University of St Andrews

Senior Honours Project

A Diagnostic Tool for FUSE Modules

Final Report

Matthew Dooler

April 2014

Supervisor: Dr. Graham Kirby

0.1 Abstract

Filesystems provide the mechanisms for retrieval and storage of information on a computer, often organising files using a hierarchical naming scheme. Filesystems generally abstract over local physical media while others access information over the network. FUSE (Filesystem in Userspace) is a framework allowing filesystems to be implemented in user space, avoiding modification of the kernel by developers and users. This project presents a software diagnostic tool that facilitates development of FUSE filesystems. The tool provides three main facilities: a wizard that interactively guides developers through implementation of a FUSE filesystem from scratch, an automated test suite, and a debugger.

0.2 Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 15,850 words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

0.3 Acknowledgements

I wish to express my sincere appreciation to Dr. Graham Kirby for his guidance during practical work and write-up of this project.

I am also grateful to Prof Alan Dearle for his advice during the initiation of this project.

Contents

List of Figures

Chapter 1

Introduction

FUSE (Szeredi & Henk, 2004) brings filesystem development out of the kernel-programming domain, allowing developers to implement filesystems in higher-level languages. This in turn allows non-root users to mount filesystems without compromising system security or stability.

Many operating systems implement a virtual filesystem mechanism, including Linux, Mac OS X and Microsoft Windows. User space programs and libraries make calls against a virtual filesystem layer (VFS) which abstracts over a concrete filesystem implemented in the kernel. FUSE for Mac OS X and Linux presents a kernel module to the VFS layer which passes calls to a user space FUSE module implementation. A user space FUSE module is a concrete filesystem implementation and is responsible for actually performing filesystem operations and maintaining internal state.

A basic FUSE implementation could simply delegate calls to an existing filesystem, while a more sophisticated implementation may store files online, abstracting over network protocols or web platforms such as SSH, Amazon S3 and Google Drive.

1.1 Context

The official FUSE documentation lists around 50 FUSE modules (FUSE, 2013), ranging from simple filesystem extensions to fully peer-to-peer filesystems with fault tolerance and load balancing. A large number of FUSE modules are also developed internally by organisations and hobbyists to address their own problems. There is therefore a large community of FUSE developers who would benefit considerably from additional tools that aid FUSE module development.

1.2 Problem definition

1.2.1 Filesystem development from scratch

Implementing a FUSE filesystem from scratch can be difficult as FUSE specifies an API of 42 functions that a filesystem should implement. Some functions, such as `utimens` which sets file access time metadata, can be left unimplemented and the filesystem would work but lack certain features.

FUSE also provides default implementations for a small number of functions if they are not defined by the FUSE filesystem. `open`, `opendir` and `release` return `0` if the function is not defined, which indicates success. This allows the caller to continue making calls (such as `read` or `readdir`) which the filesystem may still be able to handle. `statfs` also has a default implementation which populates the `statvfs` struct with some default values on basic filesystem properties. For all other unimplemented functions FUSE returns `-ENOSYS ("Function not implemented")`, which indicates to the caller that it cannot use the function. A filesystem developer should therefore aim to implement all 42 functions to avoid unexpected application behaviour.

In the early stages of development a filesystem will often simply not work, as operating systems and applications do not expect to be calling an incomplete filesystem. Filesystem prototyping is a time consuming and costly process as it requires a significant amount of effort to implement a filesystem that works at even a basic level. It also encourages a waterfall-style development model as it is difficult to begin analysing progress until a significant amount of the filesystem is implemented. Testing and evaluation are therefore more likely to be left until the end of the project, at which point it may be too late to correct missed requirements or even delay delivery of the project. Incremental approaches to filesystem development are certainly possible and could be facilitated by using a tool that guides developers through implementation of FUSE functions in a certain order.

1.2.2 Documentation

It is important that filesystem functions behave according to their specification, otherwise unexpected behaviour will occur and the filesystem could become unreliable or frustrating to use. However, information on these functions is disparate and documentation is dispersed across the Internet in varying levels of detail. There is no single canonical source of documentation on every FUSE function due to the fact that many functions are required to conform to POSIX specifications and it would not make sense to duplicate this information in the FUSE documentation. Further confusion occurs when the name of a FUSE function does not match the name of its equivalent POSIX function. Other FUSE functions are based on similar POSIX functions but are expected to perform slightly different tasks or use different data types, so these have their own custom specification in the FUSE documentation. This problem could be solved for future developers by investigating each FUSE function and compiling a set of links to online documentation on each one.

