Department of Computer Science

Formal Methods for Forensic Tools

By Thomas Porter

29th April 2019

Supervisor: Angus Marshall

Word count: 12143 (30 pages) for the main body of this report as counted by Microsoft Word

The main body of this report is followed by 15 pages of appendices that will be used to support the points made by this report.

Report on a project submitted for the degree of BEng Computer Science in the Department of Computer Science at the University of York

Table of Contents

Table of Figures

Executive summary

Aim of the reported work

The main aim of the reported work that will be carried out in this project is to test if it is possible to create a formal specification for a tool commonly used in forensic computing investigations. Additionally, this project will evaluate the usefulness of formal methods to model tools within this field. This project will mainly focus on creating a formal specification for a disk imager as this type of tool is used in most investigations to preserve the original data that is being used as evidence. It will also create another specification that can be used to map a file system onto the created image and read files from that image. This will allow for the possibility to fully model the process of extracting a file from a disk using formal methods.

Motivation

The main driving motivation for conducting this project is that tools that are used during forensic investigations have not been formally modelled. This makes evaluating the accuracy of the results that they produce considerably harder. Currently, forensic tools are trusted based on past results that are from trusted sources. This can make reproducing those results quite difficult and can cause complications during investigations where the evidence gained through these tools needs to be presented. As mentioned before, a disk imager will be formally modelled through this project because it is one of the simplest and commonly used tools during forensic investigations. The choice to model a file system on top of this disk imager specification was made because in order to extract useful information, such as a given file. A file system needs to be mapped on top of that image so that the correct data can be extracted for the required file.

Methods

Before a formal specification can be created, the requirements for that tool needs to be captured. Some work has been done in this area by NIST [1] and SWGDE [2]. Therefore, this project will build upon the requirements that have been established by these two groups. Once these requirements have been decided upon it will then a formal specification can be created in Z that shows how these requirements should be implemented for the disk imager. When creating the Z specification for a file system, once again, the requirements and implementation for a file system need to know to be known. However, unlike a disk imager, the file system that has been chosen to model (FAT) has a very well documented implementation [3]. This specification produced by Microsoft will, therefore, form the basis of

the Z specification that will be produced during this project. As the aim of modelling a file system is to map it onto the images that are produced by the other Z specification, the representation of the disks that both specifications use should be the same. The file system functions that will be in this specification are: add, remove and read. These functions will allow for a basic file system to operate.

Results found

This project has found that it is possible to specify a disk imager that operates at the bit level and therefore it is very useful to the evaluation process of tools such as this. However, specifying a file system that uses the same representation of data was not possible to do in Z given the time constraints of this project. Therefore, a more abstract version of a FAT file system was created so that it can still model the functionality of the file system. In future, this work can form the basis of a Z specification that will be able to map onto the disk imager specification. This future file system specification would need to use a different representation of data and therefore so would the disk imager. This would allow for file operations to be carried out on the disk at a bit level which is crucial for investigations.

Statement of ethics

Part of the research for this project involves looking at confidential information and working documents used by law enforcement. These documents will be treated in confidence when they are used with the consent of the document's owner. This project does not gather data of individual people that participate in this project. Therefore, there does not need to be any consideration for properly handling personal data.

Acknowledgements

I would like to thank Angus Marshall and Jeremy Jacob for their help and guidance through this project.

Chapter 1 – Introduction

1.1 – Introduction to Forensic Tools

There are many tools that are regularly used during forensic computing investigations to aid the investigators who are working with the evidence involved. An example of a forensic tool that is used very frequently is a disk imager. One of the main principles of investigating is to not change any of the original data that is used as evidence, so it can be kept as a reference. Therefore, a disk imager is used to create an exact copy of the original disk that can then be used by investigators to extract data or files for their investigation. It can do this by either creating a copy of the drive onto another empty drive or by creating a new file that contains all the information that is stored on the source drive. One problem with these tools is that there is not a formal definition of what the tool should technically do. That is, how the inputs to the tool are processed to produce an output. Instead, the accuracy of these tools is established by testing them on known data sets.

1.2 – Introduction to Formal Methods

The method that will be focused on in this project is using formal propositional logic to model the real world mathematically. This process tends to start with requirements and then a model is created that unambiguously models them. This model will have operations that are used to show how an implementation of the requirements should function. This specification can then be used to create an implementation where it is very clear how it should function and the results it should produce. There are several languages that can be used to write this logic in, including Z and circus. Each has its own way of representing mathematical principles. This project will attempt to capture the requirements for a tool used in forensic investigations (disk imager) and then create a schema in Z for them.

1.3 - Aims of this project

The aims of this project are to:

1. Evaluate the requirements that have already been captured for a disk imager
2. Capture any additional requirements for a disk imager
3. Create a formal specification for the disk imager
4. Create a Z schema that can map a file system onto an image
5. Make a file system specification with basic file operations
6. Evaluate the usefulness of using forensic methods in creating and testing tools

Chapter 2 - Literature Review

2.1 – Extracting Evidence from Digital Devices

Before any analysis can be carried out on any data during an investigation. The data must be extracted from the device that it is currently stored on. The ACPO Guide for Digital Evidence has some principles for the handling of digital evidence [4, p. 6], where it explains that anyone involved in the investigation should not change any data that may later be relied upon in court. If it is necessary to change any data, then it should be documented that this has happened. However, it is possible that changes to the source data may happen during the data extraction from the device it is held on. Using these principles, devices can be split up into three categories.

I. The first type of device is the best case for extracting data and is referred to as a dead box. This is where there is no change in the data that is held on that drive. This can happen when a hard drive is taken out of a computer and put into another. This may need a write blocker to ensure that nothing has been changed when the drive has been added to the new computer. This type of device describes the case in principle 1.

II. The second type of device is where some of the data must be changed during the extraction process. For example, to extract data from a mobile phone some software may need to be put onto it which, in turn, will overwrite some data stored on it. Also, the phone may need to be mounted to grant access to the investigator to get to the files they need. There will need to be a record of the changes made to this device for this process to abide by the principles of handling digital evidence.

III. The final type of device is referred to as remote devices. This is the worst case when trying to extract the data from a device because the data is being provided by another source and their word must be taken for its completeness or correctness. For example, data used by a website may be presented differently depending on which browser is being used or the history of that browser. Cloud storage also fits into this category as the real file system that is underneath the cloud service will not be known to the end user. Because of this, when trying to extract data from a remote source the person doing this needs to be very competent with data extraction.

2.2 - Requirements in forensic methods and tools

Digital investigations have been a part of the forensic field for as long as more “traditional” sciences such as DNA, however some methods used are fairly new when compared to other methods used in these sciences. For example, some methods of data extraction need to be formally validated. That is to ensure that they implement their requirements correctly and accurately. There are some problems with the methods that they use. The article: Standards, regulation & quality in digital investigations: The state we are in [5] discusses these problems with producing standards for forensic methods within forensic computing. Sometimes evidence gained through these new methods can be dismissed due to corners cut in the rapid adoption of these methods. Proper review and validation can be bypassed, so even though the evidence gathered is good, the method has not been proven. When standards need to be produced, there is not a clear way to show that the requirements have been satisfied. This article suggests that with traditional sciences, forensic computing needs a way of clearly verifying that the method is accurate and correct all the time. In traditional forensic science, this includes staff competence, validation of the process and proficiency of the provider. There have been standards to try and bring these into forensic computing including ISO/IEC 27001:2013 [6] which is part of the ISO/IEC 27xxx family of standards. These standards require evidence that the process satisfies the requirements for this investigation. Formally specifying the requirements of the process of disk imaging will help to provide this evidence.

The article: Requirements in digital forensics method definition: Observations from a UK study [7], it explores the need to create requirements for forensic methods. It finds that requirements of forensic tools tend to be quite non-technical in nature as they sometimes refer to a method as well as the tool itself. In the cases of some tools that have been commercially created to be sold, the requirements are not publicly available. This is due to the publishers wanting to protect their methods of capturing them or what they are. This can make validation of the tool a lot more difficult as the requirements are not clear. Therefore, there are not clear test criteria that can be used by an impartial third party to evaluate the finished tool. This article concludes that all labs should have the same core technical requirements that are established by a group with the technical knowledge within that discipline. This ensures that the requirements will accurately capture all the functionality that the tool needs to have. These technical requirements can now go on to form a specification for the tools.

As mentioned before, there is a great need for requirements to be created and standardised. Due to this, the National Institute of Standards and Technology (NIST) has produced a set of requirements for disk imagers and cloners [1]. This document includes mandatory requirements as well as optional requirements for imagers that can be used in forensic investigations. It also sets standard terminology to be used when discussing imagers in this way. The purpose of this document was to set a standard for what disk imagers should do given the large variety of ways they can be implemented and the different types of devices for storing data. While specifying definitions, NIST has set need to use bit streaming in when creating a copy or an image of a source disk. This is important because it increases accuracy by making a copy of each bit individually on the drive. Also included within this document are requirements for general functionality of disk imaging programs, including the format of the images that are created and the ability to create and delete those images. Although this is important, it is not essential for specifying the functionality of these programs. There are two major points of the mandatory requirements in the NIST report. They are that all visible and hidden data on the source drive is to be included in the image. This is important because it explicitly states that all data, regardless of whether it is visible to the user or not, will be included in the image. This removes the chance that some potential evidence that has been hidden on the source disk could be missed. Also, a disk imager must be able to handle errors such as a corrupted sector within the source disk. Once this error has been detected it needs to report its location to the user and fill the area with benign data (such as all 0s). I have noticed that one crucial area that is not mentioned within the mandatory requirements is that the source disk should remain unchanged throughout this process, although it is mentioned in the optional requirements. I believe that this should be in the mandatory requirements as this will need to be explicitly stated within the formal specification that will be created. The results of this document will form the basis of this project for when the requirements are formally specified.

2.3 - Requirements for testing forensic tools

Further work has been done by the Scientific Working Group on Digital Evidence (SWGDE) into capturing requirements for tools used during digital investigations. As part of this work, they have captured testing requirements for various forensic tools, including disk imagers [2, p. 8]. This document recommends minimum testing requirements for disk imaging tools to be used within the forensic field. The SWGDE states that throughout all the testing process a known dataset should be used and that disk imagers must use write blockers to preserve what is currently on the source disk. The requirements must also include verification that the source disk was unchanged. However, if the write blocker is separate to the disk imaging tool then this verification does not need to be included in the requirements. Throughout the development of such a tool the SWGDE recommends that if any changes are made to the tool, regardless of size, it must undergo retesting again. There are also requirements of the tester where they may need to carry out performance checks on hardware devices in accordance with the organisation's policy.

In practice, the requirements produced by the SWGDE seem sensible enough to be used until they are proven wrong. If this happens, the requirements should be updated to reflect this change. An independent group such as this would be best suited to make these changes as they will be more likely to able to do the impartial research and make judgements for the requirements. They are also more likely to be transparent about their capture process in aspects where a private company would not.

