Proving Correctness of a Controller Algorithm for the RAID Level 5 System*

Mandana Vaziri and Nancy Lynch

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

December 7, 1997

Abstract

Most RAID controllers implemented in industry are complicated and difficult to reason about. This complexity has led to software and hardware systems that are difficult to debug and hard to modify. To overcome this problem Courtright and Gibson have developed a rapid prototyping framework for RAID architectures which relies on a generic controller algorithm [Courtright94]. The designer of a new architecture needs to specify parts of the generic controller algorithm and must justify the validity of the controller algorithm obtained. However the latter task may be difficult due to the concurrency of operations on the disks. This is the reason why it would be useful to provide designers with an automated verification tool tailored specifically for the RAID prototyping system.

As a first step towards building such a tool, our approach consists of studying several controller algorithms manually, to determine the key properties that need to be verified.

This paper presents the modeling and verification of a controller algorithm for the RAID Level 5 System [Gibson95]. We model the system using I/O automata [Lynch89], give an external requirements specification, and prove that the model implements its specification. We use a key invariant to find an error in a controller algorithm for the RAID Level 6 System [Gibson95].

Keywords: Modeling, Formal verification, I/O Automata, RAID

1 Introduction

Most RAID controllers implemented in industry are complicated and difficult to reason about. This complexity has led to software and hardware systems that are difficult to debug and hard to extend. To overcome this problem Courtright and Gibson have developed a rapid prototyping framework for RAID architectures which relies on a generic controller algorithm [Courtright94]. The designer of a new architecture needs to specify parts of the generic controller algorithm and must justify the validity of the controller algorithm obtained. However the latter task may be difficult due to the concurrency of operations on the disks. This is the reason why it would be useful to provide designers with an automated verification tool tailored specifically for the RAID prototyping system.

As a first step towards building such a tool, our approach consists of studying several controller algorithms manually, to determine the key properties that need to be verified.

*This work was supported by ARPA contract F19628-95-C-0118 and AFOSR contract F49620-97-1-0337

This paper presents the correctness of a controller algorithm [Gibson95] for the RAID Level 5 system. We chose this architecture because of its popularity, as well as for its relative simplicity.

Our method consists of modeling the controller formally, giving an external requirements specification and proving that the model satisfies its specification.

Our study results in a key invariant for the controller that can be generalized for other architectures. We use this invariant to find an error in a RAID Level 6 controller [Gibson95].

The outline of the paper is as follows. Sections 2 and 3 give background on RAID systems and I/O automata respectively. Section 4 presents the RAID Level 5 system informally. Section 5 gives conventions used throughout the paper. Section 6 describes the specification and Section 7 our model of RAID Level 5 and its properties. Section 8 presents the proof of correctness and Section 9 the extension of our work to the study of RAID Level 6, as well as the error found. Finally, Section 10 is a summary of our conclusions and future work.

2 RAID Systems

Improvements in semiconductor technology make possible faster microprocessors and larger primary memory systems, making secondary storage systems the bottleneck of overall system performance. As microprocessors get faster, the overall system improvement will not be significant unless there are also improvements in secondary storage systems.

The emergence of new applications such as video, hypertext and multimedia has also increased the need for larger secondary storage systems with higher performance. RAID or Redundant Arrays of Inexpensive Disks were developed in the 1980's to address this need. They were first described at the beginning of the decade [Lawlor81, Park86], and popularized by the work of a group at UC Berkeley [Patterson88, Patterson89].

We can think of a RAID system as being composed of two parts:

- A disk array, and
- A disk array controller.

The controller's function is to receive a *high-level operation* from the user of the disk array, and to carry it out by performing a set of *low-level operations* on specific disks. The user has no knowledge about the existence of a disk array and sees it as one large, logical disk with high performance. We use the terms *operation* and *low-level op* to refer to a high-level and a low-level operation, respectively.

RAID systems have two main advantages over traditional secondary storage systems. First, the data on the disks can be accessed in parallel, which improves the I/O performance. Each file that is stored in the array is striped and placed on several disks. This scheme improves the response time when the user accesses that file [Kim86, Reddy89, Salem86]. The controller can also carry out several operations at the same time if the disks involved in these operations are different. This scheme improves the throughput.

Second, when the number of disks increases in a disk array, the availability of data and the reliability of the disk array may decrease dramatically [Gibson93]. We consider independent catastrophic disk failures, i.e., disk failures in which all data stored on the disk becomes inaccessible and the disk cannot be written any further. RAID systems are designed to be fault-tolerant by storing redundant data [Gibson90] on extra disks and to tolerate 1 or 2 disk failures. The redundancy can be an identical copy of each disk, also known as disk mirroring [Bitton88, Gray90]. In this case, if the disk containing one copy fails, the controller can use the other copy which is on a separate disk. Having two copies of each data unit also has the advantage that, in the case of read operations, if the disk containing one copy is busy with a different operation, the other disk can be used instead, improving throughput. In this form of redundancy, lost or damaged data can be recovered by using the backup copy.

Another form of redundancy is having a *parity disk* [Patterson88], for n disks containing data. The parity disk contains blocks, called parity blocks, that cover groups of n blocks independently stored. A set of n blocks along with the parity block that covers them is called a *parity group*. The parity block is computed by performing a bit-wise exclusive or operation on the blocks it covers. Given any set of n blocks, the $(n+1)$st block can be recovered by performing an exclusive or operation on the n blocks.

There are several RAID architectures that are classified into five "levels" [Patterson88]. RAID Level 1 employs disk mirroring and thus uses twice as many disks as a non-redundant disk array for the same amount of data. RAID Level 2 provides redundancy by using Hamming codes. Levels 3, 4 and 5 all use parity. RAID Level 3 is bit-interleaved, meaning that data is interleaved bit-wise over the data disks. RAID Level 4 is block-interleaved. RAID Level 5 is also block-interleaved, but distributes parity among all the disks in the array. All the architectures mentioned above tolerate a single disk failure. Two other levels have been introduced. The first one is RAID Level 6 which is a two fault-tolerant architecture. It employs two parities, one of which is computed using Reed-Solomon codes. The second one is just a non-redundant disk array, RAID Level 0, which is not fault-tolerant. Beyond the above taxonomy, a number of RAID architectures have been proposed to solve a variety of price/performance/dependability tradeoff questions.

Different RAID architectures can be distinguished based on their type of encoding, mapping and algorithms used to access data. The encoding indicates the type of redundancy information used, and the mapping the placement of data and redundant information on the disk array.

Algorithms used to access data can be classified as normal-mode and failed-mode ones. In normal-mode, the controller knows about failed disks, if any, and operates on the disk array with this knowledge. In failed-mode, a disk failure has occurred in the middle of controller operation. The controller then needs to recover from the error and complete the operation. This process is called *error recovery*.

Most RAID architectures use *forward error recovery*. This scheme consists of transitioning from a state in which an error has occurred in the middle of controller operation directly to completion. This method requires knowing about the context in which an error occurred and thus involves enumerating a large number of erroneous states. Courtright notes in his dissertation that 60 to 70% of the code found in implementations based upon forward error recovery may be devoted to error recovery. Furthermore these schemes results in architecture-specific controller algorithms, making extension to new architectures difficult.

To overcome this problem, Courtright and Gibson propose a form of backward error-recovery method [Courtright94]. Traditional backward error-recovery methods consist of undoing operations and returning the disk array to an error-free state. The disadvantage of these methods is that they are expensive. However, Courtright and Gibson's method is based on retry[1]. When an error is encountered, the state of the system is modified to note which disk has failed, and the operation is retried based on the new state.

In this approach, operations are represented as Directed Acyclic Graphs (DAGs). Each node in a DAG is a low-level op to be performed on a disk or a low-level op that computes data. DAGs provide a visualization of operations, which simplifies reasoning about the ordering of low-level ops.

Courtright and Gibson's method of error recovery [Courtright94] has two requirements. First,

[1]Courtright has since moved on to a similar but different error recovery method called *roll-away error recovery* [Courtright97]

each low-level op must be idempotent. When a DAG fails, some low-level ops may have been executed and some may have not. The controller then retries the operation with a similar DAG and low-level ops that have already been performed will be performed again. Idempotency ensures that a low-level op that is executed several times has the same effect as if it is executed only once. Secondly, the execution of DAGs must leave the array in a *consistent* state, that is the redundant information must be correct.

Due to the concurrency of low-level ops and the existence of failures, reasoning about the correctness of an algorithm using Courtright and Gibson's error recovery method may be difficult. This task would be easier if an automated verification tool were provided.

As a first step towards building such a tool, our approach consists of studying several controller algorithms manually. This paper studies the correctness of a controller algorithm [Courtright94] for the RAID Level 5 system [Gibson95], that uses Courtright and Gibson's error recovery method. We chose this architecture because of its popularity, as well as for its relative simplicity.

We model the algorithm using I/O automata [Lynch96], and give an external requirements specification. We then prove that the model implements its specification, in the sense that there exists a simulation relation [Lynch96] from the model to its specification.

3 I/O Automaton Model

I/O automata [Lynch89] provide a mathematical model suitable for describing asynchronous concurrent systems. The model provides a precise way of describing and reasoning about system components that interact with each other and permits the composition of these components.

An I/O automaton is a state machine that has labels, called *actions*, associated with its transitions. The following definitions introduce *signatures* and *I/O automata*.

Definition 3.1 *A signature S is a triple consisting of three disjoint sets of actions:*

- *$in(S)$, the input actions,*
- *$out(S)$, the output actions,*
- *$int(S)$, the internal actions.*

The input together with the output actions are called the *external actions*. The external interface is defined to be the signature ($in(S)$, $out(S)$, $\emptyset$). We define $acts(S)$ to be all actions of S and $local(S)$ to be $out(S) \cup int(S)$.

Definition 3.2 *An I/O automaton A consists of five components:*

- *$sig(A)$, a signature.*
- *$states(A)$, a (not necessarily finite) set of states.*
- *$start(A)$, a nonempty subset of $states(A)$ known as the start states or initial states.*
- *$trans(A)$, a state-transition relation, where $trans(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$. An element (s,π,s') of $trans(A)$ is called a transition, or a step of A. $trans(A)$ must have the property that for every state s and every input action π, there is a transition $(s,\pi,s') \in trans(A)$.*

- *task(A), a task partition, which is an equivalence relation on local(sig(A)) having at most countably many equivalence classes.*

We do not use the task partition in our I/O automaton models, because we do not deal with fairness or liveness issues. An action π is said to be *enabled* in a state s if there is another state s' such that (s,π,s') is a transition of the automaton. Input actions are enabled in every state; i.e automata are not able to "block" input actions from occurring. An *execution fragment* of an I/O automaton is either a finite sequence $s_0,\pi_1,s_1,\pi_2,s_2,\ldots,\pi_n,s_n$, or an infinite sequence $s_0,\pi_1,s_1,\pi_2,s_2,\ldots$, of alternating states and actions such that (s_i,π_{i+1},s_{i+1}) is a transition of the automaton for every $i \geq 0$. An *execution* is an execution fragment that begins with a start state. A state is *reachable* if it occurs in some execution. The *trace* of an execution fragment is the sequence of external actions in that execution fragment. We denote the set of traces of the executions of an automaton A, by *traces*(A).