1.2.3 Testing

As of writing, the latest version of FUSE (2.9.3) specifies 42 functions which can all be called in various sequences passing a range of different parameters. User space programs and libraries (e.g., `glibc`) expect filesystem functions to conform to POSIX specifications, but these specifications generally consider functions in isolation rather than stipulating the order in which calls will be made. This results in different programs and libraries making different call sequences to achieve the same thing, making wide coverage of a filesystem during testing difficult. For example, one function may set up internal state required for later calls to work correctly but certain programs may use a different function or even skip the initial call altogether. A simple example is the use of the Unix commands `touch test1.txt` and `nano test2.txt`. These result in creation of two identical files, but `touch` makes a call to `utime` to explicitly set the access and modification times of the file even though `create` should set these automatically. The programs also call `getattr` and `fgetattr` in different sequences.

1.2.4 Platform independence

Another cause of uncertainty is the use of FUSE on different operating systems, as their VFS layers and userspace libraries often differ slightly. This results in FUSE functions being called with different parameter values and in different orders. A Mac OS X version of FUSE, `OSXFuse`, is a superset of Linux `libfuse`, meaning that it may be necessary to implement additional functions to reliably run a Linux FUSE filesystem on Mac. The following table explains the purpose of these functions:

Function	Purpose
`setvolname`	Sets the name of the volume. Would be called if the user tried to rename the mountpoint icon on their desktop.
`exchange`	Atomically exchanges data between two files. This is optional and needs to be reported via `getattrlist` if implemented, but some applications may still require it.
`setbkuptime`	Sets backup time.
`setcrtime`	Sets creation time.
`getxtimes`	Get times set by `setbkuptime` and `setcrtime`.
`setchgtime`	Undocumented.
`chflags`	Sets file flags such as immutability or visibility.
`setattr_x`	Optional function provided for implementation convenience and increased efficiency. It replaces `chmod`, `chown`, `utimens`, `truncate`, `ftruncate`, `chflags`, `setbkuptime` and `setcrtime` with a single call.
`fsetattr_x`	Version of `setattr_x` that takes a filehandle.

Implementing `setattr_x` makes the individual setter functions (`chmod`, `chown`, etc.) redundant as they will not be called, allowing them to be removed. However, such a filesystem would no longer operate correctly on Linux as `setattr_x` would not be used and the individual setters would not be implemented. Implementing both `setattr_x` and the individual setters would result in some redundancy and increased chance of incorrect code, so it would be sensible for the developer to simply leave `setattr_x` unimplemented.

FUSE for Linux also has a number of functions that are unavailable in `osxfuse`, but must still be implemented for the filesystem to work correctly on Linux:

Function	Purpose
`ioctl`	Controls underlying device parameters of special files.
`poll`	Waits for a file to become ready for I/O.
`write_buf`	Writes a buffer to a file.
`read_buf`	Reads a file into a buffer.
`flock`	Applies or removes a lock on a file.
`fallocate`	Changes the allocated disk space for a file.

The developer therefore needs to be aware of the fact that these functions will not be called when their filesystem runs on Mac, and other functions should not rely on these Mac-only functions being called beforehand.

Extending a filesystem to support multiple platforms is therefore an involved process requiring a significant amount of manual testing to gain confidence that the filesystem will work efficiently, correctly and robustly. These discrepancies are also largely undocumented and can only be discovered by trial and error. A test suite consisting of tests for these different scenarios would provide the developer with enough information to allow them to address each issue directly.

1.2.5 Debugging

There is also another obstacle to filesystem development, which is the complexity involved in debugging a filesystem. This is theoretically possible with general-purpose debuggers, but running them against a mounted filesystem can prove difficult and a large amount of irrelevant information is displayed to the user. It would be of benefit to filesystem developers if a debugger displayed information in a FUSE-centric way, for example by focusing on FUSE function invocations and the values passed to them.

1.3 Aim

The aim of the project is to facilitate FUSE filesystem development by producing a diagnostic tool that makes development more straightforward, reducing development time and programmer effort. The tool will address all of the issues outlined in the previous section. It will allow a guided incremental approach to filesystem development, and assist the developer by providing extensive testing features that can be used to emulate the behaviour of different platforms and applications. It will also provide debugging functionality so that it is clear which calls are being made into the filesystem.

Any FUSE filesystem will be mountable using the tool, and it will be intuitive and easy to use. The tool will be suitable for developers with different levels of experience, as one part of the tool will guide new developers through implementation of a basic filesystem while another part will report more technical issues with the filesystem. Detail will also be encapsulated when it is not necessary to see everything at once.