2.4 - Using Domain Specific Languages in Forensic Investigations

Within forensic computing, there are several problems when analysing data, including a large amount of data and variety of data types and formats. A way of dealing with this is suggested by Jeroen van den Bos and Tijs van der Storm [8]. They suggested using domain-specific languages within digital forensics. Domain-specific languages (DSLs) are languages that cannot be used outside of their predefined domain (their intended purpose). The highlighted problem that DSLs attempt to solve is the vast amounts of data and a large variety of types and formats of this data. The language DERRIC is used as an example within this paper to illustrate how the separation of stages within the forensic process can make an analyst's job easier. It describes data formats that can then be analysed by other tools. This allows for separation between the extraction of files and their analysis. This separation means that the analyst does not need to know the exact formats of the data they are working on and how it is stored. They then only need worry about what they have to do with the files that they are given rather than having to extract them and deal with the complications involved.

This allows for the problems mentioned before to be completely bypassed by obstructing between the stages in the forensic process. This concept is then built on further by the forensic query language Nugget that will be touched on later.

2.4.1 – Nugget: A DSL Example

Following on with DSLs, a forensic query language called Nugget has been created and is explained by Christopher Stelly and Vasil Roussev [9]. Nugget aims to establish a standard query interface for forensic investigations. It aims to do this in a similar way to SQL and relational databases with the intention of being able to seamlessly expand upon the language in the future. The most important concept with this DSL is that intuition is important. i.e. a forensic analyst that did not originally write the query will be able to read and understand what it is and what it is doing. The user will not need to know how this is implemented in order to write a query. A language such as Nugget puts the power in the hands of investigators as they can directly specify queries that they need for their investigations. This allows the investigators to work quicker and to spend more time on the analysis of data rather than its extraction. Each query written in Nugget can be easily compared with other queries written by other investigators or teams. This is an important characteristic as it allows more traditional scientific methods to be applied as the query itself will be more reproducible. It also requires a little less documentation or explanation to go along with the query as it is, to a degree, self-documenting. Nugget aims to be a step towards formalising the languages within forensic computing investigations by standardising the way queries are carried out.

Forensic languages such as Nugget operate at a very high level and assumes that the implementation is correct and that it always works. Currently, this assumption is based on trusted past results that have been carried out by skilled analysts and investigators. Many of these tools have not been formally specified or proven and therefore, the aim of this project will be to specify one of these tools.

2.5 - File Systems

All drives that are used by computers will need to have some form of file system mapped onto them so that the computer can make sense of the data stored on that drive. File systems are an abstract concept used to make sense of the data on the hard drive and assign sections of that hard drive to files. These systems also provide common tasks that facilitate the management of files, this includes: adding, deleting and reading files. Within the field of forensic computing, a file system needs to be mapped onto a drive or a created image in order to extract files from the drive. It is at this point that languages such as Nugget may be used. The same drive can have multiple partitions on it and each partition can have different file systems in them. Some file systems then split themselves up into different sections. Within this report these sections will be called segments.

An example of a file system is the File Allocation Table (FAT) file system. It is used to easily provide a set of logical segments that can be used to store files in. FAT-based file systems were commonly used by Microsoft in their operating systems such as MS-DOS and Windows 95. It has since been replaced by NTFS [10] and other file systems in modern operating systems (OS). However, this project will still be focusing on FAT systems as they used within forensic computing and is still important as it is still commonly used by solid-state flash memory devices such as USB drives. The official specification by Microsoft can be found online [3] and will be used along with other sources to form an understanding of this. A good explanation of FAT-based file systems can be found in Forensic Computing a Practitioner’s Guide [11, pp. 172-207]. Along with a good explanation of file systems, this book also provides a greater context for how FAT-based file systems are relevant within the forensic computing community.

The main aspect of FAT file systems is that drives are split into four segments, these are reserve sectors (boot sector), file allocation table, root directory and data. See Figure 1 for a potential layout of the segments. The reserved sectors segment is always at the beginning of the file system and contains information about the OS and the drive itself. Specifically, it also contains vital information needed for running the drive, including the segments table and any other information needed to boot the system. For the purposes of this project, the most important part of this segment is the segment table as that will tell me where the other partitions started and end. Due to the nature of the boot segment, each byte has a fixed position within partition that the file system is in. For example, the partition table always starts at offsets address 1beh in the boot record. The data segment contains the data itself where the data is split up into clusters. The size of these clusters is dependent on which version of FAT is in use. The data stored in this segment does not need to be in the correct logical or physical order as the ordering is provided by the FAT segment. The file allocation table contains the order and location of each cluster needed to make up each file stored in the file system. This table functionally works like a linked list data structure when looking at how the pointers interact with each other. There can be multiple file allocation tables depending on the FAT version or the number of partitions set up on the disk. In the case of Figure 1 there are two file allocation tables starting at offset 1 and 253. The final partition is the root directory which contains all the information about the file (name, date created, etc.). More importantly, it also contains the first cluster location.

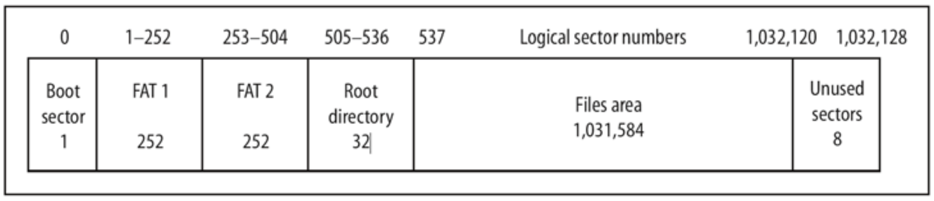


Figure 1 Layout of the segments in a FAT partition from: [11, p. 189]

When reading a file from a FAT file system first look the file up in the root directory to get the starting cluster pointer. Then, by using the FAT one can look up the corresponding entry to the starting cluster. See Figure 2. This entry will have two values – the pointer for the next cluster and a byte which is used to flag the status of the cluster. For example, a flag with the value zero means that the cluster is free for use and the value of one means the cluster is currently being used. If it is being used, then the next cluster in FAT can be read to get the next cluster address for the file. The flag has been omitted from Figure 2 to clearly show the relationship between the pointers.

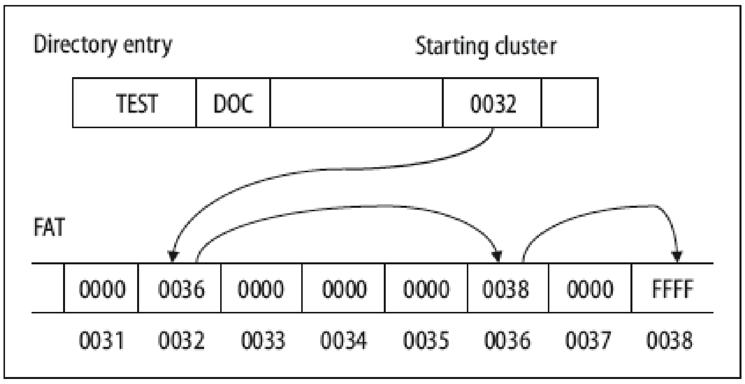


Figure 2 FAT pointer relationships from: [11, p. 190]

Deleting files from a FAT file system is very simple to do as the data itself remains unchanged; it is only marked as deleted and therefore this space can be overwritten the next time a file is saved there. To delete a file its name in the root directory has the first character overwritten with the deleted character: e5h. It should be noted that the rest of the directory entry remains intact. It then needs to be removed from the file allocation table, this is done by setting the value of the flag in each entry that is used by the file 0. As a result of this method of removing files from the disk, from a forensic viewpoint it makes data recovery very easy because, so long as the data has not been overwritten, it will remain on the drive.

2.6 - Specifying Tools

As mentioned when discussing how forensic tools are used in practice, there is a great need for formally specifying forensic tools so that their correctness and validity can be mathematically proven. There are several problems with formally specifying these tools, including the requirements not being publicly available and the companies who make these tools not usually being open about how they were captured. Another problem is that the exact implementation of some tools is not known as they are guarded by the companies who make them

Techniques for formally specifying problems are outlined in the paper by Anthony Hall [12] where he describes how certain specification techniques can help in industry. There is a great need to understand the problem and environment that is being specified. One of the better ways to do this is to capture requirements which, as mentioned before, is very difficult to do for forensic tools. The benefits provided by these techniques allow for well-established refinement by moving from an abstract design of a system to a concrete implementation (where the system design allows). For each step within the refinement, it is possible to prove that the functionality remains the same from the original specification. Once implementation has been finished, having a formal specification already written makes testing this implementation easier as the exact function has been explicitly stated with no ambiguity.

2.6.1 – Using Z as a Specification Language

In his paper, Hall suggests using Z to formally specify projects [12]. This project will be using Z as it allows schemas to be written for each state and operation that the tool needs in order to function. Each of these schemas can then be refined and tested in isolation with an easy mapping between Z code and the original requirements. This refinement of schemas allows for specifications produced in Z to have their correctness mathematically proven. The Z also facilitates natural iteration towards a concrete implementation of a design. It does this by creating another schema which is more accurate to how a programming language would be and then schema calculus can be carried out upon both sets of schemas to prove that the concrete implementation applies the abstract design. There is also a well-supported plug-in, CZT [13], for eclipse that aids the development of the Z. Further information about Z as a specification language can be found at [14]. Another feature of Z is that is supports variables that can be used for inputs or outputs to the operations that they are in. The convention in Z is that input variables have a "?" at the end of their name and output variables have a "!" at the end of theirs. These two types of variables will be very useful when writing a schema for operations used in a forensic tool.

2.7 – Literature Summary and Project Context

Data that is collected during forensic investigations needs to be preserved so it can be used later in court. This is done by using tools such as a disk imager to create a copy (image) of the disk and then the analysis can be carried out on this image. A file system will then need to be mapped onto this image to extract files from it. A problem with this process is that the tools used have not been formally specified so the process of mathematically proving their correctness and accuracy cannot be started. Once this has been done it will be necessary to show that the data remains unchanged during the process. This project aims to achieve the first part of this process by creating a formal specification for forensic tool. It will then model a file system that can be mapped on the new image. The method for doing this is discussed in the next chapter.

Chapter 3 - Method

3.1 - Capturing Requirements for a Disk Imager

Before starting any formal specification, the first thing that needs to be done is to accurately capture the requirements of a disk imager and the functionality of a FAT file system. The requirements for the disk imager will be primarily based upon the work done by NIST [1] and SWGDE [2, p. 8]. These two groups have laid the groundwork in terms of capturing requirements for disk imagers used in forensic computing investigations. The formal specification to be created will, where possible, preserve the functionality stated in the requirements.

3.2 - Modelling a Disk Imager

In order to create a model of a disk imager, Z (a formal specification language) shall be used for this project to accurately describe the imager. Throughout writing this specification the Z user manual [15] will be used as a syntax reference. One of the reasons that Z will be used throughout this project is that there exists some IDEs. Therefore, during development, Community Z Tools (CZT) [13] which is an extension for Eclipse, will be used for this project.