Two automata A and B are allowed to be composed together, if the following is true.

- The internal actions of A are disjoint from the actions of B, and vice versa.
- The set of output actions of A and B are disjoint.

When the above two properties are true, we say that A and B are *compatible*. The composition of A and B, denoted $A \times B$, is the following automaton.

- $sig(A \times B) = (in(A) \cup in(B) - out(A) \cup out(B),\ out(A) \cup out(B),\ int(A) \cup int(B))$
- $states(A \times B) = states(A) \times states(B)$
- $start(A \times B) = start(A) \times start(B)$
- $trans(A \times B)$ is the set of triples (s,π,s') such that, if $\pi \in acts(A)$, then $(s_A,\pi,s'_A) \in trans(A)$; otherwise $s_A = s'_A$, and similarly for B.
- $task(A \times B) = task(A) \cup task(B)$.

We say that an automaton A *implements* an automaton B if $traces(A) \subseteq traces(B)$. The following defines a *simulation relation* from an automaton A to an automaton B, and the following theorem states that if there exists a simulation relation from A to B then A implements B.

Definition 3.3 *Let A and B be two I/O automata with the same external interface and f a binary relation over states(A) and states(B). Then f is a simulation relation from A to B, provided that both of the following are true:*

1. *If $s \in$ start(A), then there exists u such that $u \in$ start(B) and $f(s,u)$ = true.*
2. *If s is a reachable state of A, u is a reachable state of B such that $f(s,u)$ = true, and (s,π,s') $\in$ trans(A), then there exists an execution fragment α of B starting with u and ending with some u' such that $f(s',u')$ = true and trace(α) = trace(π).*

Theorem 3.4 *If there is a simulation relation from an automaton A to an automaton B, then traces(A) $\subseteq$ traces(B).*

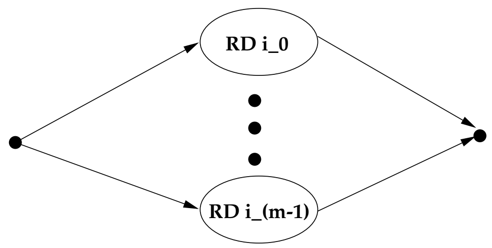

Figure 1: DAG1, Fault Free Read DAG.

4 Informal Description of the RAID Level 5 System

RAID Level 5 [Gibson95] uses parity and can tolerate one disk failure. In this architecture, data is block-interleaved and parity blocks are distributed among all the disks in the array. A parity block is the bit-wise XOR of all the blocks it covers. We assume that there are $n+1$ disks in the array.

The controller receives read and write operations from the environment sequentially. For each operation it figures out where the data to be read/written is located in the array, and what parity groups are concerned. For each parity group, the controller chooses an algorithm for carrying out the operation and starts executing it. If a disk needed in an algorithm fails while the algorithm is running, then the controller stops the execution of that algorithm and chooses another one to complete the operation. The controller assumes at most one failure.

Disk array algorithms are represented as Directed Acyclic Graphs (DAGs). Each node in a DAG represents a disk read or write or an XOR. All the low-level ops of a single DAG refer to a unique parity group and represent reads and writes to disk sectors. In a read DAG, we denote the indices of disks to be read by i_0 to i_{m-1} in an arbitrary order, where m is an integer such that $0 \leq m < n$. Similarly, in a write DAG, we denote the indices of disks to be written by i_0 to i_{m-1}. In this case, i_m to i_{n-1} represent indices of disks not to be written. We denote the index of the failed disk by *failed*. In the DAGs, the notation *RD* i means read disk with index i (similarly for *WR* i). DAGs and the criteria for choosing them are described below.

DAGs and DAG selection Figures 1 through 6 present the DAGs. Note that these are informal and do not contain information about how to compute parity blocks. We do not give formal semantics for these DAGs. An arrow from node A to node B indicates that node A must be performed before node B. Low-level ops are atomic.

- **Fault Free Read** (DAG1, Figure 1) The Fault-Free Read DAG is used when there is no failure among the disks to be read. It consists of reading disks containing the data to be read directly.

- **Degraded Read** (DAG2, Figure 2) The Degraded Read DAG is used when one of the disks to be read has failed. It consists of reading the entire array, except the failed disk, and reconstructing the missing data, by taking the bit-wise XOR of the data read.

- **Small Write** (DAG3, Figure 3) The Small Write DAG is used in the absence of failures, when less than half of the array is to be written. In the presence of a failure in a disk that is not to be written, the Small Write is also used regardless of the number of disks to be written. It consists of reading the old data on the disks to be written and the parity, computing the new parity and writing the new parity and the new data. The new parity is the bit-wise XOR of the old data, the new data and the old parity.

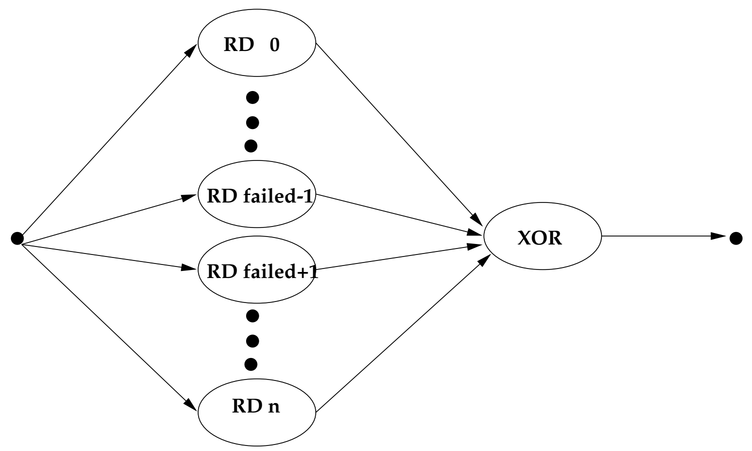

Figure 2: DAG2, Degraded Read DAG.

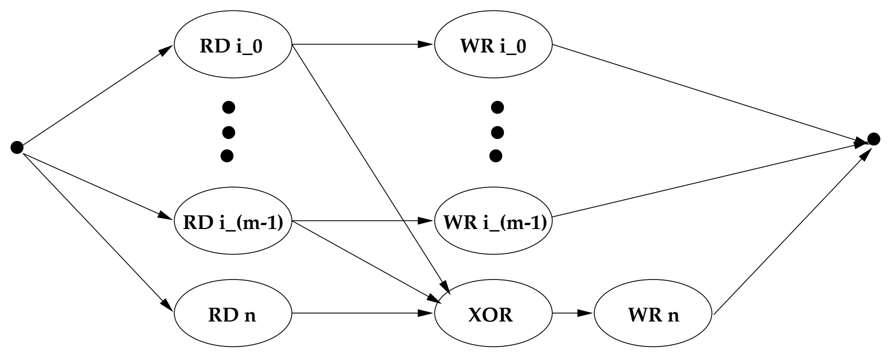

Figure 3: DAG3, Small Write DAG

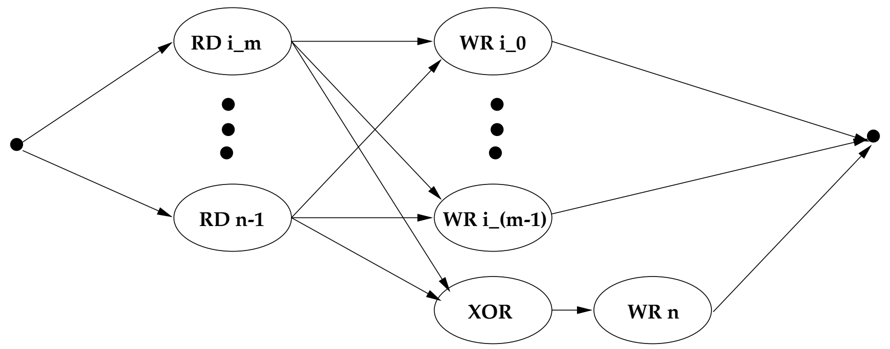

Figure 4: DAG4, Large Write DAG, no failure

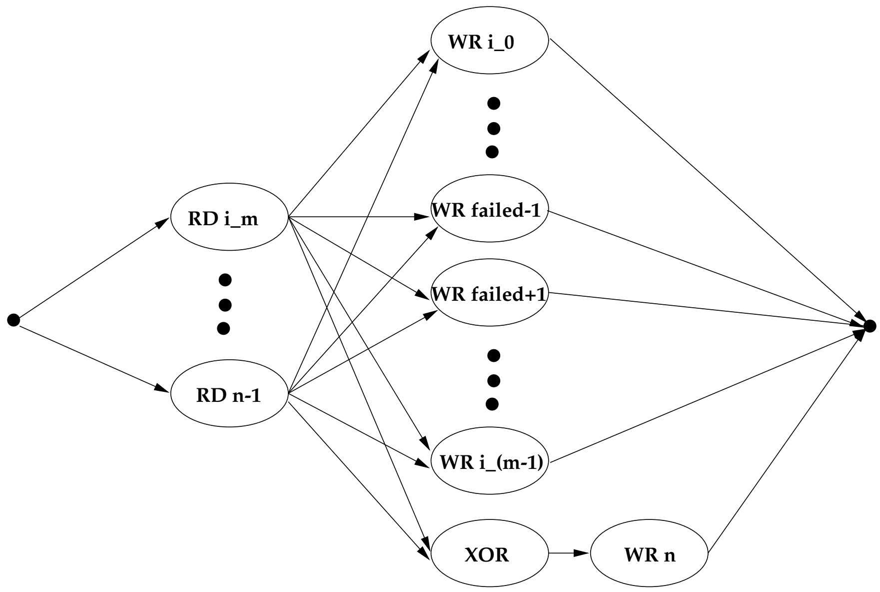

Figure 5: DAG5, Large Write DAG, failure in disk to be written.

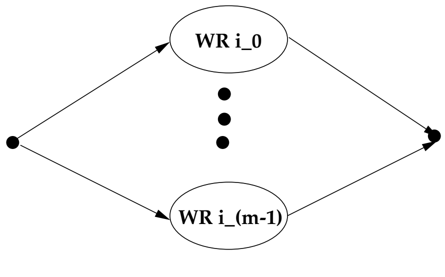

Figure 6: DAG6, Large Write DAG, parity failure.

- **Large Write, absence of failure** (DAG4, Figure 4) DAG4 is chosen when there are no failures and more than half of the array is to be written. DAG4 consists of reading the data from the disks that are not to be written, computing the parity from the data read and the data to be written and writing the new parity and data. In DAG4, there is an antecedence from each read to each write. Without these antecedences, there exists an execution of the DAG that leaves the disk array in an inconsistent state and no DAG can restore consistency.

- **Large Write, failure in disk to be written** (DAG5, Figure 5) In the presence of a failure in a disk that is to be written, DAG5 is used regardless of the number of disks to be written. DAG5 is identical to DAG4 except that the failed disk is not written.

- **Large Write, failure in parity disk** (DAG6, Figure 6) DAG6 is used when the parity disk has failed. It consists of writing the disks to be written directly.

