, Nov. 2006.

[34] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 273–288, Dec. 2004.

[35] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Transactions on Computer Systems*, 24(4):393–423, Nov. 2006.

[36] J. Yang, P. Twohey, B. Pfaff, C. Sar, and D. Engler. eXplode: A lightweight, general approach for finding serious errors in storage systems. In *Proceedings of the first Workshop on the Evaluation of Software Defect Detection Tools (BUGS '05)*, June 2005.