As is the convention in Z, there will be schemas for the base state, initialization, create an image, and search for bit pattern. Each one of these schemas will specify key requirements of the disk imager. When dealing with images or the contents of hard drives the specification will deal with them in terms of sectors of the disk. Sectors will be the smallest addressable parts of each disk and will, therefore, simplify the specification as it is only sectors that we are dealing with which are sequences of bits. The length of each sector will be able to be variable in length in order to allow the specification to fit different disk and file standards. A useful feature of Z that will allow for sectors to be the main representation is sequences. This will inherently preserve the order of the sectors.

As mentioned above (see 2.1 – Extracting Evidence from Digital Devices), there are problems and appropriate precautions that need to be taken when extracting data from different types of devices. Therefore, when modelling a disk imager, the focus of this specification will be on imaging the devices that are in the first type. That is that their contents are unchanged during the extraction process and a complete image can be produced. The specification will still allow the second type of device to be represented. It will do this by keeping track of which sectors have been changed on each device. This information can then be provided to the user when needed. Reporting the changed sectors in this way fulfils the second principal from ACPO Good Practice Guide for Digital Evidence [4, p. 6]. Once the changed sectors have been recorded the specification can then treat the devices as if they were part of the dead box type. It should be noted that the third type of device (remote) will not be modelled by the disk imager as this would unnecessarily overcomplicate the specification as it will not be known which sectors on the disk are correct.

One important requirement of a disk imager in the NIST investigation was the concept of a disk imager to be able to handle errors. They suggested that unresolved errors from reading the source would be written as benign data. Therefore, any sectors which have been corrupted will be written into the image as a sequence of zeros. The system should then report this back to the user. This will be done by outputting a set of sector addresses where there has been an error. If there are no errors this set will be equal to the empty set.

Searching for a bit pattern within a created image will be quite useful for the specification to model as that is a common operation carried out upon images after they have been created. There are two scenarios that a search for a bit pattern would be carried out in and, if possible, the specification should be able to model both. The first one is where the bit pattern is contained within one sector. In this case, the location of the sector that the bit pattern is in is then reported back to the user. The other and more complex scenario is where the bit pattern is straddling multiple sectors. For both cases, the query should report the bit offset for the sector where the bit pattern starts. If the bit pattern occurs multiple times in the same image the query should return all the locations of this pattern. However, in this specification, it will non-deterministically choose one of them and return this. This is discussed in Chapter 4 - Results - later. This can be changed by the implementer to, for example, return the location of the first bit pattern and the Z specification will remain true

3.3 - Modelling a File System

When modelling the FAT file system to use with the disk imager, there will have to be a separate Z specification that will include all the functionality of the file system. This will be to make it easier to read and use the specifications as they will be separated out by function. Also, the idea of a file system is an abstract concept that is then mapped on top of some existing data. In this way, it will represent the real world better if the two specifications operated separately and allowed data to be used between them. It will also have to represent the data from the disk in a different way to represent the partitions that exist in the file system. Because of this new representation, there will need to be specific functions that allow the user to convert between the disk imager and the file system. This allows for a file system to be mapped on top of the created images and read the files that they contain. This is very important with forensic investigations as it allows files to be extracted and analysed from target hard drives without changing the original data, so nothing is lost that may be later relied on in court. This is explicitly stated in principle 1 in [4, p. 6].

The FAT specification will have to include some key operations on top of the ones that convert between the file system and an image. These will be deleting, adding a file and reading a file. As per the requirements of a FAT file system when deleting a file, the data itself remains unchanged and it is the pointers to this data that are changed. Therefore, the delete operation must not affect the data partition and will need to both change the file name and update the flags in the file allocation table to show that the file has been deleted and the space it is currently taking up is free to use. When reading a file, the specification will use a query in Z to produce a sequence of bits that make up the file with the given filename. An inherent property of queries in Z ensures that they do not change the data that is being used as they do not have access to the post-state of an operation. They can only access the pre-state and therefore any changes will not be carried over to the post-state. This property will be used to ensure that reading a file does not change any of the data in the representation of a disk. Finally, when adding a file, the operation will need to find the correct number of free sectors, put the data in the sectors, and update the pointers in the relevant partitions.

One aspect of the FAT file system specification is that it should work alongside the specification written for the disk imager. This involves the operations written for the file system to be interacting with the data on the bit level just as it is with the disk imager. An example of how the functions will handle this interaction is that all the delete function will have to do is change the sequence of bits that make up the first character of the file name to a new specific sequence of bits which is the delete character as set in the official FAT specification.

Chapter 4 - Results

4.1 - Modelling a Disk Imager

4.1.1 - Capturing Requirements for a Disk Imager

As previously mentioned, capturing requirements for published tools can be very difficult so I looked at the requirements gathered by NIST and SWGDE. I then selected the ones that are most important to creating a formal specification. These two documents include requirements for the file types of the images that are created. As the purpose of creating requirements for this project is to use them to create a formal specification, it is not necessary to include details about the file type that will be used to store the image in. The requirements I captured can be seen in Appendix 1.1: Requirements for a Disk Imager. Overall, I agreed with the mandatory requirements presented by NIST with one exception: this was the optional feature “Acquire an unprotected digital source without modification of the source” [1, p. 8]. This requirement is optional for disk imagers because it is handled with a piece of hardware called a write blocker. I believe that this requirement should be a mandatory feature of a disk imager specification, especially within the context of forensic investigations. It can then be implemented by tested hardware when an implementation is created from the specification in the future. Therefore, I have included that in the list of requirements that I have used to create a formal specification (requirement number 2 in Appendix 1.1: Requirements for a Disk Imager). Within the NIST requirements it mentioned that all visible and hidden data need to be present in the created image of the disk. Therefore, when I went on to model a disk imager, I have assumed that it will be dealing with raw data (bit streaming) so the notion of a hidden file does not affect this representation.

4.1.2 - Creating a Specification for a Disk Imager

The finished specification can be found in Appendix 1.1 and it has the following functions:

- Initialization
- Create Image
- Find a Bit Pattern

For this specification, I chose to represent a sector as a sequence of bits and subsequently a drive as a sequence of sectors (where a bit is a defined type in Z with two values). Therefore, when creating an image each sector is written into its corresponding position in the new image, one sector at a time. In order to fulfil the requirements, there is an additional condition within this operation that ensures that the bits within each sector are the same between the source drive and the new image. This is except for corrupted sectors which will be discussed later. This fulfils requirements number two and three in Appendix 1.1: Requirements for a Disk Imager.

One key aspect of the requirements that I captured is that of error handling. The specification deals with this by creating a new sector that contains all zeros and inserting it into the place of the corrupted sector in the image. This happens when it is noted that the sector address is defined to be corrupted. It then reports this sector address back to the user so that they can be aware that there may be data missing if they use that sector in the future. It should be noted that this specification does not have a formal way of determining whether a sector is corrupted or not. Rather, it has a set of all the addresses that are corrupted for each drive that it has in the system currently provided to it. If an implementation was to be created based on this specification, then this set of corrupted addresses will need to be generated in a different way before the system can run.

As mentioned above, it is possible that during the data extraction some sectors of a source disk may be changed by the extraction. This may be necessary before the image creation can be done. If this happens then a record of this must be kept ensuring that principal 2 of handling digital evidence [4] is met. This is handled by the disk imager specification by keeping track of a set of sector addresses that have been changed per device. There are conditions that determine the nature of the classifications of each of these types. When an image of the device is being created it does not consider that some sectors within that drive may have been changed. This is because, although it is important to know which sectors have been affected in terms of the overall investigation it is not that relevant to creating an image of the disk. The create image operation is designed to make an exact copy of the device it is given. Therefore, I believe it is enough to provide a set of addresses that have been changed along with the created image. This is enough information for an investigator to analyze the data within the image.

As alluded to in the requirements there should be a time restraint upon the running of each of the operations within a disk imager. This requirement will need to be kept in mind for any implementation that is created based upon this specification.

4.1.2.1 – Searching an Image for a Bit Pattern

Once an image has been created it will be very useful to be able to search that image for a specific bit pattern. However, when creating this operation, I ran into a limitation of Z, this being that when searching for a bit pattern that straddles multiple sectors it could not determine the location within the image, rather whether it was there or not. This was because this operation created a new sequence of bits from all the sectors in the image and then used the built-in function in Z (“in”) to evaluate whether a sub sequence (bit pattern) was in the larger sequence that represented the image. See implementation1 from Figure 3 for how this was implemented in Z. It should be noted that CZT uses “infix” to mean “in”. The second, simpler, implementation in Figure 3 is searching for a bit pattern that is contained within a single sector. The operation that this Z extract is from does contain an additional condition to make sure that the bit pattern is shorter than the length of a sector. A solution such as implementation 1 will not be ideal for forensic investigations. However, this could be a feature that could be included in future iterations of the specification that use either a different language or a different representation of the data.

```
∀sectorAddress: dom (drives drive?) • ∃image: seq( BIT ) | ( ( bitPattern? infix
image ⇒ patternFound! = TRUE  ) ∨  (¬(bitPattern? infix image) ⇒
patternFound! = FALSE) ) • image = image ⁀ drives drive? sectorAddress
```

Implementation 1 – The above Z code checks if the bit pattern (bitPattern?) is in the sequence if bits in the image. It will then report true or false depending if the pattern is in the sequence or not. In this case the image is all the sectors concatenated together.

```
∀sectorAddress: dom (drives drive?) • bitPattern? infix (drives drive?
sectorAddress) ⇒ patternSectorNums! = patternSectorNums! ∪
{ sectorAddress }
```

Implementation 2 - The above Z code outputs a set of sector addresses that contain the bit pattern held in bitPattern?

Figure 3 Different implementations of searching for a bit pattern in a created image

4.2 - Modelling a File System

Just as with the disk imager, the first stage of creating a specification for a FAT file system was to capture the requirements and functionality that such a system should have. Unlike the disk, imager FAT is very well documented and therefore extracting its key functionalities and how it is implemented was a lot easier to do. It was decided that the specification would model the partitions and some set operations that can be carried out on files. These operations are adding, deleting and reading files. This basic functionality will allow the Z specification to model the basic operation of a FAT file system.