5 Conventions

In this section, we introduce the notation we use throughout the paper. We use the type $\mathcal{V} = \{0,1\}$ to denote values of bits read or written. The symbol $\perp$ denotes the undefined value, and $\perp^+$ denotes a marked version of the undefined value. The usage of the latter will become apparent when we introduce the models. The type $\mathcal{V}^+$ denotes $\mathcal{V} \cup \{\perp^+\}$. We define the ordering relation $\preceq$ on $\mathcal{V}^+ \cup \{\perp\}$ as follows.

- $v \preceq v' \Leftrightarrow (v = v') \vee (v \in \{\perp, \perp^+\} \wedge v' \in \{0,1\})$

We next define types needed for the indices of disks. We use $\mathcal{I}$ to denote the set $\{0, ..., n-1\}$ and $\mathcal{I}_n$ the set $\mathcal{I} \cup \{n\}$. $\mathcal{B}$ is the set of all subsets of $\mathcal{I}$, and $\mathcal{B}_n$ the set of all subsets of $\mathcal{I}_n$. We define the following operators, for $B \in \mathcal{B}$,

- $B^{co} = \mathcal{I} - B$.

- $B_n = B \cup \{n\}$.

We use the notation B_n^{co} to denote $(B^{co})_n$. We use the shorthand notation B/j to denote $B/\{j\}$, for $B \in \mathcal{B}_n$ and $j \in \mathcal{I}_n \cup \{none\}$. In this notation, j intuitively represents a disk that has failed. The value *none* represents no failure. So B/j intuitively means all indices in B except the one that has failed, if any.

Finally, $\mathcal{P}$ is the set of all partial functions from $\mathcal{I}$ to $\mathcal{V}^+$. Likewise, $\mathcal{P}_n$ is the set of all partial functions from $\mathcal{I}_n$ to $\mathcal{V}^+$. We use the symbol $\perp$ to represent the undefined value for a partial function. P_0 of type $\mathcal{P}$ is such that $\forall i \in \mathcal{I}$, $P_0[i] = \perp$. Similarly, P_{n0} of type $\mathcal{P}_n$ is such that $\forall i \in \mathcal{I}_n$, $P_{n0}[i] = \perp$.
We also define the following functions.

- For $P \in \mathcal{P}_n$, $indices(P) = \{i \mid P[i] \in \mathcal{V}\}$, and

- $\bigoplus P = \begin{cases} \bigoplus_{i \in indices(P)} P[i] & \text{if } indices(P) \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases}$

We also introduce the following shortcuts for the code in our models.

- For $P \in \mathcal{P}_n$, $B \in \mathcal{B}_n$ and $v \in \mathcal{V}^+$, $P(B) := v$ is equivalent to the following piece of code:
for all $i \in B$ **do** $P[i] := v$ **od** .

Env

Signature

Inputs:
$ReadBack(P)$ $\quad P \in \mathcal{P}$
$WriteOK$

Internals:

Outputs:
$Read(B)$ $\quad B \in \mathcal{B}$
$Write(P)$ $\quad P \in \mathcal{P}$

State

$ready$ — Boolean, initially *true*.

Transitions

$Read(B)$
Pre:
$ready = true$
Eff:
$ready = false$

$ReadBack(P)$
Eff:
$ready = true$

$Write(P)$
Pre:
$ready = true$
Eff:
$ready = false$

$WriteOK$
Eff:
$ready = true$

Figure 7: I/O Automaton for the environment

- For $P \in \mathcal{P}_n$, $Q \in \mathcal{P}$, $P := Q$ is equivalent to the following piece of code:
  for all $i \in \mathcal{I}$ **do** $P[i] := Q[i]$ **od** ; $P[n] := \perp$

6 Specification

In this section, we describe the specification, *Spec*, for the system. *Spec* makes the assumption that there are n bits, indexed from 0 to $n-1$ that can be read or written. It captures the property that the value returned on a read from a bit corresponds to the last write to that bit, or an arbitrary value if no write has been performed. It interacts with an environment automaton *Env*. Section 6.1 presents the environment. Section 6.2 presents the specification and finally Section 6.3 presents some properties of the specification composed with its environment.

6.1 Environment

Figure 7 shows the environment of the specification. The automaton has the following interface.

- Inputs:
  - $ReadBack(P)$, where $P \in \mathcal{P}$.
  - $WriteOK$.
- Outputs:
  - $Read(B)$, where $B \in \mathcal{B}$.
  - $Write(P)$, where $P \in \mathcal{P}$.

The environment *Env* submits one outstanding request ($Read(B)$, $Write(P)$) at a time, that is, before submitting the next request the environment waits for a response ($ReadBack(P)$, $WriteOK$) from *Spec*. The environment may read or write bits indexed from *0* to *n-1*.

6.2 Specification

Figure 8 presents the model for the specification. Underlined variables and statements are not part of the basic model. These variables are history variables and will be introduced below. The model has the following external interface.

- Inputs:
  - $Read(B)$, where $B \in \mathcal{B}$.
  - $Write(P)$, where $P \in \mathcal{P}$.
- Outputs:
  - $ReadBack(P)$, where $P \in \mathcal{P}$.
  - *WriteOK*.

Spec has the following state variables.

- *Bit* is an array of n bits, indexed from 0 to $n-1$, initially arbitrary.
- *ReadPairs* is of type $\mathcal{P}$ and is initially P_0. It is used to record what bits need to be read.
- *WritePairs* is of type $\mathcal{P}$ and is initially P_0. It is used to record what bits and values need to be written.
- *pc* ranges over {*idle,read,write*} and is initially *idle*.

Spec has the following transitions.

- $Read(B)$ is an input from the environment. It has the effect of placing a placeholder $\perp^+$ in *ReadPairs* for every index that needs to be read.
- Upon receiving a $Read(B)$ input, the automaton performs a series of internal $read(i)$ actions to perform the read. Each such action reads $Bit[i]$ and records that value in *ReadPairs*.
- When all bits that had to be read have been read, the automaton performs a $ReadBack(P)$ output, where P is identical to *ReadPairs*. This action resets state variables.
- $Write(P)$ is the other input from the environment. It has the effect of setting the variable *WritePairs* to P, thereby recording which bits need to be written with what values.
- Upon receiving a $Write(P)$ input, the automaton performs a series of internal $write(i,v)$ actions to perform the writes. Each such action writes value v to $Bit[i]$ and sets $WritePairs[i]$ to $\perp$.
- When *WritePairs* has no more values to be written, the automaton performs a *WriteOK* output, which sets *pc* back to *idle*.

We add the following history variables to *Spec*. The modified automaton is shown in Figure 8. The changes are shown by underlining. These variables are useful for the main proof of correctness.

Spec

Signature

Inputs:

Read(B)	$B \in \mathcal{B}$
Write(P)	$P \in \mathcal{P}$

Internals:

read(i)
write(i,v)

Outputs:

ReadBack(P)	$P \in \mathcal{P}$
WriteOK	

State

Bit	Array of n bits, indexed from 0 to *n-1*, initially arbitrary.
ReadPairs	$\mathcal{P}$, initially P_0.
WritePairs	$\mathcal{P}$, initially P_0.
pc	$\{idle, read, write\}$, initially *idle*.
<u>*Indices*</u>	$\mathcal{B}$, initially empty.
<u>*WritePairsPerm*</u>	$\mathcal{P}$, initially P_0.

Transitions

Read(B)
Eff:
$ReadPairs(B) := \perp^+$
$pc := read$
$\underline{Indices := B}$

read(i)
Pre:
$pc = read$
$ReadPairs[i] = \perp^+$
Eff:
$ReadPairs[i] := Bit[i]$

ReadBack(P)
Pre:
$pc = read$
$\forall i \in \mathcal{I},\ ReadPairs[i] \neq \perp^+$
$P = ReadPairs$
Eff:
$ReadPairs := P_0$
$pc := idle$
$\underline{Indices := \{\}}$

Write(P)
Eff:
$WritePairs := P$
$pc := write$
$\underline{Indices := indices(P)}$
$\underline{WritePairsPerm := P}$

write(i,v)
Pre:
$pc = write$
$WritePairs[i] = v$, where $v \in \mathcal{V}$
Eff:
$WritePairs[i] := \perp$
$Bit[i] := v$

WriteOK
Pre:
$pc = write$
$WritePairs = P_0$
Eff:
$pc := idle$
$\underline{Indices = \{\}}$
$\underline{WritePairsPerm := P_0}$

Figure 8: I/O Automaton for *Spec*.

- *Indices* of type $\mathcal{B}$, initially empty. Actions *Read*(B) and *Write*(P) set it and *ReadBack*(P) and *WriteOK* reset it. It records what bits need to be read during a read operation and what bits need to be written during a write.

- *WritePairsPerm* of type $\mathcal{P}$, initially P_0. It records bits and values to be written during a write operation. It does not change as the write progresses. Actions *Write*(P) and *WriteOK* modify it.

6.3 Properties of *Spec*

We use *Spec'* to denote the composition of *Spec* with *Env*. *Spec'* has the following properties. The symbol * indicates that the lemma is used in the main proof of correctness (Section 8.4). The first lemma is a basic lemma. Part 1 states that for all $i \in \mathcal{I}$, *WritePairs*$[i]$ is not $\perp^+$. Part 2 states that the variables *ReadPairs* and *WritePairs* are undefined at indices that are not to be read or written.

Lemma 6.1 * *In all reachable states of Spec'*,

1. *For all* $i \in \mathcal{I}$, *WritePairs*$[i] \neq \perp^+$.
2. *For all* $i \notin$ *Indices*, *ReadPairs*$[i]$ = *WritePairs*$[i] = \perp$.

The first part of the second lemma states the fact that if *Spec'* has read a value and stored it in *ReadPairs*, then this value is correct, that is, it is the same value as what is stored in *Bit*. The following invariant states the fact that if *Spec'* has written a value it was supposed to write, then *Bit* contains the value written. Finally, the third part states that if there is a value to be written in *WritePairs* then this value is the same as the corresponding one in *WritePairsPerm*.

Lemma 6.2 * *In all reachable states of Spec', for all* $i \in \mathcal{I}$,

1. *If ReadPairs*$[i] \in \mathcal{V}$, *then ReadPairs*$[i]$ = *Bit*$[i]$.
2. *If WritePairsPerm*$[i] \in \mathcal{V}$, *and WritePairs*$[i] = \perp$, *then Bit*$[i]$ = *WritePairsPerm*$[i]$.
3. *If WritePairs*$[i] \in \mathcal{V}$, *then WritePairsPerm*$[i]$ = *WritePairs*$[i]$.

7 Model for the RAID Level 5 System

In this section, we give our model for the RAID Level 5 System. Section 7.1 presents some assumptions. Section 7.2 presents the model and Section 7.3 presents some properties of the model composed with its environment.

7.1 Assumptions

The following lists the assumptions and justifications for these assumptions made by the RAID Level 5 model.

- The controller uses the same set of DAGs on every parity group and two DAGs on different parity groups do not interfere with each other's execution. Therefore it is sufficient to show that the controller's behavior is correct on one parity group. Thus the model assumes a single parity group consisting of $n+1$ disks, indexed from 0 to n, where disk n is the parity disk.