1.4 Success criteria

The criteria for success of this project were that it met all of its primary and secondary objectives as originally defined. Meeting the tertiary objective was not essential but would enhance the tool further.

A working tool was delivered which fulfilled all of the original primary and secondary objectives to a high standard, so the project was considered a success. It also has a number of qualitative features that are essential for a development tool, as it is very robust, efficient and easy to use. It also runs on both Linux (via libfuse) and Mac OS X (via OSXFuse), and was tested extensively on both platforms both during development and at the end of the project. See Appendix A for detailed information on how the project was tested and evaluated against these criteria.

1.5 Use cases

A diagnostic tool for FUSE filesystems would suit a wide range of potential use cases. This section will present some of the main example use cases of such a tool.

1.5.1 Filesystem development from scratch

A developer may need to write their own filesystem from scratch, but lack filesystem-programming experience. They are likely to choose FUSE since it is simpler to debug software in userspace and is also very unlikely to crash the kernel if any errors occur in their implementation. However, they would find that the documentation presents them with a very large interface to implement (i.e., `getattr` and the other 41 functions), and that it lacks overall guidance and detailed information about many of the functions.

The developer could use a diagnostic tool to guide them through the process of implementing the filesystem, providing guidance on order of implementation in order to get something at least partially working. This would allow them to develop much faster than when taking a trial-and-error approach to implementing functions, and provide them with a partially functional filesystem that they can begin to test against. With relevant and detailed documentation at each stage the filesystem would also be implemented to a much higher standard, and be less likely to fail due to non-conformance to specifications.

1.5.2 Debugging on a new platform

An experienced filesystem-developer may have produced a filesystem but find that their filesystem is performing unexpectedly on a different platform. They could use a debugger to find that the platform is making function calls in an order that they would not normally expect, and is also making requests for platform-dependent files that did not occur before. They can then adapt their implementation to support this behaviour, now that they know precisely how the platform is behaving.

1.5.3 Quality assurance of a filesystem

A development team may build a filesystem and find that it is becoming very complex. They have written unit tests and tested it on their development platforms, but they find that their filesystem can be used in a wide range of complex ways and manually testing every case would be too time-consuming and expensive. A tool that exercises their filesystem automatically as if it were being used in a production environment would help highlight unexpected problems and provide the team with higher confidence in their code and assurance that the filesystem works correctly. It may also be very difficult to reproduce the exact behaviour that causes rare bugs, and a tool that could replay workloads repeatedly and deterministically would help filesystem developers address these issues.

1.6 Report structure

The remainder of the report is divided into the following nine chapters:

- Chapter 2 covers the main objectives of the project, classified in order of importance.
- Chapter 3 describes the work already done in this area, referencing background literature and related projects.
- Chapter 4 specifies the properties that the tool must have, in the form of a software requirements specification document.
- Chapter 5 discusses the software development approach taken and justification for its adoption.
- Chapter 6 considers the ethical issues associated with the project.
- Chapter 7 covers the high level design and structure of the tool, discussing design decisions, challenges and justifications for unusual design features.
- Chapter 8 discusses the implementation of the diagnostic tool in more depth, focusing on mechanisms and data structures.
- Chapter 9 critically evaluates the project against the original objectives and other related projects.
- Chapter 10 summarises the project, emphasising key achievements, drawbacks, and future work that could be done on the project.

Chapter 2

Objectives

A number of project objectives were identified and classified as primary, secondary or tertiary. Meeting all of the primary objectives will be essential for the project to succeed. Achieving any secondary objectives will be considered a major success, but they are not essential and may not be completed due to time constraints or unforeseen complexity. The project will gain even more value if tertiary objectives are achieved, but these will only be attempted if time remains after completing all primary and secondary objectives.

The primary objectives of this project were to produce a number of software tools that assist the programmer during development and verification of FUSE filesystems. These included:

- a **wizard** designed to guide a developer through implementing a filesystem
- a **test suite** to find defects in a FUSE implementation
- a **debugger** to visualise execution of a filesystem.

A secondary objective was to extend the test suite by automatically analysing real call sequences and resulting behaviours in different operating systems, and running these as unit tests against a filesystem. This would produce an extensive and realistic set of tests, resulting in wider code coverage and increased confidence in the robustness of an implementation.

Another secondary objective was to operate with any FUSE filesystem that can run as a program, initially covering filesystems written in C. The ultimate aim of this objective is to keep the solution generic enough to allow support for any language binding that wraps the shared FUSE userspace library. It is not expected to work with language bindings that use their own private version of the FUSE library.