When modelling the FAT file system, I chose to attempt to make the specification fit any of the versions. This was done by allowing the size and number of clusters present to be variable or set beforehand. This allowed for a more general specification to be made that I feel can be applied to different versions and be used within different applications. The general structure that I chose to follow for this Z specification was to have one of each of the partitions that are used for a FAT file system. These are:

- Boot sector
- File allocation table
- Root directory
- Data

Initially, the intention was to create a file system that could be mapped on top of the images that are created by the imager specification. The full version of this specification can be found in Appendix 1.3. This design created several problems when it came to writing operations that are essential for a file system (such as adding and deleting files). The main problem experienced was that, due to each partition in the file system being a sequence of bits, there was not a simple enough solution to these operations and was limited by Z with this representation. For example, it was possible to remove a file from the root directory, however it was not possible to remove a file from the file allocation table due to the relationship of entries within the table not being obvious in a sequence of bits. At this point, I could have simplified the representation by dealing with individual sectors that were on each disk, the size of which would be determined by the version of FAT that was being modelled. However, as key functionality would not have been possible in this design, I redid this specification but at a more abstract level that has preserved key concepts and functionalities of a filesystem. This specification can be found in Appendix 1.4. At this point, it should be noted that using this abstract representation of a disk will mean that this specification is now separate to the disk imager and it is not possible to convert between either of them. This is a good trade-off as it allows for a filesystem to be formally specified in this way. From now on within this report, the original implementation of a filesystem that represents the data as a sequence of bits (Appendix 1.3) will be referred to as version 1 and the more abstract representation of the file system (Appendix 1.4) will be version 2.

An example of how a file is deleted from the root directory partition can be seen in Figure 4 where representation 1 is taken from the specification that represents of the partitions as a sequence of bits and representation 2 is from the abstract representation of the partitions. In this example, error reporting has been admitted for conciseness. Both are taken from the delete function that is given the file name (name?) and with only this information located it and remove it from the root directory. Representation 1 is truer to the way files are deleted from the root directory in implemented FAT file structures. This is done by changing the first character of the file name to a set value whereas representation 2 simply removes the relation from the file name to the root directory entry in the partition.

$$\exists \mathit{char} : \mathrm{seq}(\ \mathrm{BIT}\ ) \bullet (\mathit{char} = \mathrm{head}\ \mathit{name?} \land \mathit{name?}\ \mathrm{infix}\ (\mathit{drives}\ \mathit{device?}\ 3)) \Rightarrow \\ (\ \exists \mathit{fileNameStart} : \mathbb{N} \bullet (\mathit{drives}\ \mathit{device?}\ 3)\ \mathit{fileNameStart} = \mathit{char} \land (\mathit{drives}'\ \\ \mathit{device?}\ 3) = (\mathit{drives}\ \mathit{device?}\ 3) \oplus \{\ \mathit{fileNameStart} \mapsto \mathit{DeleteChar}\ \}\ )$$

Representation 1 – The section of Z code shown above checks if the file name (name?) is in the rood directory (drives device? 3). If it is then it overwrites the character position that is the first character in the file name (fileNameStart) with the delete character that is defined elsewhere in the Z specification.

$$\mathit{RootDirectory}'\ \mathit{drive?} = \mathit{RootDirectory}\ \mathit{drive?} \setminus \\ \{\ \mathit{name?} \mapsto (\ (\ \mathit{RootDirectory}\ \mathit{drive?}\ )\ \mathit{name?}\ )\ \}$$

Representation 2 – The section of Z code shown above removes the relation from the root directory that has a mapping that starts with a given file name (name?).

Figure 4 Example of removing a file from the root directory

Usually, a FAT file system has two file allocation tables. The first of these is primarily used for file operations and the second is used for error checking and recovery. The finished specification is limited to only allow one of each partition type per device or drive. This does not allow for there to be multiple file allocation tables in the Z specification in its current state. This was done because it simplifies the representation of the file system and does not require multiple versions of the same section of a filesystem to be updated when an operation is carried out. If it is required to include an additional file allocation table, it will be quite straightforward to implement into each version of the specification. In version 1, it just needs to have another sub sequence within the main sequence that represents each section of the file system. As version 2 uses a more abstract representation of the parts of a file system, it just needs to have two versions of the FAT relation or to allow one drive to partake in the existing relation more than once with additional predicates to ensure this.

When creating the operation to add a file to the system it became apparent that in order to make sure that the file got added to all the partitions with the correct pointers to the clusters that that file was using.

The add operation needed to be split into three separate operations that use preconditions to determine their order of execution. The first of these operations is to add the file to the root directory. In this example, the root directory is represented by a file name pointing to a positive natural number, where the natural number is the starting pointer for that file. In the official FAT specification, more information about the file is stored in the root directory however information such as the date and time that it was greeted is not relevant to the basic file operations that this specification aims to model. Therefore, it has been admitted but can be re-added later if it is necessary. The next operation assigns clusters from the new file into free clusters within the data partition. It uses the file allocation table to determine whether a cluster is free or not. When updating the statuses of the sectors that have been reassigned there is an additional relation which has been used which maps filenames to the setup pointers that they are using within the data partition. This relation (called clustersInFile) is then used to update the status of the file allocation table entries to show that the sectors are busy and cannot be used by another file. The most important section of a file system such as this is the file allocation table. This is because this part contains all information that links different sectors within the same file together. When designing the specification, the way I chose to represent this partition was to have a sequence of items where each item within this sequence represents an entry in the table. Each entry consists of a relation that maps the status of a sector onto a natural number. In practice, the number in this relation is only ever used when the status marks that cluster as being used. In this case, the number is the pointer to the next cluster in the file. In all other cases, the number is set to -1. One problem that was encountered by using this representation was that, due to the nature of relations in Z, they are represented as a set of bindings. Therefore, it was possible to have one "row" in the table to have many different relations that represent it in one entry. Then, in order to keep this representation true to the actual implementation of entries in the FAT, this set has then been restricted to only contain one binding in it. This can be seen in Figure 5 where sections of the schema that are unrelated to the file allocation table have been omitted.

```
—
  STATUS ::= USED | FREE | CORRUPTED | RESERVED | EOF
└
┌ FATworld
 allDrives: 𝔽 DRIVE
 FileAllocationTable: DRIVE ⤔ seq( STATUS ⤔ ℕ )
|
 ∀drive: allDrives • ∀FATentry: ran ( FileAllocationTable drive )
        • #FATentry = 1
└
```

Status is uses as a custom type that can have the values defined above. The FileAllocationTable relation is made up of drives mapped to a sequence where each element in the sequence is a status mapped to a natural number. So that this will represent a table where the sequence position is the row, the number of elements that can partake in the inner relation has been restricted to only contain one item. This is done by the line in the bottom section of the schema.

Figure 5 Implementation of the File Allocation Table

4.3 – Results Summary

This project has produced Z schemas for a disk imager and a file system. The disk imager Schema models how an image can be created at the bit level. This schema is based on requirements that have been based on looking at how existing tools operate, and requirements created by independent groups to establish how these tools should function. There were some problems implementing some of the functionality needed in a file system in Version 1 of the FAT schema. However, in version 2 these problems were overcome by using an abstract representation of the segments in the file system. Although version 2 works as a file system schema it is no longer able to work with the disk imager schema due to the abstraction used. The finished schemas can be now used to explain how tools and file system functions operate on the data with which they are being used.

Chapter 5 - Evaluation and Conclusions

5.1 - Evaluating the Disk Imager Z Schema to its Requirements

5.1.1 – Capturing Requirements of Forensic Tools

The full list of requirements created for a disk imager for this project can be seen in Appendix 1.1. This list drew on the work done in this area by NIST and SWGDE with some minor modifications that have been discussed earlier in Chapter 4.1.1. The modifications made to the requirements were done to force the Z schema to ensure that the source disk would not be changed during the process of an image being created. In practice this is handled by hardware (a write blocker) but the schema specifies this in the create image operation. This allows the schema to be used as technical requirements and as a guide for an implementation to be created. With regards to technical requirements for tools used in this field, this project found that there was not much material for use to explain how a tool should process its inputs. Therefore, there should be more work done in this area as this will allow for people who do not technically know how a tool functions to understand how it operates at a base level. This could be done either through creating formal specifications or with natural language. If natural language is used, then it is very important that it is unambiguous in order to avoid confusion.

5.1.2 – Creating a Disk Imager Z specification

The most important requirements for the disk imager are points one, two and five (full list in Appendix 1.1). These three requirements ensure that the created image is an exact copy of the data on the source disk. They also ensure that the data on the source disk is left unchanged. The imager also needs to handle any errors produced by corrupted sectors being present in the source disk.

- Through the requirements, it became very clear that all data that is stored on a disk needs to be included in the image. This includes any data that has previously been marked as hidden or has been deleted in the past. By representing the data as a sequence of bits it ignores concepts such as hidden files. This is because hidden files are an abstract concept used by the file system on top of the data. Also, depending on which file system is being mapped onto the image, deleted files may still be present on the disk. Therefore, they will be included in the image by default.
- Error handling came to be one of the most important requirements through doing research into existing tools and requirements as

discussed earlier. Therefore, implementing this requirement as was suggested by NIST fulfils this while preserving the functionality of the disk imager. The section of the operation that does this can be seen in Figure 6.

- The Z specification ensures that the source disk remains unchanged during this operation. Figure 7 shows how this was done by using the pre and posts states prop in Z. Also, this operation ensures that all the sectors that are on the source drive are the same in the new image at the same position. It does this for all the sectors that are not corrupted and handles this elsewhere in the same operation.

```
∀sectorAddress: dom sDrive • ∃s: seq( BIT ) |
        image = image ⊕ { sectorAddress ↦ s } ∧
        errorSectors! = errorSectors! ∪ { sectorAddress } •
        sectorAddress ∊ (corrupted source?) ∧
        #s = #(sDrive sectorAddress) ∧ s ↾ { ONE } = ∅
```

For sector addresses that are a member of the set of all corrupted sectors for that device it overwrites the corresponding sector in the image to be a sequence of bits called s. S must be the same length as the corrupted sector in the source drive and it must be made up of all zeros.

Figure 6 Error handling for corrupted sectors in disk imaging

```
drives′ source? = drives source?
```

Using pre and post conditions to ensure that the source drive remains unchanged.

```
∀sectorAddress: dom sourceDrive • sectorAddress ∉ corrupted source? ⇒
        sourceDrive sectorAddress = image sectorAddress
```

For all addresses in the source drive, if the sector address is not corrupted, the sector must be the same as the sector in the image with the same address.

Figure 7 Implementation of requirement 2

As mentioned before, the specification for a disk imager includes queries that are designed to search for a specified bit pattern within the created image. Currently, it is not possible to find the location of a bit pattern that spans multiple sectors. For this case, it returns a Boolean value if it finds the pattern or not. The reason that it cannot currently find the location is that Z has an operator "*a in b*" that gives a true value if the sub-sequence (a) is in the other sequence (b) [15, p. 119]. This does not provide a way of getting a position of the subsequence, rather it asserts if it is there or not. This is not ideal for a disk imager that will be used within forensic investigations as it will make the job of finding strings or file information a lot more complex. If this feature was to be added by the implementer then to verify this functionality, a formal specification may need to be created in a different language or this functionality may need to be verified through testing when this function has been created. Another way this could be achieved whilst still using the specification produced by this project would be to re-right it with a different representation of the data that is on the disk or in the image. This new representation may then allow for a query such as this to be carried out upon them.