- All parity computations are bit-wise. Therefore the model assumes one bit per disk, and this restriction can be removed trivially.
- The model assumes at most one disk failure.

7.2 Model

We present the RAID Level 5 model in Figures 9, 10, and 11, and refer to this model as *RAID*. In the figures, underlined variables and statements are not part of the basic model. These variables are history variables and will be introduced below.

The external interface of *RAID* is the same as the one for *Spec*. The environment automaton *Env* (Figure 7) presented earlier interacts with *RAID*.

RAID has the following state variables.

- *Disk* is an array of *n+1* bits, indexed from *0* to *n*. This variable models a single parity group in a physical RAID system. *Disk*[*n*] models the parity. The values in *Disk* are initially such that: $\bigoplus_{0\leq i\leq n} \mathit{Disk}[i] = 0$.
- *Indices* is of type $\mathcal{B}$, initially empty, and is used to hold the indices of disks to be written or read.
- *ReadPairs* is of type $\mathcal{P}_n$, initially P_{n0}. It is used to remember values read from *Disk*. It is also used to indicate which indices need to be read.
- *WritePairs* is of type $\mathcal{P}_n$, initially P_{n0}. It is used to remember values that are to be written to *Disk* and that have not been written yet.
- *WritePairsPerm* is of type $\mathcal{P}$, initially P_0. It is used to remember the values that are to be written to disks other than the parity, for the duration of a write operation.
- *DAG* ranges over {*none*,*chooseR*,*chooseW*,*1*,*2*,*3*,*4*,*5*,*6*} and indicates which DAG is currently running or whether a DAG is to be chosen. It is set to *none* when the automaton is idle, which is also its initial value.
- *f* is of type $\mathcal{I}_n \cup \{\mathit{none}\}$ and indicates which disk has failed. If it is equal to *none*, then no disk has failed. A failed disk cannot be read or written any further. This models the catastrophic failure of disks. *f* is initially *none*.

We define the following derived variables for *RAID*, which are used in the statement of properties and proofs in subsequent sections.

- $\mathit{All} = \mathcal{I}_n / f$.
- *StartedWriting*, is a boolean equivalent to $\exists i \in \mathit{Indices},\ \mathit{WritePairs}[i] \neq \mathit{WritePairsPerm}[i]$.

RAID does not represent DAGs explicitly. DAG nodes are represented by the actions *read*(*i*), *write3*(*i*,*v*), *write45*(*i*,*v*) and *write6*(*i*,*v*). DAG precedences are represented in the preconditions of low-level writes. DAG selection is done using actions *chooseDAG1*, through *chooseDAG6*. Note that XOR nodes in the DAGs are not present in the model as separate actions, the XORs are computed in the preconditions of low-level writes to *Disk*[*n*].

We now explain how *RAID* works.

RAID

Signature

Inputs:

Read(B)	$B \in \mathcal{B}$
Write(P)	$P \in \mathcal{P}$

Internals:

read(i), *fail*(i)	$i \in \mathcal{I}_n$
write3(i,v), *write45*(i,v), *write6*(i,v)	$i \in \mathcal{I}_n$, $v \in \mathcal{V}$
chooseDAG1, *chooseDAG2*	
chooseDAG3, *chooseDAG4*	
chooseDAG5, *chooseDAG6*	

Outputs:

ReadBack(P)	$P \in \mathcal{P}$
WriteOK	

State

Disk	Array of n+*1* bits, indexed from 0 to n, initially such that $\bigoplus_i Disk[i] = 0$.
Indices	B, initially empty.
ReadPairs	$\mathcal{P}_n$, initially P_{n0}.
WritePairs	$\mathcal{P}_n$, initially P_{n0}.
WritePairsPerm	$\mathcal{P}$, initially P_0.
DAG	{*none*, *chooseR*, *chooseW*, *1*, *2*, *3*, *4*, *5*, *6*}, initially *none*
f	$\mathcal{I}_n \cup \{none\}$, initially *none*.
<u>*VD*</u>	$\mathcal{V}$, initially 0.
<u>*Rec*</u>	Boolean, initially *false*.

Transitions

Read(B)
Eff:
$Indices := B$
$ReadPairs(B) := \perp^+$
$DAG := chooseR$

chooseDAG1
Pre:
$DAG = chooseR$
$f \notin Indices$
Eff:
$DAG := 1$

chooseDAG2
Pre:
$DAG = chooseR$
$f \in Indices$
Eff:
$DAG := 2$
$ReadPairs(Indices_n^{co}) := \perp^+$

read(i)
Pre:
$DAG \in \{1,2,3,4,5\}$
$i \neq f$
$ReadPairs[i] = \perp^+$
Eff:
$ReadPairs[i] := Disk[i]$

ReadBack(P)
Pre:
$DAG \in \{1,2\}$
$\forall i \in Indices/f,\ P[i] = ReadPairs[i]$
$\forall i \in Indices^{co},\ P[i] = \perp$
$\forall i \in Indices/f,\ ReadPairs[i] \in \mathcal{V}$
if $DAG = 2$
then $\forall i \in Indices_n^{co}\ ReadPairs[i] \in \mathcal{V}$
$P[f] = \bigoplus ReadPairs$
fi
Eff:
$ReadPairs := P_{n0}$
$Indices := \{\}$
$DAG := none$
<u>$Rec := false$</u>

Figure 9: I/O Automaton for RAID with history variables.

RAID *(Continued)*

$Write(P)$
Eff:
$WritePairs$:= P
$WritePairsPerm$:= P
$Indices := indices(P)$
$DAG := chooseW$

$chooseDAG3$
Pre:
$DAG =$ chooseW
$f = none \wedge |Indices| \leq n/2$
$\vee\ f \in Indices^{co}$
Eff:
$DAG := 3$
$ReadPairs(Indices_n) := \perp^+$
$WritePairs[n] := \perp^+$

$chooseDAG4$
Pre:
$DAG = chooseW$
$f = none \wedge n/2 < |Indices|$
Eff:
$DAG := 4$
$ReadPairs(Indices^{co}) := \perp^+$
$WritePairs[n] := \perp^+$

$chooseDAG5$
Pre:
$DAG = chooseW$
$f \in Indices$
Eff:
$DAG := 5$
$ReadPairs(Indices^{co}) := \perp^+$
$WritePairs[n] := \perp^+$

$chooseDAG6$
Pre:
$DAG = chooseW$
$f = n$
Eff:
$DAG := 6$

$write3(i,v)$
Pre:
$DAG = 3$
$i \neq n$
$i \neq f$
$WritePairs[i] = v,\ v \in \mathcal{V}$
$ReadPairs[i] \neq \perp^+$
Eff:
$WritePairs[i] := \perp$
$Disk[i] := v$

$write3(n,v)$
Pre:
$DAG = 3$
$n \neq f$
$WritePairs[n] = \perp^+$
$v = \bigoplus ReadPairs$
$\oplus \bigoplus WritePairsPerm$
$\forall i \in Indices_n$
$ReadPairs[i] \neq \perp^+$
Eff:
$WritePairs[n] := \perp$
$Disk[n] := v$

$write45(i,v)$
Pre:
$DAG \in \{4,5\}$
$i \neq n$
$i \neq f$
$WritePairs[i] = v,\ v \in \mathcal{V}$
$\forall i \in Indices^{co}$
$ReadPairs[i] \neq \perp^+$
Eff:
$WritePairs[i] := \perp$
$Disk[i] := v$

$write45(n,v)$
Pre:
$DAG \in \{4,5\}$
$n \neq f$
$WritePairs[n] = \perp^+$
$v = \bigoplus ReadPairs$
$\oplus \bigoplus WritePairsPerm$
$\forall i \in Indices^{co}$
$ReadPairs[i] \neq \perp^+$
Eff:
$WritePairs[n] := \perp$
$Disk[n] := v$

$write6(i,v)$
Pre:
$DAG = 6$
$i \neq n$
$WritePairs[i] = v,\ v \in \mathcal{V}$
Eff:
$WritePairs[i] := \perp$
$Disk[i] := v$

Figure 10: I/O Automaton for RAID with history variables *(Continued)*.

RAID *(Continued)*

WriteOK
Pre:
$DAG \in \{3,4,5,6\} \wedge$
$WritePairs = P_{n0}$
Eff:
$Indices := \{\}$
$ReadPairs := P_{n0}$
$WritePairsPerm := P_0$
$DAG := none$
if $f \in Indices$
then $VD := WritePairsPerm[f]$
$Rec := false$

fail(i)
Pre:
$f = none$
Eff:
$f := i$
if $ReadPairs[i] = \perp^+ \vee WritePairs[i] \neq \perp$
then if $DAG \in \{chooseR, 1\}$
    then $ReadPairs(Indices) := \perp^+$
        $DAG := chooseR$
    fi
    if $DAG \in \{chooseW, 3, 4\}$
    then $ReadPairs := P_{n0}$
        $WritePairs := WritePairsPerm$
        $DAG := chooseW$
    fi
    $Rec := true$
fi
if $i \neq n$ **then** $VD := Disk[i]$ **fi**

Figure 11: I/O Automaton for RAID with history variables *(Continued)*.

- *Read*(B). When *RAID* receives a *Read*(B) input, it records which indices are to be read in the variable *Indices*. It also sets *ReadPairs*[i], for $i \in B$, to $\perp^+$. The symbol $\perp^+$ is used as a placeholder. The values read from the disk array will be placed into this state variable and $\perp^+$ will be replaced with values read.

- *Write*(P). When *RAID* receives a *Write*(P) input, it records which indices are to be written along with corresponding values in variable *WritePairs*.

- *chooseDAG*. After receiving an input, *RAID* proceeds to choosing a DAG to execute. The selection criteria appear in the precondition of each *chooseDAG* action. DAG1 is selected if the operation is a read and there is no failure among disks to be read. DAG2 is selected if the operation is a read and there is a failure among disks to be read. DAG3 is selected if the operation is a write and either there is no failure and the number of disks to be written is fewer than half of the number of disks in the disk array, or if there is a failure in a disk not to be written. DAG4 is selected if the operation is a write, there is no failure, and the number of disks to be written is greater than half. DAG5 is chosen if there is a failure in a disk to be written. Finally, DAG6 is selected if the parity disk has failed.

  chooseDAG actions may change *ReadPairs* by putting placeholders for indices to be read. They may also change *WritePairs*[n] to $\perp^+$, which signifies that *Disk*[n] needs to be written. When *Disk*[n] is actually written, then *WritePairs*[n] is set back to $\perp$.

- *read*(i). The precondition for this action includes *ReadPairs*[i] $= \perp^+$, which means that this low-level read needs to be performed and has not been performed yet. The action records the value read in place of the placeholder $\perp^+$.

- *write3*(i,v), *write45*(i,v), *write6*(i,v). The preconditions for these actions include *WritePairs*[i] $\in \mathcal{V}$, if $i \neq n$, and *WritePairs*[i] $= \perp^+$, if $i = n$. These expressions indicate that the low-level write needs to be performed and has not been performed yet. The action has the effect of setting *WritePairs*[i] to $\perp$. The precondition of this action also encodes the precedences in DAGs.