A tertiary objective was to investigate how such filesystems perform under high demand by producing a stress testing utility. Not only does this highlight performance issues but it is also more likely to detect race conditions in multi-threaded implementations.

Chapter 3

Context Survey

In this chapter we survey previous literature covering filesystem development, with focus on user-space and FUSE filesystems. We also cover existing tools that aid filesystem testing and debugging, critically evaluating them and explaining their relationship to this project.

3.1 User-space filesystems

Filesystems are responsible for organising data on a computer system, and usually abstract over physical media providing logical, reliable and efficient access. They allow application developers to program against a single interface rather than requiring low-level knowledge about every storage medium. This ensures that programs are more compatible across different computer systems, and also ensures that this is not at the cost of stagnating innovation in storage media.

The earliest filesystems were indeed designed to abstract over local storage devices, and were originally closely intertwined with the hardware and operating system. The virtual file system (VFS) abstraction was later supported by many filesystems, which introduced a new layer allowing different concrete filesystem implementations to be accessed using a single interface. Later filesystems, such as Network File System (NFS) originally developed in 1984, allowed computers to access resources over the network as if they were stored on a local drive. This and the success of the Internet gave rise to a number of other distributed filesystems, but development was still time-consuming and generally restricted to kernel programmers.

Userfs was written in 1993 as a proof-of-concept to demonstrate how users can securely mount their own filesystems. The author intended it to be a prototyping tool, allowing filesystems to be developed in user-space and eventually ported to the kernel (Fitzhardinge, 2002). According to the author, this tied filesystems too closely to the kernel and it was later discovered that many of these filesystems would never even be implemented as kernel modules (Fitzhardinge, 2002). Userfs was significant as it demonstrated the potential for user-space filesystems, and showed that they could be efficient, secure and practical. It also highlighted the difficulties in developing and prototyping filesystems for the kernel.

The Linux Userland Filesystem (LUFS) consisted of a kernel module which delegated VFS calls to a user-space daemon, and was a framework designed to allow implementation of filesystem logic in user space. LUFS was eventually replaced by FUSE, which entered development two years later in 2004 (Malita, 2013; Szeredi & Henk, 2004).

3.2 FUSE filesystems

FUSE was successful as its filesystems are represented by executable programs, as opposed to LUFS which required its filesystems to be implemented in shared objects (S. Singh, 2006). Mounting a FUSE filesystem by simply running its executable made development and debugging easier than when linking shared objects that represent the filesystem. FUSE now supports a large number of language bindings (C++, Java, Haskell, Python, Ruby, and many more), and it was officially merged into the Linux kernel in version 2.5.14 (Calleja, 2005).

FUSE has also allowed proprietary filesystems to be used on Linux, as some cannot be included with the official Linux release due to licensing issues. For example, exFAT is a filesystem designed for flash drives, and aims to allow interoperability between Mac OS and Windows. It could not be included as part of official Linux releases due to licensing restrictions, but support was enabled via a FUSE module (Hachman, 2013), allowing users to easily mount exFAT-formatted drives on any Linux system with FUSE installed.

3.3 FUSE for Mac OS X

FUSE was ported to Mac OS X 10.4 by the MacFUSE project, and supported existing FUSE filesystems along with additional language bindings. Mac OS X was claimed to have lacked strong filesystem support before this, and only a small number of filesystems were available (Shavit, 2007). The MacFUSE project is, however, no longer being maintained and development stopped in 2009 rendering it incompatible with Mac OS X 10.7 and above (*MacFUSE Project*, n.d.).

Its successors were Fuse4X and OSXFuse, and Fuse4X is currently merging with OSXFuse to avoid confusion (*Fuse4X Project*, 2013). OSXFuse aims to be compatible with filesystems developed for MacFUSE and supports filesystems written for Linux FUSE (Fleischer, 2011). Therefore at this point in time and the near future, OSXFuse appears to be the most viable framework to use for FUSE on Mac OS X.

3.4 General purpose debugging

Valgrind is an analysis tool that can be used to profile programs and detect memory management and threading bugs. Many users have experienced problems mounting filesystems using Valgrind, and the proposed workarounds can be complex (Kutzner, 2007). However as it is still technically possible to use Valgrind with FUSE filesystems, this project will avoid addressing issues Valgrind already supports.