5.2 – Conclusion for the Disk Imager Z Specification

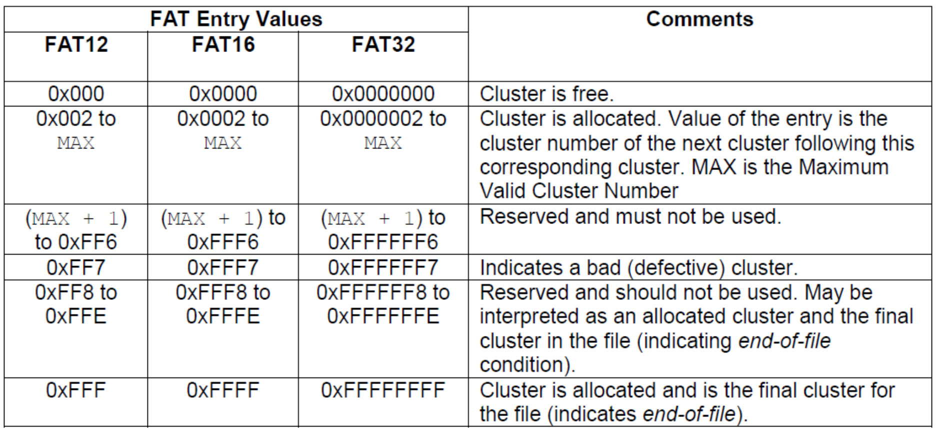
This project has found that the requirements that are created for forensic tools are lacking in some areas. For example, they tend to miss how the tool should be implemented to achieve the best accuracy. This is because they tend to focus on end user requirements and how the user should interact with the tool. This has led to the need to rely on test-based validation of the tools which is not ideal when they must work accurately on uncommon cases. This project has found that it is indeed possible to create a specification for a tool that is commonly used in forensic investigations. It is therefore possible to make specifications for different types of tool and begin the process of proving that they are correct. It has been proven that when these methods are used to implement or evaluate a piece of software, that software will more accurately implement those requirements that the Z specification is based on. Also, tools that have been implement the exact functionality of the formal specification will be able to predictably handle the uncommon data sets that would normally remain undetected through most of the testing. If these cases have been handled by the way that the formal specification defines the environment that it operates in then the tool will function normally and reliably under these extreme use cases.

This Z schema also allows for end users to gain an understanding of how the tool is designed to function at a base level. This understanding may increase the trust that is put into these tools and it will also make it easier for them to be updated or expanded upon. The schema also includes complex concepts such as hardware functionality. Features such as this were included in the requirements so that users were aware of the importance of this feature. It also can model and keep track of any sectors that need to be changed to extract the data held on the device as was discussed in chapter 2.1 – Extracting Evidence from Digital Devices. Using formal specifications to specify tools that are used in digital investigations allows for complex concepts to be represented quite concisely and clearly. If these concepts were to be described using natural language, it will require considerably more information and even then, it may still be ambiguous. Therefore, this is one of the reasons that using formal methods to evaluate forensic tools in this manner can be very useful.

5.3 – Evaluating the File System Z Schema

There are several limitations with the finished FAT specification. One of these is that there can only be one file system on any drive in the system. This was a design choice made to simplify the representation of drives and the file systems that are used on them. However, a more complex relationship between drives and their systems may allow for a drive to have multiple FAT file systems or different types of file systems mapped onto different regions of that drive. Many drives in the real world can often have multiple file allocation tables and data segments in them. This is a limitation of the current FAT Z schema and it would be good to have this as a feature in the future. This is not currently possible within the finished Z specification because the way that it keeps track of segment is, each drive has one relation for each segment. Each drive can only partake in that relationship once. Once again, just as with only allowing one drive to have one file system, this was done to simplify the representation of segments within the file system. If this specification should be turned into a more concrete implementation, the segment relationships could be split up into multiple versions that allow the same drive to participate in all the new relations. This still preserves the abstract representation that is in the created Z specification. Another limitation with this specification is that there cannot be two files with the same name on a drive at any one time. To allow this feature, it would need a more complicated way of identifying each file, for example using a file signature along with the file name. This can use more information about the file that is stored in the root directory or it could use an ID system to identify the files.

In the official FAT specification provided by Microsoft [3], the file allocation table consists of a set of entries where each one is a predetermined number of bits long. The size of each entry is determined by the version of the file system. Microsoft explains the different entry values that each entry can take (see the table in Figure 8). Part of these values is used as flags which states the status of the cluster that the entry refers to. In the case that the cluster is marked as allocated then the next cluster to be read on the disk will contain the pointer for the next cluster in the file. See Figure 8 for an example of how these flags are represented between the two specifications. The Z specification for the file system also can assign each one of these flags to an entry in the file allocation table. The representation of this table differs slightly between the specifications in Z and by Microsoft. The Z specification uses a great deal more abstract whilst keeping all the key information about each entry. This is done by using the relations within the Z as discussed earlier, although the way the file allocation table is represented between the two specifications varies a lot. Both representations, functionally, hold the same information about each of the clusters that they would refer to.



FAT Entry Values			Comments
FAT12	FAT16	FAT32	
0x000	0x0000	0x0000000	Cluster is free.
0x002 to MAX	0x0002 to MAX	0x0000002 to MAX	Cluster is allocated. Value of the entry is the cluster number of the next cluster following this corresponding cluster. MAX is the Maximum Valid Cluster Number
(MAX + 1) to 0xFF6	(MAX + 1) to 0xFFF6	(MAX + 1) to 0xFFFFFF6	Reserved and must not be used.
0xFF7	0xFFF7	0xFFFFFF7	Indicates a bad (defective) cluster.
0xFF8 to 0xFFE	0xFFF8 to 0xFFFE	0xFFFFFF8 to 0xFFFFFFE	Reserved and should not be used. May be interpreted as an allocated cluster and the final cluster in the file (indicating *end-of-file* condition).
0xFFF	0xFFFF	0xFFFFFFFF	Cluster is allocated and is the final cluster for the file (indicates *end-of-file*).

File allocation table entry values for different versions of FAT from the official FAT specification [3, p. 16]

```
STATUS ::= USED | FREE | CORRUPTED | RESERVED | EOF
```

Defined flag values that are used in the Z schema found in Appendix 1.4

Figure 8 How FAT values are implemented in the Z schema

In the official FAT specification and the Z schema, the root directory is used to hold key information about the files that are currently being held on the disk. These two implementations differ in that each entry in the directory from the official specification contains additional information about the file. This additional information includes creation date and time, when the file was last accessed or modified and the total file size. This extra information is very useful for when people are looking at files and need to organize them. However, when modelling a file system this is unnecessary information to store and will needlessly complicate the representation of this segment. The information that is not currently in the Z representation of the root directory is the total file size. This could be used by the specification if it were added to check that all the data was present in the file when reading it from the disk. That said it is not essential to modelling the core functionality of the file system so therefore it has been left out.

In the official FAT specification by Microsoft, the boot segment has defined sections that are used to hold information used by the operating system and if booting from the drive. While the Z specification does have a way of storing the contents of the Boot sector for each drive, it remains as a sequence of bytes. This is because once the segments have been separated out, it is not essential for basic file operations which this specification is designed to do. Due to the representation used within the Z specification, it currently does not read into this partition to find out where the other segments start and end on the disk. Instead, the user of this system simply provides the abstract representation of the other segments when a new drive is added. The data partition is represented in a very similar way between the official specification and the Z specification. This is because in Z it is done by using a sequence of bytes. The position of a byte in that sequence is the address of where that byte would be on a drive. This is very similar to how bytes and sectors are addressed within hard drives. As a byte is a defined type in Z it can be redefined to be a cluster of bits that can be any length without changing any of the logic used by the specification.

While modelling a file system the most important decision that was made was to change to a more abstract representation of the data on the disk. This decision was made because certain operations that are key to a file systems operation would have been too complex or not possible within the constraints of the project, such as the language being used or the amount of time available. As discussed in chapter 4.2 there were problems during the implantation of the delete operation in version 1 of the specification. This included not being able to update the flags of the clusters that are used by a file. Z could not iteratively read along the chain of sectors and their pointers to update them all with the representation used in version 1. To solve this problem version 2's abstract representation uses an additional relationship to map a file name to a sequence of sector pointers that are used by that file. This new relationship also holds the order of the sectors that are used by that file. Due to changing the representation to what is used in version 2, the specification lost the ability to map the exact data from created images from the disk imager specification. This was a good trade-off as it allowed for the key functionality of a filesystem to still be formally specified.

5.4 – Conclusions for the File System Z Schema

Although version 2 of the file system specification does fully model the basic functions needed by a file system, within forensic computing investigations, functions such as add and delete are not required by investigators as these operators will change the data that may be relied on in court. This is expressed in the principles of handling digital evidence [4, p. 6]. Therefore, when mapping a file system onto an image that will be used for this purpose, that system does not need to have complete functionality as so long as it can extract files from the image that file system, it will be fulfilling its purpose. For a FAT system, this would only need to have the functionality to read into the image and ascertain where the segments of the file system start and finish then read files from the other 3 segments.

With regards to requirements for the FAT file system, they were already published and quite extensive however there was not a version that used a formal specification to describe how a file system functions on the data on a disk. Creating a file system in Z, along with the disk imager, allows for the process of data acquisition and extraction to be accurately modelled. This can be used to explain how a file system operates on a created image and it explicitly shows that when a file is being read from an image, the data held within the image is unchanged. As discussed earlier, this is a key philosophy of data extraction within forensic computing. In version 2 of the file system Z schema abstraction was used to represent the data and allow for operations to be created. It was important to add these operations to this schema because although they are not technically needed for an investigation they form a complete environment to create a full picture of what a file system will do on the data. They can be used for many other aspects of dealing with data in an image.

5.5 – Project Conclusions

This project has found that using formal methods to evaluate or to be used during development processes can be very useful to increase the accuracy of the tools that are created. The biggest problem that forensic tools currently have is the inconsistencies in the different levels of documentation that are produced for each tool. As mentioned before, this could be for several reasons including the commercial and legal issues surrounding them. Creating formal specifications can start to standardise this as it will specify the key functionalities within the tool and how it should process them to remain accurate and correct throughout. If a library of specifications is created for the different tools that are used during forensic computing investigations, then this library was publicly available and was maintained by an independent group. It would be easy to build on a tool or change it by a different person or group to fit a slightly different investigation while still maintaining the accuracy gained from using a specification, to begin with. It has been shown that using formal techniques within industry can increase the efficiency and accuracy of the tools produced [12] . Having the specifications would also increase the transparency of how the tools operates on the data that they are processing. This is important for forensic investigations to show that the tools are following a predefined and validated process. Finally, the specification created for tools used in digital forensic investigations can be used to create test criteria that can then be used to make testing quicker, easier and more accurate.