- *ReadBack*(P). If $DAG \in \{1,2\}$ and all the appropriate low-level reads have been performed, then the controller performs a *ReadBack*(P) output. In the case of DAG2, this action computes the value of disk f by taking the xor of every value in *ReadPairs*. This variable has a value in $\mathcal{V}$ for every $i \in \mathcal{I}_n$ except f. Therefore the value computed for disk f is the XOR of every other disk. The effect of the action is to reset state variables.

- *WriteOK*. If $DAG \in \{3,4,5,6\}$ and all the appropriate low-level writes have been performed, then the controller performs a *WriteOK* action. Its effect is to reset state variables.

- *fail*(i). A *fail*(i) action may occur at most once. It sets variable f to i. The test $ReadPairs[i] = \perp^+ \vee WritePairs[i] \neq \perp$ checks whether there is any low-level read or write on *Disk*[i] that needs to be executed. In that case, the action stops the execution of the current DAG by changing DAG to either *chooseR* or *chooseW*, which causes a new DAG to be chosen using the same rules as before. Otherwise, DAG remains unchanged.

We add the following history variables to *RAID*. The changes are shown in Figures 9, 10, and 11 by underlining.

- *VD* of type $\mathcal{V}$ stands for Virtual Disk and is initially 0. It is used to keep the last value written to a disk, except the parity disk, if it has failed. *VD* is updated as indicated in the figure.

- *Rec*, is a boolean, initially *false*. It is *true* when a DAG has stopped execution because of a failure and a second DAG is running to complete the operation. *Rec* is updated by *fail*(i), *ReadBack*(P) and *WriteOK*.

7.3 Properties of *RAID*$'$

We use *RAID*$'$ to denote the composition of *RAID* and *Env*. In this section, we give some properties of *RAID*$'$. The symbol * indicates the lemmas used in the main proof of correctness (Section 8.4). The Lemmas not marked with this symbol are used in the proof of the Consistency Lemma presented in Section 8.1. The proofs for these Lemmas are simple inductive arguments, which require the introduction of other very simple Lemmas.

Lemma 7.1 presents some basic properties of *RAID*$'$.

Lemma 7.1 *In all reachable states of RAID$'$,*

1. *If $ready = true$, then $DAG = none$.*
2. * *If $f = none$, then $Rec = false$.*
3. *If $DAG = chooseW$ then $StartedWriting = false$.*

Lemma 7.2 gives some properties concerning the variable *ReadPairs*. Invariant 1 states that during the execution of DAGs 1 and 2, if the controller has read a value, then this value is equal to the value on the corresponding disk. Invariant 2 states that during DAG2, the failed disk is not read. Invariant 3 presents the fact that during DAGs 4 and 5, the controller does not read disks to be written. Invariant 4 says that if action *WriteOK* is enabled, i.e., at the end of execution of a write DAG, no read of the disk array is enabled. Finally, Invariant 5 states that during the execution of DAG 4, if the DAG has started writing, then all disks not to be written have been read, and the values read are equal to the corresponding values on the disk, or to *VD* if a disk has failed.

Lemma 7.2 *In all reachable states of $RAID'$,*

1. * *If $DAG \in \{1,2\}$, and for $i \in \mathcal{I}_n/f$, $ReadPairs[i] \in \mathcal{V}$, then $ReadPairs[i] = Disk[i]$.*
2. * *If $DAG = 2$, then $ReadPairs[f] = \perp^+$.*
3. *If $DAG \in \{4,5\}$, then for all $i \in Indices_n$, $ReadPairs[i] = \perp$.*
4. *If $WriteOK$ is enabled, then for all $i \in \mathcal{I}_n$, $ReadPairs[i] \neq \perp^+$.*
5. *If $DAG = 4 \wedge StartedWriting$ then for all $i \in Indices^{co}$:*
$$ReadPairs[i] = \begin{cases} Disk[i] & \text{if } i \neq f, \\ VD & \text{otherwise.} \end{cases}$$

Lemma 7.3 gives some properties concerning *WritePairs*. Invariant 1 states that for all i not equal to n, $WritePairs[i]$ is never set to $\perp^+$. Invariant 2 presents the fact that if there is a value in *WritePairs* that has not been written yet, then this value is the same as the corresponding value in *WritePairsPerm*. Invariant 3 states that during the execution of DAG5, the failed disk is not written. Invariant 4 says that when $DAG \in \{chooseW,3,4\}$, if a read of a disk to be written is enabled, then the write to that disk has not been performed yet. Finally, Invariant 5 states that the controller only writes indices that are to be written.

Lemma 7.3 *In all reachable states of $RAID'$,*

1. * $\forall i \in \mathcal{I}$, $WritePairs[i] \neq \perp^+$.
2. * *If for some $i \in \mathcal{I}$, $WritePairs[i] \in \mathcal{V}$, then $WritePairsPerm[i] = WritePairs[i]$.*
3. * *If $DAG = 5$, then $WritePairs[f] \in \mathcal{V}$.*
4. * *If $DAG \in \{chooseW,3,4\}$, and for some $i \in Indices$, $ReadPairs[i] = \perp^+$, then $WritePairs[i] \in \mathcal{V}$.*
5. *If for $i \in \mathcal{I}$, $WritePairs[i] \in \mathcal{V}$, then $i \in Indices$.*

Lemma 7.4 states properties concerning the *f* variable. Invariant 1 says that if DAG6 is executing then the parity disk has failed. Invariant 2 states that if DAG 2 or 5 is running then a disk in *Indices* has failed.

Lemma 7.4 *In all reachable states of $RAID'$,*

1. *If $DAG = 6$, then $f = n$.*
2. *If $DAG \in \{2,5\}$, then $f \in Indices$.*

Lemma 7.5 gives a property concerning the state of DAG 3, or DAG 4 when it has started writing. The expression "for some $i \in \mathcal{I}_n$, $ReadPairs[i] = \perp^+ \vee WritePairs[i] \neq \perp$", implies that there is a low-level read or write to disk i to be done in this state. The invariant states that i must be in $Indices_n$. For DAG3 this is easy to see, since DAG3 only reads and writes disks with indices in $Indices_n$. For DAG4, this is true because the DAG has started writing and the precedences are such that all the reads to indices not in $Indices_n$ have been performed.

Lemma 7.5 *In all reachable states of $RAID'$, if $DAG = 3 \vee (DAG = 4 \wedge StartedWriting)$ and for some $i \in \mathcal{I}_n$, $ReadPairs[i] = \perp^+ \vee WritePairs[i] \neq \perp$, then $i \in Indices_n$.*

Lemma 7.6 presents properties concerning the *Disk* variable. Invariant 1 states the fact that if the parity disk has been written, then $Disk[n]$ contains the value that is indicated. Invariant 2 says that, for write DAGs, if a write has been performed, then the disk contains the value written, if the disk has not failed.

Lemma 7.6 *In all reachable states of RAID′,*

1. *If* $DAG \in \{3,4,5\}$ *and* $WritePairs[n] = \perp$*, then* $Disk[n] = \bigoplus ReadPairs \oplus \bigoplus WritePairsPerm$.

2. *If* $DAG \in \{3,4,5,6\}$ *and for* $i \in Indices/f$, $WritePairs[i] = \perp$*, then* $Disk[i] = WritePairsPerm[i]$.

8 Proof of Correctness

This section presents the proof of correctness. Section 8.1 presents the key invariant called *Consistency*. Section 8.2 gives the simulation relation to be proved. Section 8.3 gives the step correspondence of the proof. Finally Section 8.4 presents the proof of the simulation relation.

8.1 Consistency

In this section, we present and prove the Consistency property. Informally, a parity group with no failure is consistent, if the XOR of all bits is equal to 0. If there is a failure at a disk other that the parity disk, then the XOR of all bits, except the one that has failed, is equal to the last value written to the failed bit. Thus consistency can be expressed as: If $f \neq$ n, then $\bigoplus_{i\in All} Disk[i] = VD$. Note that if there is no failure then $VD = 0$.

The Consistency property consists of two parts as indicated in Lemma 8.1. The first part expresses under what conditions the parity group is consistent. These conditions are: when the controller is idle or doing a read operation, when the controller is about to choose a write DAG and the failed disk, if any, is not among the disks to be written, and when DAG4 is executing and it has not started writing. This last condition is true because of the dependencies in DAG4 that force all the writes to occur after all the reads.

When the controller is executing a write DAG and it has started writing, the parity group is no longer consistent. The second part of Lemma 8.1 expresses an invariant relevant to the execution of write DAGs 3,4, and 5, that is needed to restore the parity group to a consistent state when the write operation is done. Note that the parity group is trivially consistent after the execution of DAG6, since $f = n$ in that case.

Informally, the second part expresses the fact that the parity group resulting from the execution of a write DAG is in a consistent state. A write DAG has finished executing if action *WriteOK* is enabled. In such a state we have the following.

$$\bigoplus_{i\in All} Disk[i] = \begin{cases} VD & \text{if } f \notin Indices, \\ v & \text{otherwise, where } v = WritePairsPerm[f] \end{cases}$$

We have the following.

$$\begin{aligned} \bigoplus_{i\in All} Disk[i] &= \bigoplus_{i\in Indices/f} Disk[i] \oplus \bigoplus_{i\in Indices^{co}/f} Disk[i] \oplus Disk[n] \\ &= \bigoplus_{i\in Indices/f} Disk[i] \oplus \bigoplus_{i\in Indices^{co}/f} Disk[i] \\ &\quad \oplus \bigoplus ReadPairs \oplus \bigoplus WritePairsPerm, \text{by Part 1 of Lemma 7.6.} \end{aligned}$$

Part 2 of Lemma 7.6 implies that

$$\bigoplus_{i \in Indices/f} Disk[i] \oplus \bigoplus WritePairsPerm = \begin{cases} 0 & \text{if } f \notin Indices, \\ v & \text{otherwise, where v} = WritePairsPerm[f]. \end{cases}$$

Therefore we have that

$$\bigoplus_{i \in Indices^{co}/f} Disk[i] \oplus \bigoplus ReadPairs = \begin{cases} VD & \text{if } f \notin Indices, \\ 0 & \text{otherwise.} \end{cases}$$

When a write DAG is executing, *ReadPairs* gets its values from the disks. This motivates the second part of the Consistency Lemma.

Lemma 8.1 * Consistency *In all reachable states of RAID′, if* $n \neq f$*, then:*

1. *If* $DAG \in \{none, chooseR, 1, 2\} \vee (DAG = chooseW \wedge f \notin Indices) \vee (DAG = 4 \wedge \neg\, StartedWriting)$*, then* $\bigoplus_{i \in All} Disk[i] = VD$*.*
2. *If* $DAG \in \{3,5\} \vee (DAG = 4 \wedge StartedWriting)$*, then*

$$\bigoplus ReadPairs \oplus \bigoplus_{ReadPairs[i] = \perp^+} Disk[i] \oplus \bigoplus_{i \in Indices^{co}/f} Disk[i] = \begin{cases} VD & \text{if } f \notin Indices \\ 0 & \text{otherwise} \end{cases}$$

In the following proof we refer to the statements above as Invariant 1 and Invariant 2.