The GNU Project debugger (GDB) is a popular tool used to debug running programs, supporting programs written in C and other languages. It is useful for finding the cause of lower-level problems such as memory errors, and also supports breakpoints and stepping through code execution. GDB usually requires programs to be compiled with debugging symbols enabled, making it difficult to debug filesystems which have no public source code available as they cannot be recompiled. This also means that GDB does not debug the production binary. This difference in machine code is problematic when debugging problems such as segmentation faults as the memory layout is likely to be different. For example, a segmentation fault may occur in a filesystem due to an error in the implementation, but when compiled with debugging symbols enabled this could hide the fact that a memory access violation occurred and prevent the developer from finding the root cause of the problem. A higher-level approach that focuses on FUSE API calls would therefore be more appropriate when debugging a FUSE filesystem.

3.5 Filesystem monitoring

When mounting a FUSE module it is possible to pass a debug flag that will instruct the FUSE library to print debugging information every time a FUSE function is called. However the data is very limited, printing a very small amount of information for each function call. It does not dump the contents of the bigger data structures, but these contain large amounts of important information that may be crucial to the developer when trying to understand the cause of a problem. No interactive facilities are provided, such as being able to step-through execution of a filesystem.

HFSDebug was a tool built by Amit Singh, who was the main developer of MacFUSE. It was a debugger for Apple's HFS+ filesystem and was intended to allow monitoring of read-only HFS+ filesystems to acquire statistics and other volume information (A. Singh, 2004). fileXray is a more modern commercial tool that has superseded the functionality of HFSDebug (iohead, n.d.) and allows the user to monitor real-time filesystem activity, along with some advanced volume introspection utilities. It works by using its own custom HFS+ implementation that accesses the volume contents directly, providing it with more information and greater control than what is available through the standard programming interface.

Project Goanna proposes a monitoring framework for filesystem development (Spillane, Wright, Sivathanu, & Zadok, 2007). It uses the Unix `ptrace` system call to intercept filesystem calls at the operating system level (i.e., the point at which system calls are made into the OS). The framework is designed to work with existing monitoring libraries, and claims that it can be used to debug any type of filesystem. It appears to be aimed at kernel-mode filesystem development, with the assumption that user-space libraries like FUSE are only used for prototyping and will eventually be ported to the kernel. Unfortunately it has no documentation and as of writing has not yet been publicly released. If a stable version of the framework had been available with documentation then it could have been investigated further and possibly used in this project to make and intercept filesystem calls. However, although the framework operates in user-space it monitors kernel system calls, and certainly not FUSE-level calls. This would be less useful to FUSE filesystem developers as the reported calls would not always match up with the FUSE functions being called.

The Linux Test Tool Matrix includes 12 generic filesystem diagnostic tools (Martin, 2006). `dbench`, `fs_maim`, `IOZone`, `PostMark`, `stress`, `mongo` and `xdd` are useful for stress testing, benchmarking and performance analysis. The others are testing utilities and are discussed in the next section.

3.6 Testing

The remaining filesystem diagnostic tools in the Linux Test Tool Matrix are testing utilities. `Bonnie` tests creation, reading, and deletion of small files. `fsx` is a tool by Apple which exercises a filesystem by creating, writing, reading, closing, and deleting a test file. `LTP` (The Linux Test Project) is a test suite for the Linux kernel, and includes `fs_inode` which tests directory structures, and `lftest` which tests support for large files.

Tuxera ported an extensive POSIX filesystem test suite to Linux (originally written by Pawel Jakub Dawidek for FreeBSD), that rigorously tests 11 system calls with 3601 tests (Tuxera, 2009). Passing this test suite would provide the user with a high level of confidence that the filesystem is implemented correctly. The tests appear to have been written manually but unfortunately only include a small amount of human-readable information for failing tests. This makes it difficult to understand which tests fail and why, and the developer is not given any positive guidance about how to fix the problem. The suite is also built specifically for a number of concrete filesystems (UFS, ZFS, ext3, XFS and NTFS-3G) as it aims to ensure each one conforms precisely to specification rather than ensuring general robustness across a range of filesystems.

3.7 Summary

All of the existing test suite utilities are either frameworks or aimed at specific filesystem types, so a FUSE filesystem developer must either write their own tests or run the test suite with the expectation that some tests will not be relevant. The first option is often not a viable solution due to the complexity of filesystems, and the second is unreliable especially when there is limited documentation surrounding the tests. Finally, developers are not given any positive guidance about how to fix issues causing failing tests even though developers of different filesystems are likely to face very similar challenges.