When requirements are written with the aim of creating a formal specification based on them, it can force the requirements to be written to a higher, more technical, standard. This is mainly as they will not be so ambiguous, causing the tools that are produced from the requirements be more accurate. The tools that are produced will also be able to have their accuracy proven by using the specification which wouldn’t have been able to be done in this way before. The better requirements will be able to provide a better understanding to the end user about how the tool functions and will allow them to use it more effectively. This will help investigations be carried out quicker and more accurately.

Chapter 6 - Suggested Further Work

There are some areas in this project that can be worked on further and some aspects of this project that can be used as the basis for other projects worked on by different groups. This chapter will explore the possibilities of these areas.

When the specification for the disk imager needs to deal with the sectors on the disk that are corrupted, it currently has no way of detecting whether the sector itself is corrupted or not. It looks up the sector address in a set of all the corrupted sectors that have been provided to the system before the operation starts. There is no other way of currently detecting if a sector is corrupted, within this operation. In the future, it would be very useful to have a way for this specification to detect these sectors within the system as this is a feature that an implemented disk imager would have to do itself. Some file systems do have a way of marking if a sector has been corrupted or not. However, using this method requires being able to map a file system on top of the disk before the image is created. Using this method will have to be done with caution to make sure that none of the data on the original disk is changed.

An area that has been touched on previously within this report is that the Z schema for the FAT file system does not currently operate at the bit level. Instead, it uses abstraction to allow it to show the functionality of the file system. In the future, it will be useful to have a version of the specification which operates at the bit level and can perform file operations in terms of bits or bytes. This will be very important for specifying file systems as it will then be possible to empirically compare the outcomes from the formal specification to what is on the disk of the implemented file system. This implementation would be able to take a sequence of bits and then be able to read the contents of that sequence and determine where the segments start and finish based on the lookup table in the Boot sector of the file system. This was not able to be done within this project due to time constraints and limitations within Z itself. These limitations include not being to convert binary into natural numbers easily. To overcome this limitation a more complex structure within Z is needed or, a different specification language may be needed to carry this out. Another approach to this could be the one that was mentioned towards the end of chapter 5.3. This is where file systems that will be used exclusively within forensic investigations do not need a full set of file management functions. They only actually need to be able to read the data on the disk. Therefore, it may be possible to create a cut down version of most file systems that can then be used to extract files from various devices that are involved within investigations. These different versions of the file systems may be finished implementations of the cutdown file system or, initially, they may start out as a formal specification similar to what was used during this project. These specifications will be able to represent data without having to use abstract concepts that were used in version 2 of the FAT file system Z schema in this project. This will make it easier to specify many different file systems at the bit level and may, therefore, cause more and more file systems to be mathematically specified.

One way that both the specification for the disk imager and the file system could be used in future projects is that they can be used to create a set of test criteria for existing tools. These test criteria would be better than just using requirements on their own as it is mathematically shown what the end outcome and functionality of the implementation should be. This is because requirements can sometimes be ambiguous whereas a set of test criteria based on a formal specification can be provably correct. Within forensic investigations, this will be very useful as it will give tools a higher certainty that they provide correct and accurate results all the time.

Within law enforcement, there has already been work done to attempt to capture requirements for data extraction and analysis. This work does provide a useful set of guidelines for people who are conducting the extraction and analysis however they are very high level and from an end user's point of view. For example, there is no reference to the technical inputs and outputs that the tool used by these investigators would need to be able to do. This project aimed to start be able to fill in the gaps left in this work. Therefore, it would be possible to create a more technical set of requirements or guidelines using the specifications created in this project or other formal specifications made for other tools. One of the main points that are mentioned within these requirements is that the user should not directly use the source information without having prior knowledge or training regarding handling digital evidence. If a tool were created from a formal specification such as the one created during this project, it would be mathematically provable and easy to see that the information in the image is an exact copy of the source information. This specification also shows using formal logic how the operations interact with the original data and that they do not change the original data. During investigations, this will be useful to show that the evidence has not been tampered with in the process of investigation.

Appendices

Appendix 1.1: Requirements for a Disk Imager

The requirements that were captured during this project. They have been used to the create the disk imager Z schema found in Appendix 1.2.

I. Copy ALL the data exactly between a source disk and a new target disk that has enough space on it for the new image.
II. While creating the image form the data in the source disk, the data must not be changed during image creation.
III. It must use bit streaming to copy each individual bit rather than copying through sectors or chunks of the source disk. This will increase the accuracy of the disk imager.
IV. The disk imager must leave the source disk completely unchanged throughout the entire copying process. This will protect any potential evidence that could be held on the source disk. It will therefore require a write blocker to prevent writing to the source disk. The blocker can be integrated into the disk imager or be an external program. If it is external, we do not need to make this consideration within the requirements.
V. If the disk imager comes across a corrupted sector on the source disk, then it must be able to handle this. This can be dealt with by not copying the corrupted sector across to the target disk, leaving empty space (all 0s), and carry on copying the rest of the disk. This would need to flag up that is has encountered a corrupted sector and the location of that sector. This can then be dealt with by the user in the future.
VI. A way of verifying that the new disk is indeed an exact copy of the source disk. This should be in accordance with a defined standard.
VII. Create an image that can be viewed at a later data by another piece of software. The image should be in the needed format by the given image reader.
VIII. There should be a time constraint on the whole imaging process. You should be able to prove that creating an image of a given disk size in a reasonable amount of time.

Appendix 1.2: Disk Imager Z schema

Z specification for a disk imager. Text in red are the comments for the schema below it. Each line in the comments corresponds with the matching line in the schema.

```
— section Imager parents standard_toolkit
└
— [ DEVICE, SECTOR ] └
—
  BIT ::= ONE | ZERO
└
—
  BOOL ::= TRUE | FALSE
└
```

The three types of devices that imaging can be done in
deadbox - the device remains unchanged through the image creation
pTwo - the device may have been changed during this process. It is not known how much of the data has been changed
remote - the data is on something like a server so NON of the data is original
corrupted - a set of all sector addresses that are corrupted
SectorLength - a constant for the length of a sector for the imager. This will be set to 8 bits (byte)

```
|
  deadbox, pTwo, remote: 𝔽 DEVICE
  corrupted: DEVICE ⟶ 𝔽 ℕ
  SectorLength: ℕ
|
  deadbox ∩ pTwo ∩ remote = Ø
  SectorLength = 8
└
```

drives map a drive to a relation that maps a sequence of sectors (represented as a sequence of bits). The position in the sequence is the address for that sector
images is the new disk images that will be created
old unchanged devices ⟶ new images
changed relates the drives to the addresses that have been changed when extracting the data
---
All drives will be in changed (if they are unchanged then they will point to an empty set)
All of the created images will be in the drives domain
All the addresses that have been changed are a subset of all of the addresses on the drive - addresses are the sequence position
deadboxes have no address that have been changed
pTwo and remote devices can have any number of addresses changed

```
┌ World
  drives: DEVICE ⟶ seq( seq( BIT ) )
  images: DEVICE ⤔ DEVICE
  changed: DEVICE ⇸ 𝔽 ℕ
|
  deadbox ⊆ dom images
  dom changed =  dom images
  dom drives ⊆ ran images
  ∀i: dom images •  changed i ⊆ dom ( drives i )
  changed⦇ ran images ∩ deadbox ⦈ = { }
  changed⦇ ran images ∩ (pTwo ∪ remote) ⦈ ⊆ ℙ ℕ
└
```

At initialisation there are no images that have been created

```
┌ WorldInit
  World ′
|
  images′ = ∅
└
```

source? - source drive
target? - the drive that the image will be related to
image - created image
sDrive - a temp var for the source drive with the changed sectors removed from the sequence
errorSectors! - set of all addresses where there was a corrupted sector/error
---
the target drive has not already been used to create another image
there is no changes to any device during this operation
the source drive is unchanged during this operation
sDrives is equal to the original source drive without the changed sectors in it
if the sector is NOT corrupted then the sector is overwritten to the new image in the same position
If the sector is corrupted, then the address is added to errorSectors! and a sequence of all ZEROs is put in the new image at the address of the corrupted sector
ensures that the sectors in the source and the image are the same and in the same order except for corrupted sectors
adds the image relating to the target disk to the drives
adds the device with the new image on it to the images relation

```
┌ CreateImage
  ΔWorld
  source?, target? : DEVICE
  image, sDrive: seq( seq( BIT ) )
  errorSectors!: 𝔽 ℕ
|
  target? ∉ dom drives
  changed′ = changed
  drives′ source? = drives source?
  ∀sectorAddress: dom (drives source?) | sDrive = sDrive ⊕
          { sectorAddress ↦ (drives source?) sectorAddress } • sectorAddress ∉
          changed source?
  ∀sectorAddress: dom sDrive | image = image ⊕
          { sectorAddress ↦ sDrive sectorAddress } • sectorAddress ∉ (corrupted
          source?)
∀sectorAddress: dom sDrive • ∃s: seq( BIT ) | image = image
```

```
        ⊕ { sectorAddress ↦ s } ∧ errorSectors! = errorSectors! ∪ {
        sectorAddress } • sectorAddress ∈ (corrupted source?) ∧ #s = #(sDrive
        sectorAddress) ∧ s ↾ { ONE } = ∅
 ∀sectorAddress: dom sDrive • sectorAddress ∉ corrupted
        source? ⇒ sDrive sectorAddress = image sectorAddress
 drives′ = drives ∪ { target? ↦  image }
 images′ = images ⊕ { source? ↦ target? }
└
```

this operation will not find a bit pattern that does NOT spans different sectors

drive? - the drive that has the image that you want to search on it
bitPattern - the pattern of bits that you want to search for
---
if there is no bit pattern to search for then there are no locations for it in the searched image
patternLocation! - a set of sector addresses that have that bit pattern in it
        "infix" matches one sequence into another (like in) -> p119 in z-manual

```
┌ SearchImage2
 ΞWorld
 drive?: DEVICE
 bitPattern?: seq( BIT )
 patternSectorNums!: 𝔽 ℕ
|
 drive? ∈ dom drives
 #bitPattern? ≤ SectorLength
 bitPattern? = ∅ ⇒ patternSectorNums! = ∅
 ∀sectorAddress: dom (drives drive?) • bitPattern? infix
        (drives drive? sectorAddress) ⇒ patternSectorNums! =
        patternSectorNums! ∪ { sectorAddress }
└
```

this operation will not find a bit pattern that spans different sectors

drive? - the drive that has the image that you want to search on it
bitPattern - the pattern of bits that you want to search for
patternStartOffset! - the bit offset for the starting point of the bit pattern that you are searching for
startingSectorNum! - the number of the sector that the bitpattern starts in
---
The drive must be in the system
For all sector addresses there exists an image that is the concatenation of all of the sectors in the image of drive?. If the bit pattern is in the concatenated image then the starting locations are reported back to the user
        Does NOT report the bit offset, rather non deterministically choses a natural number to report