Proof. Initially, $DAG = none$, $\bigoplus_i Disk[i] = 0$, $f = none$, and $VD = 0$. Therefore Consistency is satisfied in any initial state.

We show that each step (s,π,s') of $RAID'$ preserves the invariant. Consider the following cases. In the following, a variable name not preceded by the name of a state, stands for the variable in state s.

Case: $\pi \in \{Read(B), Write(P)\}$
Since $s.ready = true$, then by Invariant 1 of Lemma 7.1, $s.DAG = none$. By the inductive hypothesis, Invariant 1 is true in s. This invariant is trivially preserved in s'.

Case: $\pi = read(i)$
Consider the following cases.

Subcase: $s.DAG \in \{1,2\} \vee s.DAG = 4 \wedge \neg\, StartedWriting$
In this case, the invariant is trivially preserved.

Subcase: $s.DAG \in \{3,5\} \vee s.DAG = 4 \wedge StartedWriting$
In this case, Invariant 2 is true in s. The effect of π is such that this invariant is preserved.

Case: $\pi = chooseDAG3$
By the precondition of π, $s.DAG = chooseR$. Therefore Invariant 1 is true in s. Action π sets DAG to 3, and assigns $\perp^+$ to all $ReadPairs[i]$ such that $i \in Indices_n$. We also know that $f \notin Indices_n$. We have that $\bigoplus s'.ReadPairs = 0$. Therefore Invariant 2 is true in s'.

Case: $\pi = chooseDAG4$
Action π sets DAG to 4. By Invariant 3 of Lemma 7.1, $s.StartedWriting$ is false. We know that $s.DAG = chooseW$, and $s.f = none$. By the inductive hypothesis, Invariant 1 is true, and this invariant is preserved in s'.

Case: π = *chooseDAG5*
Action π sets DAG to 5, and assigns $\perp^+$ to all $ReadPairs[i]$ such that $i \in Indices^{co}$. We have that $\bigoplus s'.ReadPairs = 0$ and $f \in Indices$. Therefore Invariant 2 is true in s' and the invariant is preserved in this case.

Case: π = *write3*(i,v)
We know that $s.ReadPairs[i] \neq \perp^+$, also $i \in Indices$, by Invariant 5 of Lemma 7.3. Therefore action π preserves the invariant trivially.

Case: π = *write45*(i,v)
By Lemma 5, $i \in Indices$ and by Lemma 3 $ReadPairs[i] = \perp$. Consider the following cases.

Subcase: $s.DAG = 5 \vee s.DAG = 4 \wedge StartedWriting$
The invariant is preserved trivially in this case.

Subcase: $s.DAG = 4 \wedge \neg\, StartedWriting$
In this case, Invariant 1 is true in s. Invariant 5 of Lemma 7.2 implies that

$$\bigoplus ReadPairs \oplus \bigoplus_{i \in Indices^{co}/f} Disk[i] = \begin{cases} VD & \text{if } f \in Indices/f \\ 0 & \text{otherwise} \end{cases}$$

By the precondition of π, $\forall j \in Indices/f$, $ReadPairs[j] \neq \perp^+$. Thus Invariant 2 is true in s'.

Case: π = *write6*(i,v)
Since $f = n$, by Lemma 1, the invariant is preserved trivially.

Case: π = *WriteOK*
By Lemma 4, for all $i \in \mathcal{I}_n$, $s.ReadPairs[i] \neq \perp^+$. By the inductive hypothesis, Invariant 2 is true in s. In the following, we use Lemmas 1 and 2.

$$\begin{aligned} \bigoplus_{i \in All} s.Disk[i] &= \bigoplus_{i \in Indices/f} Disk[i] \oplus \bigoplus_{i \in Indices^{co}/f} Disk[i] \\ &\quad \oplus \bigoplus ReadPairs \oplus \bigoplus WritePairsPerm \\ &= \begin{cases} WritePairsPerm[f] & \text{if } f \in Indices \\ VD & otherwise \end{cases} \end{aligned}$$

If $f \in Indices$, then $WritePairsPerm[f]$ is written to $s'.VD$. Otherwise VD does not change with action π and Invariant 1 is true in s'. Thus the invariant is satisfied in this case.

Case: π = *fail*(i)
We have that $s.f = none$. Action π sets VD to $Disk[i]$. Note that $s.DAG \notin \{2,5,6\}$. Consider the following cases.

Subcase: $DAG \in \{none, chooseR, chooseW, 1\} \vee DAG = 4 \wedge \neg\, StartedWriting$.
In this case, Invariant 1 is true in s and is preserved in s'.

Subcase: $DAG = 3 \vee DAG = 4 \wedge StartedWriting$
In this case, Invariant 2 is true in s. First assume that $ReadPairs[i] \neq \perp^+ \wedge WritePairs[i] = \perp$. In this case Invariant 2 is preserved in s'.

Now assume that $ReadPairs[i] = \perp^+ \vee WritePairs[i] \neq \perp$
By Lemma 7.5, $i \in Indices_n$. Since $s'.DAG = chooseW$, the invariant is trivially satisfied in this case.

■

8.2 Simulation Relation

Let s and u be states of $RAID'$ and $Spec'$ respectively, and f the following relation.

$f(s,u) \Leftrightarrow f_1(s,u) \wedge f_2(s,u) \wedge f_3(s,u) \wedge f_4(s,u) \wedge f_5(s,u)$,

where $f_1(s,u)$ through $f_5(s,u)$ are defined below.

- $f_1(s,u) \Leftrightarrow \forall\ i$ **s.t.** $0 \leq i <$ n
  if $s.f \neq i$ **then** $u.Bit[i] = s.Disk[i]$
  else $u.Bit[i] = s.VD$
- $f_2(s,u) \Leftrightarrow \forall i \in s.Indices$
  if $s.DAG \in \{chooseR,1,2\} \wedge (s.Rec = false \vee i = s.f)$
  then $s.ReadPairs[i] = u.ReadPairs[i]$
  else $s.ReadPairs[i] \preceq u.ReadPairs[i]$
- $f_3(s,u) \Leftrightarrow \forall i \in s.Indices$
  if $s.DAG \in \{chooseW,3,4,5,6\} \wedge (s.Rec = false \vee i = s.f)$
  then $u.WritePairs[i] = s.WritePairs[i]$
  else $u.WritePairs[i] \preceq s.WritePairs[i]$
- $f_4(s,u) \Leftrightarrow$ **if** $s.DAG = none$ **then** $u.\mathrm{pc} = idle$
  elseif $s.DAG \in \{chooseR,1,2\}$ **then** $u.\mathrm{pc} = read$
  else $u.\mathrm{pc} = write$
- $f_5(s,u) \Leftrightarrow u.WritePairsPerm = s.WritePairsPerm \wedge u.ready = s.ready.$

f_1 gives the correspondence between *Bit* and *Disk* variables. If disk i has failed, then $u.Bit[i] = s.VD$. f_2 and f_3 give the correspondence for *ReadPairs* and *WritePairs* variables. If the controller is running a DAG right after receiving an input ($s.Rec = false$), then the variables are equal to their counterparts in *Spec*. On the other hand, if a DAG has failed and the controller is running a second DAG to complete the operation then the variables are related to their counterparts with the $\preceq$ relation, defined previously in Section 5. In either case, if a disk i has failed, then these variables evaluated at i are equal to their counterparts. f_4 gives the correspondence between DAG and *pc*. f_5 gives some trivial equalities.

We show the following theorem.

Theorem 8.2 *f is a simulation relation from $RAID'$ to $Spec'$.*

8.3 Step Correspondence

In order to prove that f is a simulation relation, we need to show that each execution of $RAID'$ has a corresponding execution in $Spec'$ having the same trace. For each transition of $RAID'$ we need to give the corresponding sequence of steps in $Spec'$. In this section, we give a corresponding sequence of steps for each transition.

Let s and u be reachable states of $RAID'$ and $Spec'$, respectively, such that $f(s,u) = true$, and $(s,\pi,s') \in trans(RAID')$.

- $\pi = Read(B)$
  Let the corresponding execution fragment be $\underline{u,Read(B),u'}$.
- $\pi = Write(P)$
  Let the corresponding execution fragment be $\underline{u, Write(P),u'}$.
- $\pi = read(i)$
  - If $s.DAG \in \{3,4,5\} \vee s.DAG = 2 \wedge i \notin s.Indices$.
    Let the corresponding execution fragment be <u>none</u>. In this case, the value read by π is not directly returned to *Env*. These reads are performed to compute parity or the value of a lost data being read.
  - If $s.DAG = 1 \vee (s.DAG = 2 \wedge i \in s.Indices)$, and $u.ReadPairs[i] \notin \mathcal{V}$.
    Let the corresponding execution fragment be $\underline{u,read(i),u'}$ In this case, the value read by π needs to be returned directly to *Env*, and $Spec'$ has not performed a read yet.
  - If $s.DAG = 1 \vee (s.DAG = 2 \wedge i \in s.Indices)$, and $u.ReadPairs[i] \in \mathcal{V}$.
    Let the corresponding execution fragment be <u>none</u>. In this case, the value read by π also needs to be returned directly to *Env*. However, $Spec'$ has already performed a read, meaning that $RAID'$ is running a second DAG for the current operation. Here idempotency of reads is used to show that s' and u correspond via the simulation relation.
- $\pi \in \{write3(i,v), write45(i,v), write6(i,v)\}$, where $i \neq n$
  - $u.WritePairs[i] \neq \perp$
    Let the corresponding execution fragment be $\underline{u,write(i,v),u'}$.
  - $u.WritePairs[i] = \perp$
    Let the corresponding execution fragment be <u>none</u>. In this case, $Spec'$ has already performed the write, meaning that $RAID'$ is running a second DAG for the current operation. Here idempotency of writes is used to show that s' and u correspond via the simulation relation.
- $\pi = ReadBack(P)$
  - If $s.DAG = 1$
    Let the corresponding execution fragment be $\underline{u,ReadBack(P),u'}$.
  - If $s.DAG = 2$
    Let the corresponding execution fragment be $\underline{u,read(s.f),u'',ReadBack(P),u'}$.
    In this case, there is a failure among disks to be read. In $RAID'$, reads of failed disks do not occur. But the value needs to be read in $Spec'$. We use the Consistency property presented in the next section, to argue that $RAID'$ returns the right value for the failed disk.

- π = *WriteOK*
  - If $s.DAG \in \{3,4,6\}$
    Let the corresponding execution fragment be $\underline{u, WriteOK, u'}$.
  - If $s.DAG = 5$
    Let the corresponding execution fragment be $\underline{u, write(s.f, v), u'', WriteOK, u'}$, where v = *WritePairsPerm*[*f*]. In this case, there is a failure among disks to be written. In *RAID'*, writes to failed disks do not occur. But the write needs to be performed in *Spec'*.

- $\pi \in \{fail(i),\ chooseDAG1 - chooseDAG6,\ write(n,v)\}$
  Let the corresponding execution fragment be none.

Recall from the introduction that, informally, the two conditions for correctness are consistency and idempotency. We indicate in what follows where these properties are used. Consistency is used to prove the step correspondence for *ReadBack*(P), in the case where there is a failure among disks to be read. Idempotency of read (write) low-level ops is used to prove the step correspondence of *read*(i) (*write*(i,v)), in the case where *RAID'* is running a second DAG to complete an operation and this low-level op has been performed once before.