Chapter 4

Requirements Specification

4.1 Preface

The following document is based on the Volere Requirements Specification Template (Robertson & Robertson, 2012), and includes functional and non-functional requirements specifying the properties that the diagnostic tool must have to meet the project objectives. It is divided into user, system and domain requirements. Most of the requirements were elicitated early on in the project through discussions with the project supervisor, but some arose later on during prototyping. Some requirements depend on other requirements being satisfied before they can be achieved, and these are specified in a dependency list as part of the requirement.

4.2 Glossary

GUI: Graphical User Interface
FUSE Module: An implementation of a FUSE filesystem
FUSE Binary: A compiled and linked FUSE module
Passthru Filesystem: A FUSE module that provides a wrapper around an existing filesystem. This is often used when a fully-functional FUSE module is required.

4.3 User requirements

4.3.1 General

Requirement #1	**Requirement Type:** Functional user requirement	**History:** Created 2013/09/27
Description: The tool should allow the user to input the location of a FUSE filesystem written in C.		
Rationale: The original FUSE implementation is written in C so this will ensure all "standard" FUSE filesystems are supported. This also allows the potential to support filesystems written against frameworks that extend the original C FUSE implementation.		
Originator: Student & Project Supervisor		
Dependencies: None.		**Importance:** High

Requirement #2	**Requirement Type:** Functional user requirement	**History:** Created 2013/09/27
Description: The tool should take a compiled FUSE binary as input, rather than source code.		
Rationale: To give the developer the flexibility to use their own development toolchain when compiling their filesystem. This also avoids the filesystem having any dependencies on the diagnostic tool, making rollout to production easier and less error-prone.		
Originator: Student		
Dependencies: Requirement #1.		**Importance:** High

Requirement #3	**Requirement Type:** Functional user requirement	**History:** Created 2013/09/27
Description: The tool should be able to to work with filesystems written using FUSE bindings for other languages (such as Java and Python) provided that the binding uses the shared userspace FUSE library that is implemented in C.		
Rationale: To allow developers to use the tool to implement and diagnose filesystems written in other languages than C.		
Originator: Student & Project Supervisor		
Dependencies: Requirements #1 and #2.		**Importance:** Low

Requirement #4	**Requirement Type:** Non-functional user requirement	**History:** Created 2013/11/28
Description: The tool should allow the user to quickly and conveniently switch between any of the three core features (wizard, test suite and debugger). It should not take any longer than 15 seconds to operate a different feature, assuming the user understands how to use the tool.		
Rationale: To avoid user frustration and ultimately reduce development time.		
Originator: Student		
Dependencies: All requirements regarding implementation of the wizard, test suite and debugger.		**Importance:** Low

Requirement #5	**Requirement Type:** Functional user requirement	**History:** Created 2013/11/28
Description: The tool should provide access to the three core features (wizard, test suite and debugger) from a single GUI.		
Rationale: To reduce redundancy and allow the user to quickly switch between different features, helping to meet requirement 4.		
Originator: Student		
Dependencies: All requirements regarding implementation of the wizard, test suite and debugger.		**Importance:** Medium

4.3.2 Wizard

Requirement #6	**Requirement Type:** Functional user requirement	**History:** Created 2013/10/01
Description: The wizard should take a FUSE module from the user and inform them what functionality needs to be implemented next.		
Rationale: To allow the developer to build the filesystem incrementally.		
Originator: Student & Project Supervisor		
Dependencies: Requirements #1 and #2.		**Importance:** High

Requirement #7	**Requirement Type:** Non-functional user requirement	**History:** Created 2013/10/01
Description: The wizard should provide positive guidance, telling the developer what they should do rather than what does not work.		
Rationale: To reduce the amount of time spent by the developer investigating and deducing what an error message actually means in terms of what needs to be implemented next.		
Originator: Student & Project Supervisor		
Dependencies: Requirement #6.		**Importance:** High

Requirement #8	**Requirement Type:** Functional user requirement	**History:** Created 2013/10/01
Description: The wizard should allow the user to skip the implementation of certain functionality if desired, and allow them to move on to the next increment of functionality.		
Rationale: Not all functionality will be applicable to every type of filesystem.		
Originator: Student		
Dependencies: Requirement #6.		**Importance:** Medium

Requirement #9	**Requirement Type:** Functional user requirement	**History:** Created 2013/11/28
Description: The wizard should display links to resources documenting the required implementation of the current function under test.		
Rationale: It is difficult to find documentation for some functions, so this reduces development time and provides more incentive for the developer to follow the specification.		
Originator: Student		
Dependencies: Requirement #6.		**Importance:** Low