Reports if the bit pattern is in the image or not

```
┌ SearchImage
 ΞWorld
 drive?: DEVICE
 bitPattern?: seq( BIT )
 patternStartOffset!: ℕ
 startingSectorNum!: ℕ
 patternFound!: BOOL
|
 drive? ∊ dom drives
 ∀sectorAddress: dom (drives drive?) • ∃image: seq( BIT ) •
          ∃Location: ℕ | bitPattern? infix image ⇒
          patternStartOffset! = Location ∧ startingSectorNum! = (Location −
          (Location mod SectorLength)) div sectorAddress •  image = image ⁀
          drives drive? sectorAddress
  ∀sectorAddress: dom (drives drive?) • ∃image: seq( BIT ) |
          ( ( bitPattern? infix image ⇒ patternFound! = TRUE  ) ∨  (¬(bitPattern?
          infix image) ⇒ patternFound! = FALSE) ) • image = image ⁀ drives
          drive? sectorAddress
└─
```

Appendix 1.3: File system Z schema

Z specification to model a FAT file system using a representation that shows the disks as a sequence of bits. In this specification, the operation - DeleteFile *does not update the file allocation table to reflect that the file has been removed. Text in red are the comments for the schema below it. Each line in the comments corresponds with the matching line in the schema.*

This specification is referred to as version 1 within this document.

```
— section FATsystem parents standard_toolkit
└
— [ DEVICE ] └
—
  BIT ::= ONE | ZERO
└
—
  ERROR ::= FileNotOnDisk | NoError
└
```

The char used to mark if the file is deleted

1110010 is E5h in hex which is the delete char for FAT

```
I
  DeleteChar, Zeros: seq( BIT )
  SectorLength: ℕ↘1↖
|
  DeleteChar = 〈 ONE, ONE, ONE, ZERO, ZERO, ONE, ZERO, ONE 〉
   Zeros = 〈 ZERO, ZERO, ZERO, ZERO, ZERO, ZERO, ZERO, ZERO 〉
  SectorLength = 8
└
```

drives - all the devices that have a fat file system in the world
partitionTables - all the partition tables for the drives in the system
binConverter - a relation that maps binary numbers (as a sequence) to their equivalent natural numbers
---
each drive must have 4 partitions and the first one must be 512 sectors long.
Each sector in each partition must be 8 bits (a byte) long
the first partition offset must be 1 (boot always at the start and index starts at 1)
and there must be 4 partition offsets in exact partition table

```
┌ FATworld
  drives: DEVICE ⟶ seq( seq( seq( BIT ) ) )
  partitionTables: DEVICE ⟶ seq( ℕ )
  binConverter: seq( BIT ) ↔ ℕ
|
  ∀disk: ran drives • ∀part: ran disk • #part = 4 ∧ #(head
            part) = 512 ∧ (∀sector: ran part • #sector = SectorLength)
   ∀offsets: ran partitionTables • offsets(1) = 1 ∧ #offsets = 4
└
```

```
┌ FATworldInit
  FATworld ʼ
|
  drivesʼ = ∅
  partitionTablesʼ = ∅
└
```

change to get the image from the other world schema

image? - the image that is being loaded into system
device? - the device that the image relates to from the world
boot, fat, root, data - temp vars to store the partitions from the image
---
the device must not be already in the system of all drives
the device must have its partition table already in the fat world
the partition table remains unchanged

all of the partitions are set to be sub sequences of the image

these sub sequences are then added to the drives relation as a fat file system

```
┌ ImageToFat
  ΔFATworld
  image?: seq( seq( BIT ) )
  device?: DEVICE
  boot, fat, root, data: seq( seq( BIT ) )
|
  device? ∉ dom drives
  device? ∈ dom partitionTables
  partitionTablesʼ = partitionTables
  binConverterʼ = binConverter

  ∀sectorAddress: dom image? • ∃s: seq( seq( BIT ) ) | boot =
            s • s = s ⊕ { sectorAddress ↦ image?(sectorAddress)} ∧ sectorAddress
            ≤ (partitionTables device?)(2)
  ∀sectorAddress: dom image? • ∃s: seq( seq( BIT ) ) | fat =
            s • s = s ⊕ { sectorAddress ↦ image?(sectorAddress)} ∧ sectorAddress
            ≥ (partitionTables device?)(2) ∧ sectorAddress ≤ (partitionTables
            device?)(3)
  ∀sectorAddress: dom image? • ∃s: seq( seq( BIT ) ) | root =
            s • s = s ⊕ { sectorAddress ↦ image?(sectorAddress)} ∧ sectorAddress
            ≥ (partitionTables device?)(3) ∧ sectorAddress ≤ (partitionTables
            device?)(4)
  ∀sectorAddress: dom image? • ∃s: seq( seq( BIT ) ) | data =
            s • s = s ⊕ { sectorAddress ↦ image?(sectorAddress)} ∧ sectorAddress
            ≥ (partitionTables device?)(4)

  drivesʼ = drives ∪ { device? ↦ 〈 boot, fat, root, data 〉 }
└
```

```
┌ FATtoImage
  ΞFATworld
```

```
 image!: seq( seq( BIT ) )
 device?: DEVICE
|
 device? ∈ dom drives
 image! = drives device?(1) ⁀ drives device?(2) ⁀ drives device?(3) ⁀ drives
         device?(4)
└─
```

image? - the image that the partitions need to be found for
device? - the device that the image is on
---
the device must not already be in the partitionTables relation
the device must not already be in the drives relation
drives remains unchanged through this operation

converts the value of the byte in the numbered sectors to decimal and adds the partitions to the table relation (the first one is always 1 as it is at the start of the disk)

```
┌─ AddPartitionTable
 ΔFATworld
 image?: seq( seq( BIT ) )
 device?: DEVICE
|
 device? ∉ dom partitionTables
 device? ∉ dom drives
 drives′ = drives
 binConverter′ = binConverter

  ∃part2, part3, part4: ℕ • part2 = binConverter(image?
         464) ∧ part3 = binConverter(image? 480)  ∧ part4 =
         binConverter(image? 496)  ∧ partitionTables′ = partitionTables ∪
         {device? ↦ ⟨1, part2, part3, part4⟩}
└─
```

device? - the device that has the file on
name? - the name of the file to delete as a sequence of bytes
---
the device must be in the fat system

All the bytes in the file name must be 8 bits long

If the file name is not in the root directory, then report the error and the drives remain unchanged
The entry in the root directory (3rd partition in the FAT disk) has its first char overwritten with the DeleteChar - drives device? 3 is the root directory in device?

```
┌─ DeleteFile
 ΔFATworld
 device?: DEVICE
 name?: seq( seq( BIT ) )
 error!: ERROR
|
 device? ∈ dom drives
```

device? ∊ dom partitionTables
partitionTables′ = partitionTables
binConverter′ = binConverter

∀bytes: ran name? • #bytes = SectorLength

¬(name? infix (drives device? 3)) ⇒ error! = FileNotOnDisk ∧ drives′ = drives
∃char: seq(BIT) • (char = head name? ∧ name? infix
(drives device? 3)) ⇒ error! = NoError ∧ (∃fileStart: ℕ • (drives device? 3) fileStart = char ∧ (drives′ device? 3) = (drives device? 3) ⊕ { fileStart ↦ DeleteChar })
└─

Appendix 1.4: File system Z schema

Z specification to model a FAT file system using an abstract representation that models the functionality of the file system. Text in red are the comments for the schema below it. Each line in the comments corresponds with the matching line in the schema.

This specification is referred to as version 2 within this document.

```
— section FATsystem2 parents standard_toolkit
└
— [ DRIVE, FILENAME, BYTE ] └
```

STATUS is the flag for each cluster's entry in the file allocation table. ZERO = available, ONE = busy

```
—
  STATUS ::= USED | FREE | CORRUPTED | RESERVED | EOF
└
```

BootSector - Maps drives to a sequence of bytes
FileAllocationTable - represented by a sequence of ClusterPointer mapped to a status flag (in this case 0 or 1 for free or busy). The position in the sequence represents the row of the FAT
Data - represented by a cluster pointer mapping to a byte of data
RootDirectory - represented by filenames mapped to pointers. No other file information will be stored in this specification - NOTE there can only be one file with any given file name
---
If a drive is in allDirves then it must be in each of the partition relations
Each row in the FAT can only have 1 entry per row
If a file is in the root directory, then it is also in the clustersInFile relation and the first pointer in the sequence is the pointer that is in the root directory

```
┌ FATworld
  allDrives: 𝔽 DRIVE
  BootSector: DRIVE ⤔ seq( BYTE )
  FileAllocationTable: DRIVE ⤔ seq( STATUS ⤔ ℕ)
  Data: DRIVE ⤔ seq( BYTE )
  RootDirectory: DRIVE ⤔ ( FILENAME ⤔ ℕ↘1↖ )

  clustersInFile: DRIVE ⤔ ( FILENAME ⤔ seq( ℕ↘1↖ ) )
|
  dom BootSector = allDrives ∧ dom FileAllocationTable =
          allDrives ∧ dom Data = allDrives ∧ dom RootDirectory = allDrives ∧
          dom clustersInFile = allDrives
  ∀drive: allDrives • ∀FATentry: ran ( FileAllocationTable drive ) • #FATentry = 1
  ∀drive: allDrives • ∀FileName: dom ( RootDirectory drive )
          • FileName ∊ dom (clustersInFile drive) ∧ head(clustersInFile drive
          FileName) = ( RootDirectory drive ) FileName
└
```

Initialises all the relations to the empty set

```
┌ FATworldInit
  FATworld ′
|
  allDrives′ = ∅
  BootSector′ = ∅
  FileAllocationTable′ = ∅
  Data′ = ∅
  RootDirectory′ = ∅
  clustersInFile′ = ∅
└
```

newDrive? - the drive that you want to add the FAT system to
boot? - the sequence of bytes that make up the boot partition
numOfClusters? - the number of clusters that will be in the data partition - must be a positive natural
---
the newDrive must not be in the system already
adds the new drive to allDrives
Adds the new drive to the partition relations
Data is set up to have the number of clusters that have been set by the input
Adds the drive to clustersInFile

```
┌ AddNewDrive
  ΔFATworld
  newDrive?: DRIVE
  boot?: seq( BYTE )
  numOfClusters?: ℕ↘1↖
|
  newDrive? ∉ allDrives

  allDrives′ = allDrives ∪ { newDrive? }
  BootSector′ = BootSector ∪ { newDrive? ↦ boot? }
  FileAllocationTable′ = FileAllocationTable ∪ { newDrive? ↦ ∅ }
  ∃data: ℕ ⟶ BYTE | Data′ = Data ∪ { newDrive? ↦ data } •
            #data = numOfClusters? ∧ max dom data ≤ numOfClusters?
  RootDirectory′ = RootDirectory ∪ { newDrive? ↦ ∅ }
  clustersInFile′ = clustersInFile ∪ { newDrive? ↦ ∅ }
└
```