8.4 Proof of Theorem 8.2

In the following proof, any state variable not preceded by a state name, is implicitly preceded by state s.

Proof. Assume $s \in start(RAID')$. We show that there exists a state u of *Spec'* such that $u \in start(Spec')$. In s, f = *none* and the variable *Bit* of *Spec'* has arbitrary values. Therefore there exists a state u of *Spec'* such that $f_1(s,u)$ is true. Secondly, $s.DAG = none$ and $u.pc = idle$ initially. Thus $f_2(s,u)$, $f_3(s,u)$, and $f_4(s,u)$ are also true. Finally, the *WritePairsPerm* variables variables are equal to P_0, and *ready* variables are *true*. Therefore $f_5(s,u)$ is true. Hence for any arbitrary initial start state s of *RAID'*, there exists a start state u of *Spec'* such that $f(s,u)$ is true.

Now assume that s is a reachable state of *RAID'*, u is a reachable state of *Spec'* such that $f(s,u) = true$, and $(s,\pi,s') \in trans(RAID')$. We show that for all action π, there exists an execution fragment α of *Spec'* starting at u and ending in u' such that $f(s',u') = true$ and $trace(\alpha) = trace(\pi)$.

Case: π = *Read*(B)
Let the corresponding execution fragment of *Spec'* be $u, Read(B), u'$. The two actions modify the variables *ReadPairs* in an identical way. Also *DAG* is set to *chooseR* in *RAID'* and *pc* to *read* in *Spec'*. Therefore, $f(s',u')$ is true.

Case: π = *Write*(P)
Let the corresponding execution fragment of *Spec'* be $u, Write(P), u'$. The two actions modify the variable *WritePairs* in an identical way. Also *DAG* is set to *chooseW* in *RAID'* and *pc* to *write* in *Spec'*. Therefore, $f(s',u')$ is true.

Case: π = *read*(i)
We know that $i \neq s.f$ and $s.DAG \in \{1,2,3,4,5\}$ by the precondition of π. We consider the following cases.

Subcase: $s.DAG \in \{3,4,5\} \vee s.DAG = 2 \wedge i \notin Indices$
Let the corresponding execution fragment be none. $f(s',u')$ is trivially true.

Subcase: $s.DAG = 1 \vee s.DAG = 2 \wedge i \in \mathit{Indices}$
By the inductive hypothesis ($f_2(s,u)$), either $s.\mathit{ReadPairs}[i] = u.\mathit{ReadPairs}[i]$, or $s.\mathit{ReadPairs}[i] \preceq u.\mathit{ReadPairs}[i]$. Since $s.\mathit{ReadPairs}[i] = \perp^+$, then $u.\mathit{ReadPairs}[i] \in \mathcal{V}^+$. We consider the following cases.

Subsubcase: $u.\mathit{ReadPairs}[i] = \perp^+$
Let the corresponding execution fragment of *Spec′* be u,*read*(i),u'. Since $i \neq s.f$, then by the inductive hypothesis ($f_1(s,u)$), $s.\mathit{Disk}[i] = u.\mathit{Bit}[i]$. Therefore the two actions change variables *ReadPairs* in an identical way, and $f(s',u')$ is true.

Subsubcase: $u.\mathit{ReadPairs}[i] \in \mathcal{V}$
Let the corresponding execution fragment of *Spec′* be none. By Invariant 1 of Lemma 6.2, $u.\mathit{ReadPairs}[i] = u.\mathit{Bit}[i]$. By the inductive hypothesis ($f_1(s,u)$), $u.\mathit{Bit}[i] = s.\mathit{Disk}[i]$. Also $s'.\mathit{ReadPairs}[i] = s.\mathit{Disk}[i]$. Therefore $s'.\mathit{ReadPairs}[i] = u'.\mathit{ReadPairs}[i]$ and $f(s',u')$ is true [2].

Case: $\pi \in \{\mathit{write3}(i,v), \mathit{write45}(i,v), \mathit{write6}(i,v)\}$, where $i \neq n$
By Lemma 6.1, $u.\mathit{WritePairs}[i] \neq \perp^+$. We also know that $i \neq s.f$. We consider the following cases.

Subcase: $u.\mathit{WritePairs}[i] \in \mathcal{V}$
Let the corresponding execution fragment be u,*write*(i,v),u'. Since $u.\mathit{WritePairs}[i] \in \mathcal{V}$, then by the inductive hypothesis ($f_3(s,u)$), $u.\mathit{WritePairs}[i] = s.\mathit{WritePairs}[i]$. Action *write*($i$,$v$) and π change variables *WritePairs* and, *Bit* and *Disk* in an identical way. Therefore $f(s',u')$ is true.

Subcase: $u.\mathit{WritePairs}[i] = \perp$
Let the corresponding execution fragment be none. Since $i \neq s.f$, then by the inductive hypothesis ($f_1(s,u)$), $s.\mathit{Disk}[i] = u.\mathit{Bit}[i]$. By the inductive hypothesis ($f_5(s,u)$), $u.\mathit{WritePairsPerm}[i] = s.\mathit{WritePairsPerm}[i]$. Also by Invariant 2 of Lemma 7.3, $s.\mathit{WritePairsPerm}[i] = s.\mathit{WritePairs}[i]$, which is in $\mathcal{V}$. By Invariant 2 of Lemma 6.2, $u.\mathit{Bit}[i] = u.\mathit{WritePairsPerm}[i]$. So $u.\mathit{Bit}[i] = s.\mathit{WritePairs}[i]$. Action π has the effect of writing $s.\mathit{WritePairs}[i]$ to $\mathit{Disk}[i]$. Therefore we have $u'.\mathit{Bit}[i] = s'.\mathit{Disk}[i]$. Thus $f(s',u')$ is true [3].

Case: $\pi = \mathit{ReadBack}(P)$
We know that $s.DAG \in \{1,2\}$. We consider the following cases.

Subcase: $s.DAG = 1$
Let the corresponding execution fragment be u,*ReadBack*(P),u'. We show that *ReadBack*(P) is enabled in u. We know that for all $i \in \mathit{Indices}$, $s.\mathit{ReadPairs}[i] \in \mathcal{V}$. Therefore, the inductive hypothesis ($f_2(s,u)$) implies that for all $i \in \mathit{Indices}$, $u.\mathit{ReadPairs}[i] \in \mathcal{V}$. Thus the action *ReadBack*(P) is enabled in u. The two actions change the state in an identical way. Therefore we have $f(s',u')$ is true.

Subcase: $s.DAG = 2$
Let the corresponding execution fragment be u,*read*(f),u'',*ReadBack*(P),u'. We first show that action *read*(f) is enabled in u. By the inductive hypothesis ($f_2(s,u)$), $u.\mathit{ReadPairs}[f] = s.\mathit{ReadPairs}[f]$.

[2] If read low-level ops were not idempotent, then this part of the proof would break.
[3] If write low-level ops were not idempotent, then this part of the proof would break.

By Invariant 2 of Lemma 7.2, $s.ReadPairs[f] = \perp^+$. Therefore $u.ReadPairs[f] = \perp^+$. Thus action $read(f)$ is enabled in u.

Next we show that action $ReadBack(P)$ is enabled in u''. We know that $\forall i \in All$, $s.ReadPairs[i] \in \mathcal{V}$. Therefore, by the inductive hypothesis ($f_2(s,u)$), for all $i \in Indices/f$, $u.ReadPairs[i] = s.ReadPairs[i]$. The action $read(f)$ has an effect such that $u''.ReadPairs[f] \in \mathcal{V}$. Therefore for all $i \in Indices$, $u''.ReadPairs[f] \in \mathcal{V}$.

In order to show that $ReadBack(P)$ is enabled in u'', it remains to show that $P[f] = \bigoplus s.ReadPairs$. We know that $P[f] = u''.ReadPairs[f]$, and that $u''.ReadPairs[f] = u.Bit[f]$. By the inductive hypothesis ($f_1(s,u)$), $u.Bit[f] = s.VD$.

By Invariant 1 of Lemma 7.2, $\forall i \in All$, $s.ReadPairs[i] = s.Disk[i]$. Thus $\bigoplus(s.ReadPairs) = \bigoplus_{i \in All} s.Disk[i]$. By Part 1 of Lemma 8.1 (Consistency), $\bigoplus_{i \in All} s.Disk[i] = s.VD$. Therefore $\bigoplus s.ReadPairs = u''.ReadPairs[f]$. Thus action $ReadBack(P)$ is enabled in u''. The two $ReadBack(P)$ actions affect the state in such a way that $f(s',u')$ is true.

Case: $\pi = WriteOK$
We consider the following cases.

Subcase: $s.DAG \in \{3,4,6\}$
Let the corresponding execution fragment of $Spec'$ be $u, WriteOK, u'$. We know that $s.WritePairs = P_{n0}$. By the inductive hypothesis ($f_3(s,u)$) and Invariant 2 of Lemma 6.1 $u.WritePairs = P_0$, and action $WriteOK$ is enabled in u.

The actions have a similar effect. Thus $f(s',u')$ is true.

Subcase: $s.DAG = 5$
Let the corresponding execution fragment of $Spec'$ be $u, write(f,v), u'', WriteOK, u'$, where $v = u.WritePairsPerm[f]$. We first show that $write(f,v)$ is enabled in u. By the inductive hypothesis ($f_3(s,u)$), $u.WritePairs[f] = s.WritePairs[f]$. By Invariant 3 of Lemma 7.3, $s.WritePairs[f] \in \mathcal{V}$. Therefore $u.WritePairs[f] \in \mathcal{V}$. By Invariant 3 of Lemma 6.2, $u.WritePairs[f] = u.WritePairsPerm[f]$. Therefore action $write(f,v)$ is enabled in u.

We now show that action $WriteOK$ is enabled in u''. Since $s.WritePairs = P_{n0}[,]$ then by the inductive hypothesis ($f_3(s,u)$), we have for all $i \neq f$, $u.WritePairs[i] = \perp$. Also action $write(f,v)$ sets $u.WritePairs[f]$ to $\perp$. Therefore $u''.WritePairs = P_0$. Thus action $WriteOK$ is enabled in u''.

Action π and $WriteOK$ have identical effects on the states of both automata. Therefore $f(s',u')$ is true.

Case: $\pi = fail(i)$
Let the corresponding execution fragment be none. Note that $s.DAG \notin \{none,2,5,6\}$. In the following, i denotes the index of the failed disk in s'.

Action π assigns the value of $s.Disk[i]$ to $s.VD$. By the inductive hypothesis ($f_1(s,u)$), $s.Disk[i] = u.Bit[i]$. Therefore $u'.Bit[i] = s'.VD$. Thus $f_1(s',u')$ is true. Action π either leaves DAG unchanged, or changes it in such a way that $f_4(s',u')$ is true. Also we have that $f_5(s',u')$ is trivially true. We show that $f_2(s',u')$ and $f_3(s',u')$ are also true. We consider the following cases.