4.3.3 Test suite

Requirement #10	**Requirement Type:** Functional user requirement	**History:** Created 2013/10/01
Description: The test suite should execute sequences of FUSE calls against the filesystem under test and a passthru filesystem, comparing the user-visible states of the two filesystems after each call. "User-visible state" refers to files and directories that a user would be able to browse, including all visible metadata on each file and directory.		
Rationale: Allows errors to be reported to the developer as early as possible, and provides enough information for the developer to find the root cause of a filesystem problem.		
Originator: Student & Project Supervisor		
Dependencies: None.		**Importance:** High

Requirement #11	**Requirement Type:** Functional user requirement	**History:** Created 2013/10/01
Description: The test suite should allow the user to import a file containing sets of call sequences.		
Rationale: Allows the user to build test data, and also distribute it.		
Originator: Student & Project Supervisor		
Dependencies: Requirement #10.		**Importance:** High

Requirement #12	**Requirement Type:** Functional user requirement	**History:** Created 2013/10/01
Description: Part of the test suite should include a logger, which logs calls made against a FUSE filesystem of the user's choice and allows them to be grouped into logical call sequences by the user.		
Rationale: Constructing test data by hand is time-consuming and error-prone, but automatic logging allows much more rigorous test data to be generated very quickly.		
Originator: Student & Project Supervisor		
Dependencies: Requirement #11.		**Importance:** Medium

Requirement #13	**Requirement Type:** Functional user requirement	**History:** Created 2013/10/01
Description: The logger for the test suite should allow the user to tag call sequences with a short description of their action and the application used. The sequence will also be automatically tagged with the name and version of the operating system that the logger is running on.		
Rationale: Adds significant value to test sequences by putting them in context.		
Originator: Student & Project Supervisor		
Dependencies: Requirement #12.		**Importance:** Medium

Requirement #14	**Requirement Type:** Functional user requirement	**History:** Created 2013/10/01
Description: The logger for the test suite should allow captured data to be exported in the format expected by the test suite.		
Rationale: Allows the user to use their captured data in the test suite.		
Originator: Student & Project Supervisor		
Dependencies: Requirements #11 and #12.		**Importance:** Medium

Requirement #15	**Requirement Type:** Non-functional user requirement	**History:** Created 2013/10/01
Description: The logger should have a quick user interface and be simple to use. For example, instructions should be clear and concise, and confirmation prompts should be avoided.		
Rationale: The user will be interacting with the logger UI and filesystem simultaneously so will need to carefully and quickly control what sequences are captured when.		
Originator: Student		
Dependencies: Requirement #12.		**Importance:** High

4.3.4 Debugger

Requirement #16	**Requirement Type:** Functional user requirement	**History:** Created 2013/10/01
Description: The debugger should take a FUSE module, mount it, and display every call invocation and return. It should visualise parameters and return values in a tree structure in the GUI.		
Rationale: The user may need to see any data passed to any FUSE function, and as the data often consists of complex nested objects a tree is the most natural structure. Any linear data, such as an array of integers, would be represented by a root node (the variable name) with a child for every integer. It is also often useful for the user to be able to view function invocations and returns as separate events as there could be a problem while the function is executing or they could interleave.		
Originator: Student & Project Supervisor		
Dependencies: None.		**Importance:** High

Requirement #17	**Requirement Type:** Non-functional user requirement	**History:** Created 2013/10/01
Description: Call parameters displayed in the debugger should be concise and compact, but allow the user to drill down into more detailed information.		
Rationale: A high number of FUSE calls occur on a mounted filesystem, even for seemingly small operations. Some FUSE structs also have many attributes, so there is too much data to show all at once.		
Originator: Student & Project Supervisor		
Dependencies: Requirement #16.		**Importance:** High

Requirement #18	**Requirement Type:** Functional user requirement	**History:** Created 2013/10/01
Description: The debugger should optionally let the user manually pause and advance execution of the filesystem using the GUI.		
Rationale: Allows the user to step through a possibly overwhelming number of calls, and also observe how applications behave when certain calls are being made.		
Originator: Student		
Dependencies: Requirement #16.		**Importance:** Medium

4.4 System requirements

4.4.1 General

Requirement #19	**Requirement Type:** Functional system requirement	**History:** Created 2013/10/04
Description: Methods will be invoked on FUSE modules by intercepting the mount attempt from within libfuse (or osxfuse on Mac OS X).		
Rationale: This avoids a complete mount of the filesystem, which would be prone to interference from the operating system. It also allows the diagnostic tool to interact with compiled FUSE binaries without requiring any custom integration or instrumentation of the filesystem code.		
Originator: Student		
Dependencies: All requirements for the wizard and test suite.		**Importance:** High