This operation works like AddNewDrive but it is for a drive that already has data stored on it. It needs the partition information given to it for it to run

```
┌ AddExistingDrive
  ΔFATworld
  newDrive?: DRIVE
  boot?: seq( BYTE )
  fat?: seq( STATUS ⤔ ℕ)
  data?: ℕ↘1↖ ⤔ BYTE
  root?: FILENAME ⤔ ℕ↘1↖
  allFilePointers?: FILENAME ⤔ seq( ℕ↘1↖ )
|
  newDrive? ∉ allDrives

  allDrives′ = allDrives ∪ { newDrive? }
  BootSector′ = BootSector ∪ { newDrive? ↦ boot? }
  FileAllocationTable′ = FileAllocationTable ∪ { newDrive? ↦ fat? }
```

```
 Data′ = Data ∪ { newDrive? ↦ data? }
 RootDirectory′ = RootDirectory ∪ { newDrive? ↦ root? }
 clustersInFile′ = clustersInFile ∪ { newDrive? ↦ allFilePointers? }
└
```

name? - the name of the file to add
drive? - the drive that you want to add the file to
---
the drive must be in the FATworld system
the file name must not already be in the RootDirectory
an entry is added to the root directory mapping the name to a cluster address. This cluster address must be a cluster that is on the drive and the cluster must be free (status of 0) in the FileAllocationTable

```
┌ AddFileToROOT
 ΔFATworld
 name?: FILENAME
 drive?: DRIVE
|
 drive? ∈ allDrives
 name? ∉ dom ( RootDirectory drive? )
 FileAllocationTable = FileAllocationTable′ ∧ Data = Data′
          ∧ BootSector′ = BootSector
 ∃freeCluster: ℕ • ∃FATrow:ℕ | RootDirectory′ drive? =
          RootDirectory drive? ∪ { name? ↦ freeCluster } • freeCluster ∈ dom (
          Data drive? ) ∧ ( ( FileAllocationTable drive? ) FATrow ) FREE =
          freeCluster
└
```

name? - the name of the file to add
drive? - the drive that you want to add the file to
newData? - the sequence of bytes that is the new file
---
the drive must be in the FATworld system
the file name must not already be in the RootDirectory
the file must not already be in clustersInFile
Finds a disjoint sequence of pointers that are marked as free (available) in the FAT and uses these pointers to override the bytes in Data for these pointers. The order of the data is preserved by this sequence. Also saves this sequence to clustersInFile

```
┌ AddFileDATA
 ΔFATworld
 name?: FILENAME
 drive?: DRIVE
 newData?:seq( BYTE )
 freeDataPoints: 𝔽 ℕ
|
 drive? ∈ allDrives
 name? ∈ dom ( RootDirectory drive? )
 name? ∉ dom ( clustersInFile drive? )
 FileAllocationTable = FileAllocationTable′ ∧
          BootSector′ = BootSector
 ∃freePoints: seq( ℕ ) • freePoints ∩ freePoints = ∅ ∧ max
```

(ran freePoints) ≤ max (dom (Data drive?)) ∧ #freePoints = #newData? ∧ (∀point: ran freePoints • FREE ∊ dom (FileAllocationTable drive? point) ∧ Data′ drive? = Data drive? ⊕ { point ↦ newData? (freePoints point) } ∧ clustersInFile′ drive? = clustersInFile drive? ∪ { name? ↦ freePoints })

└

name? - the name of the file to add
drive? - the drive that you want to add the file to
---
the drive must be in the FATworld system
the file name must already be in the RootDirectory
the file must already be in clustersInFile
Updates each clusters entry in the FAT to show that it is busy and changes its "next pointer" to point to the next sequence in the file

"(clustersInFile drive? name?) FATrow) + 1" gets the next cluster in the sequence of all clusters in the file

if it is the last cluster in the file the next pointer in FAT will be set to "-1". If it is less then it will be set to the next pointer in the file

┌ AddFileToFAT
ΔFATworld
name?: FILENAME
drive?: DRIVE
|
drive? ∊ allDrives
name? ∊ dom (RootDirectory drive?)
name? ∊ dom (clustersInFile drive?)
Data = Data′ ∧ clustersInFile = clustersInFile′ ∧
BootSector′ = BootSector
∀clusterNum: dom (FileAllocationTable drive?) •
∃updatedFATrow: STATUS ⟶ ℕ | FileAllocationTable′ drive?= FileAllocationTable drive? ⊕ { clusterNum ↦ updatedFATrow • clusterNum ∊ ran (clustersInFile drive? name?) ∧ clustersInFile drive? name? clusterNum = max dom (clustersInFile drive? name?) ⇒ (updatedFATrow = FileAllocationTable drive? clusterNum ⊕ { EOF ↦ -1 }) clustersInFile drive? name? clusterNum < max dom (clustersInFile drive? name?) ⇒ (updatedFATrow = FileAllocationTable drive? clusterNum ⊕ { USED ↦ ((clustersInFile drive? name?) clusterNum) + 1 })

└

drive? - the drive has the FAT system that the file is stored on
FileName? - the name of the file that you want to delete

The drive must be in the collection of allDrives (by extension it is in the partition relations)
the file name must be in the root directory
the file must be a part of clustersInFile
The data partition will not be changed by this operation
If the file name is in the directory so it is the relation that contains the file name is removed from the RootDirectory

For each row in the FAT that has the same row number as the pointer that used by the current file. It overwrites that rows status to be ZERO
"(FileAllocationTable′ drive?) remCluster ⊕ { 0 ↦ ZERO }" updates the entry for the row that matches with the removed cluster (next = 0, status = 0). Next is set to 0 as it does not matter what the pointer is set to as the file is deleted
removes the file form clustersInFile

```
┌ DeleteFile
  ΔFATworld
  drive?: DRIVE
  name?: FILENAME
  FirstPointer: ℕ
|
  drive? ∊ allDrives
  name? ∊ dom (RootDirectory drive?)
  name? ∊ dom ( clustersInFile drive? )
  Data′ = Data ∧ BootSector′ = BootSector
  RootDirectory′ drive? = RootDirectory drive? ⩥ { name? ↦
          ( ( RootDirectory drive? ) name? ) }
  ∀removedCluster: ran ( clustersInFile drive? name? ) •
          ∀rowNum: dom( FileAllocationTable drive? ) • removedCluster ∊ ran ( (
          FileAllocationTable drive? ) rowNum ) ⇒ ( FileAllocationTable drive? )
          removedCluster  = ( FileAllocationTable′ drive? ) removedCluster ⊕ {
          FREE ↦ 0 }
  clustersInFile′ drive? = clustersInFile drive? ⩥
          { name? ↦ clustersInFile drive? name? }
└
```

drive? - the drive has the FAT system that the file is stored on
name? - the name of the file that you want to delete

The drive must be in the collection of allDrives (by extension it is in the partition relations)
If the file name must be in the root directory
the file name must be in the clustersInFile relation
For all the cluster pointers in the file it gets the corresponding byte of data and then puts it in the output sequence in the same position that the pointer was in for clustersInFile

```
┌ ReadFile
  ΞFATworld
  name?: FILENAME
  drive?: DRIVE
  file!: seq( BYTE )
|
  drive? ∊ allDrives
  name? ∊ dom ( RootDirectory drive? )
  name? ∊ dom ( clustersInFile drive? )
  ∀filePointer: ran( clustersInFile drive? name? ) • file! = file! ⊕ { ( clustersInFile
          drive? name? filePointer ) ↦ ( Data drive? filePointer ) }
└
```

drive? - the drive has the FAT system that the file is stored on

ClusterNum? - the number of the cluster that is now corrupted
---
the cluster number must be in the range of the sector addresses
it sets the FAT entry to be corrupted

```
┌─ SetCurrupted
  ΔFATworld
  drive?: DRIVE
  ClusterNum?: ℕ↘1↖
|
  ClusterNum? ∊ dom ( FileAllocationTable drive? )
  FileAllocationTable′ drive? ClusterNum? = { CORRUPTED ↦ -1 }
└─
```

References

[1] National Institute of Standards and Technology, "Digital Data Acquisition Tool Specification," 10 April 2004. [Online]. Available: https://www.nist.gov/sites/default/files/documents/2017/05/09/pub-draft-1-dda-require.pdf. [Accessed 03 02 2019].

[2] Scientific Working Group on Digital Evidence, "SWGDE Minimum Requirements for Testing Tools used in Digital and Multimedia Forensics," 22 January 2019. [Online]. Available: https://www.swgde.org/documents/Current%20Documents/SWGDE%20Minimum%20Requirements%20for%20Testing%20Tools%20used%20in%20Digital%20and%20Multimedia%20Forensics. [Accessed 22 January 2019].

[3] Microsoft, "Microsoft FAT Specification," 30 August 2005. [Online]. Available: http://read.pudn.com/downloads77/ebook/294884/FAT32%20Spec%20%28SDA%20Contribution%29.pdf. [Accessed 15 2 2019].

[4] Association of Chief Police Officers, "ACPO Good Practice Guide for Digital Evidence," March 2012. [Online]. Available: https://www.digital-detective.net/digital-forensics-documents/ACPO_Good_Practice_Guide_for_Digital_Evidence_v5.pdf. [Accessed 18 04 2019].

[5] A. Marshall, "Standards, regulation & quality in digital investigations: The state we are in," *Digital Investigation,* vol. 8, no. 2, pp. 141-144, 2011.

[6] International Organization for Standardization, "ISO/IEC 27001:2013 Information technology -- Security techniques -- Information security management systems -- Requirements," International Organization for Standardization, October 2013. [Online]. Available: https://www.iso.org/standard/54534.html. [Accessed 15 April 2019].

[7] A. Marshall and R. Paige, "Requirements in digital forensics method definition: Observations from a UK study," *Digital Investigation,* vol. 27, pp. 23-29, 2018.

[8] T. van der Storm and J. van den Bos, "Bringing domain-specific languages to digital forensics," *International Conference on Software Engineering (ICSE),* vol. 33, pp. 671-680, 2011.

[9] C. Stelly and V. Roussev, "Nugget: A digital forensics language," *Digital Investigation,* vol. 24, pp. S38-S47, 2018.

[10] Active@ Data Recovery Software , "NTFS Basics," [Online]. Available: https://www.ntfs.com/ntfs_basics.htm. [Accessed 2019 03 01].

[11] T. Sammes and B. Jenkinson, Forensic Computing a Practitioner's Guide, 2 ed., Swindon: Springer-Verlag London Limited, 2007.

[12] A. Hall, "What does industry need from formal specification techniques?," *Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques,* pp. 2-7, 1998.

[13] Community Z Tools Project, "CZT: Community Z Tools," 09 April 2016. [Online]. Available: http://czt.sourceforge.net/. [Accessed 10 February 2019].

[14] J. M. Spivey, "An introduction to Z and formal specifications," *Software Engineering Journal ,* vol. 4, no. 1, pp. 40-50, 1989 .

[15] J. M. Spivey, The Z Notation: A Reference Manual, 2nd ed., Oxford: Prentice Hall International (UK) Ltd, 1998.

Bibliography

NPCC project to produce common standards for digital forensics