Subcase: $s.DAG \in \{chooseR,1\}$
We have that $f_3(s',u')$ is trivially true. In this case, $s.WritePairs = P_{n0}$. If $s.ReadPairs[i] \neq \perp^+$ then $f_2(s',u')$ is trivially satisfied. Assume that $s.ReadPairs[i] = \perp^+$.

By Invariant 2 of Lemma 7.1, $s.Rec = false$. So by the inductive hypothesis ($f_2(s,u)$), for all $j \in s.Indices$, $s.ReadPairs[j] = u.ReadPairs[j]$. Action π has the effect of assigning value $\perp^+$ to all

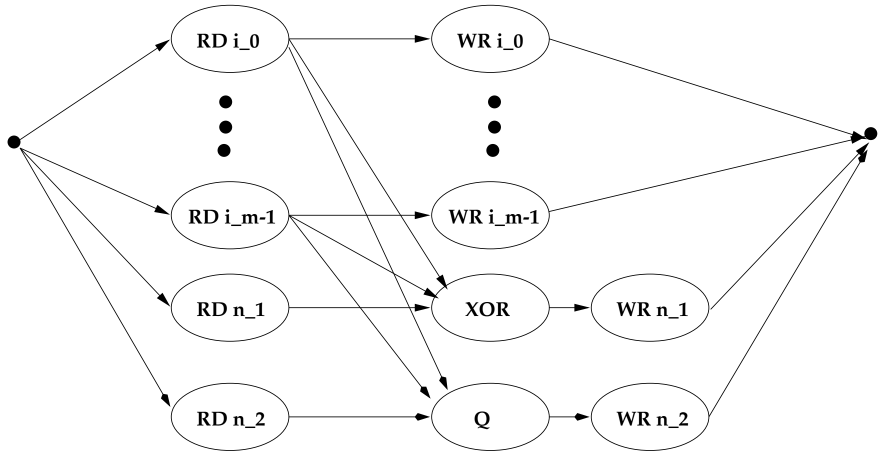

Figure 12: RAID Level 6 - Small Write.

$s.ReadPairs[j]$ for $j \in s.Indices$. Therefore for all $j \in s.Indices$, $s'.ReadPairs[j] \preceq u'.ReadPairs[j]$. Also if $i \in s.Indices$, $s'.ReadPairs[i] = u'.ReadPairs[i]$. Therefore $f_2(s',u')$ is true.

Subcase: $s.DAG \in \{chooseW, 3, 4\}$
In this case, $f_2(s',u')$ is trivially true. If $s.ReadPairs[i] \neq \perp^+ \wedge s.WritePairs[i] = \perp$, then $f_3(s',u')$ is also trivially true. Assume that $s.ReadPairs[i] = \perp^+ \vee s.WritePairs[i] \neq \perp$.

By Invariant 2 of Lemma 7.1, $s.Rec = false$. So by the inductive hypothesis ($f_3(s,u)$), for all $j \in s.Indices$, $s.WritePairs[j] = u.WritePairs[j]$. Action π has the effect of assigning $s.WritePairsPerm$ to $s.WritePairs$. By Invariant 2 of Lemma 7.3, if $s.WritePairs[j] \in \mathcal{V}$, then $s.WritePairs[j] = s.WritePairsPerm[j]$. Therefore for all $j \in s.Indices$, $u'.WritePairs[j] \preceq s'.WritePairs[j]$.

It remains to show that if $i \in s.Indices$ then $s'.WritePairs[i] = u'.WritePairs[i]$. Assume $i \in s.Indices$. If $s.ReadPairs[i] = \perp^+$, then $s.WritePairs[i] \in \mathcal{V}$, by Invariant 4 of Lemma 7.3. If $s.WritePairs[i] \neq \perp$, then $s.WritePairs[i]$ must be in $\mathcal{V}$, since by Invariant 1 of Lemma 7.3, it cannot be equal to $\perp^+$. Therefore action π does not change $s.WritePairs[i]$. Since we know that $s.WritePairs[i] = u.WritePairs[i]$, then we have $s'.WritePairs[i] = u'.WritePairs[i]$. Thus $f_3(s',u')$ is true. ■

9 Extension

We now turn our attention to a controller algorithm for the RAID Level 6 architecture [Gibson95]. We generalize part of the main invariant, Consistency, and use it to find an error in a RAID Level 6 DAG.

9.1 RAID Level 6

RAID Level 6 uses two parity blocks for each group of n blocks stored on separate disks. It can tolerate two disk failures. One parity block (n_1) is computed by taking the xor of all data blocks. The other parity (n_2) is computed using Reed-Solomon codes.

RAID Level 6 uses Courtright and Gibson's error recovery method analogously to RAID Level 5. It has DAGs that are similar. In particular the Small Write DAG is shown in Figure 12. The symbol Q indicates the computation of n_2.

9.2 Error found

We used a generalized version of part of the Consistency Lemma to find an error in the Small Write DAG, without performing the entire proof of correctness for RAID Level 6. Consider the following part of the Consistency Lemma (Part 1): In all reachable states of $RAID'$, if $n \neq f$ and $DAG = chooseW \wedge f \notin Indices$, then $\bigoplus_{i \in All} Disk[i] = VD$.

Consider the case in which there is a failure in a disk not to be written. Intuitively, this invariant says that if the controller is about to choose a write DAG, then the value implied by the system for the failed disk ($\bigoplus_{i \in All} Disk[i]$) is equal to the value of the last write to that disk (VD).

Consequently, for this invariant to be true, it must be that if a write DAG fails in such a way that there is a failure among the disks not to be written, then the value implied by the system for the failed disk is equal to the value last written to it. In other words, the failure of a write DAG should not cause the loss of data in disks not to be written.

We used this idea to find an error in the DAG presented above. Consider a scenario in which, $RD\ i_0$ and $WR\ i_0$ are performed, then disk indexed i_m (not to be written) fails. This does not cause the DAG to stop. But suppose disk i_1 then fails as well. In this case, the DAG stops and a new DAG needs to be chosen to complete the operation. However the value of disk i_m has been lost, because the array has been partially updated. In addition, when a DAG fails the state is reset and thus it is impossible to recover the value of the failed disk.

10 Conclusions

In this paper, we used I/O Automata to model and verify a controller algorithm for the RAID Level 5 system, which uses Courtright and Gibson's error recovery method. By performing this case study, we formalized a key invariant, consistency, which helped in finding an error in a different more complicated RAID controller algorithm.

This project started out by a preliminary study using the model checker SMV [McMillan92]. We modeled the DAGs for RAID Level 5 separately and used the tool to show that the DAGs preserve consistency. However it became clear that our notion of consistency was not accurate and that we needed to formalize this property. This led us to the study of the controller algorithm as a whole.

With the formalization of the consistency invariant we can envisage a tool that takes a model of a controller algorithm based on Courtright and Gibson's error recovery, and checks that consistency is preserved in all reachable states. Such a tool can be built based on a model checker.

The advantage of such a tool is that it would be specifically tailored to Courtright and Gibson's prototyping system. Its users will not need to know about the formalization of the Consistency property and will not need to reproduce the hand-proof present in this case study. However hand-proofs are essential at this stage of the design of the tool, because they allow us to determine the exact expression of properties to verify.

Courtright credits our work in his PhD thesis [Courtright97] as playing a role in debugging his designs and he encourages continued work in this direction, especially in collaboration with industry partners.

Future work consists of proving correctness of other RAID controllers using Courtright and Gibson's error recovery, as well as considering controller algorithms that use Courtright's latest error recovery method [Courtright97]. Finally, we plan to build a special-purpose verification tool.

References

[Bitton88] D. Bitton and J. Gray, "Disk Shadowing," *Proceedings of the 14th Conference on Very Large Data Bases*, 1988, pp. 331–338.

[Courtright94] W. V. Courtright II and G. A. Gibson. "Backward error recovery in redundant disk arrays." *Proceedings of the 20th International Conference for the Resource Management and Performance Evaluation of Enterprise Computing Systems (CMG).* December 4–9 1994, pp. 63–74.

[Courtright97] William V. Courtright II, "A Transactional Approach to Redundant Disk Array Implementation." Dept. of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA, Ph.D. thesis, April 1997.

[Gibson90] Garth Gibson. "Redundant Disk Arrays: Reliable, Parallel Secondary Storage". PhD thesis, University of California at Berkeley, 1990. Report UCB/CSD 91/613.

[Gibson93] G. A. Gibson and D. A. Patterson, "Designing disk arrays for high data reliability", *Journal of Parallel and Distributed Computing.* 17(1-2), 1993, 4-27.

[Gibson95] G. Gibson, W. Courtright II, M. Holland, and J. Zelenka, "RAIDframe: Rapid prototyping for disk arrays," Computer Science Technical Report CMU-CS-95-200, Carnegie Mellon University, 1995.

[Gray90] G. Gray, B. Horst, and M. Walker, "Parity Striping of Disc Arrays: Low-Cost Reliable Storage with Acceptable Throughput," *Proceedings of the Conference on Very Large Scale Data Bases,* 1990, pp. 148–160.

[Reddy89] A. L. Narasimha Reddy and Prithviraj Banerjee, "An evaluation of multiple-disk I/O systems." *IEEE Transactions on Computers*, Vol. 38, No. 12, December 1989, pp. 1680–1690.

[Kim86] M. Kim. "Synchronized Disk Interleaving". *IEEE Transactions on Computers* 35(11), November 1986, pp 978-988.

[Lawlor81] F. D. Lawlor. "Efficient Mass Storage Parity Recovery Mechanism", *IBM Technical Disclosure Bulletin* 24(2):986-987, July 1981.

[Lynch87] N. Lynch and M. Tuttle. "Hierarchical correctness proofs for distributed algorithms." Technical report MIT/LCS/TR-387, MIT Laboratory for Computer Science, Cambridge, MA, April 1987.

[Lynch89] N. Lynch and M. Tuttle. An Introduction to Input/Output Automata. *CWI-Quaterly*, 2(3): 219-246, September 1989. Centrum voor Wiskunde en Informatica, Amsterdam, The Netherlands.

[Lynch96] Nancy A. Lynch, "Distributed Algorithms", Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[McMillan92] K.L. McMillan, "Symbolic Model Checking: an Approach to the State Explosion Problem", Ph.D. Thesis, Carnegie Mellon University, 1992, CMU-CS-92-131.

[Patterson88] David A. Patterson, Garth A. Gibson, and Randy Katz. "A Case for Redundant Arrays of Inexpensive Disks (RAID)". *Proceedings SIGMOD International Conference on Data Management*, 1988, pp. 109-116.

[Patterson89] David A. Patterson, Peter Chen, Garth Gibson and Randy Katz. Introduction to Redundant Arrays of Inexpensive Disks (RAID). *Spring COMPCON'89* San Fransisco, CA, pp 112-17. IEEE, March 1989.

[Park86] Arvin Park and K. Balasubramanian. “Providing Fault Tolerance in Parallel Secondary Storage Systems”. Technical Report CS-TR-057-86. Department of Computer Science, Princeton University, November 1986.

[Salem86] K. Salem and H. Garcia-Molina. “Disk Striping”. *Proceedings of the 2nd International Conference on Data Engineering*, 1986, pp. 336–342.