Requirement #20	**Requirement Type:** Functional system requirement	**History:** Created 2013/10/04
Description: Real method calls will be intercepted and logged using wrappers around the libfuse/osxfuse code that calls these functions. Communication with the diagnostic tool will be done through named semaphores and pipes.		
Rationale: Inter-process communication is more reliable for debugging as it avoids crashing the diagnostic tool if an error were to occur in the provided filesystem.		
Originator: Student		
Dependencies: All requirements for the debugger and test suite logger.		**Importance:** High

4.4.2 Wizard

Requirement #21	**Requirement Type:** Functional system requirement	**History:** Created 2013/10/01
Description: Each test in the wizard will be given a weighting depending on how many dependencies each has on functions being exercised by other tests. Optional functions will have lower priority.		
Rationale: It is more natural for the developer to implement functions in-order due to the dependencies between them.		
Originator: Student & Project Supervisor		
Dependencies: All wizard requirements.		**Importance:** High

4.4.3 Test suite

Requirement #22	**Requirement Type:** Functional system requirement	**History:** Created 2013/10/01
Description: The logger for the test suite will export call sequences to a JSON file, consisting of sets of call sequences organised into groups. Each call will include all parameters and the values passed.		
Rationale: JSON is lightweight and can be parsed easily. JSON will also be used for almost all inter-process communication throughout the project so exporting in this format avoids redundant re-formatting.		
Originator: Student		
Dependencies: All test suite requirements.		**Importance:** Medium

Requirement #23	**Requirement Type:** Functional system requirement	**History:** Created 2014/02/11
Description: The call logger for the test suite will log pointers (i.e., memory addresses) if it is unable to serialise the data being pointed to. It also does not make sense to serialise certain data, such as an empty buffer passed to a `read` function or a pointer to an internal callback function implemented by `libfuse`. File handles also lose their context as they are generated by the filesystem at runtime. The test suite should therefore discard pointers from test data and reallocate them at runtime. The test suite should also track file handles passed to it by the filesystem at runtime and pass these to functions that require a file handle. Internal callback functions that would have been implemented by `libfuse` should be reimplemented and the new function pointer passed to any functions that require it.		
Rationale: Buffers, function pointers and file handles cannot be serialised reliably and their pointers are meaningless outside of the filesystem process. The test suite would fail very quickly without reallocation.		
Originator: Student		
Dependencies: All test suite requirements.		**Importance:** High

4.4.4 Debugger

Requirement #24	**Requirement Type:** Functional system requirement	**History:** Created 2013/10/25
Description: Some FUSE functions will be called with pointers to values or structs that the function can set. For example, `getattr` is called with an empty `stat` struct that the implementation should populate to return file attributes. The debugger should therefore capture changes to values passed via pointers in function parameters.		
Rationale: Many FUSE functions return results through their parameters, as the return value usually only indicates success or failure.		
Originator: Student		
Dependencies: All debugger requirements.		**Importance:** High

4.5 Domain requirements

Requirement #25	**Requirement Type:** Functional domain requirement	**History:** Created 2013/09/27
Description: The tool will run on Mac OS X (10.7.5) and Linux (Scientific Linux 6.4) using the latest versions of osxfuse and libfuse.		
Rationale: The FUSE community is centred around Linux, but there is also significant use of FUSE on Mac OS X.		
Originator: Student & Project Supervisor		
Dependencies: None.		**Importance:** High

Requirement #26	**Requirement Type:** Non-functional domain requirement	**History:** Created 2013/09/27
Description: The tool should not interfere with the developer's environment when not in use.		
Rationale: Avoids any dependencies being made on the presence of the tool, as such a filesystem may fail outside of the developer's environment.		
Originator: Student		
Dependencies: None.		**Importance:** Medium

Chapter 5

Software Engineering Process

5.1 Methodology

Software development methodologies intend to ease software development, increase productivity and improve overall software quality. Some are more formal than others and describe different approaches to activities undertaken during the development process. A suitable software development model should therefore be chosen carefully to ensure that it is appropriate to the nature and scale of the project, making sure to consider factors such as the likelihood of requirements changing later in the project.

A waterfall model consists of a number of defined phases which are traversed sequentially. This has the advantage of breaking the project down into manageable chunks (requirements analysis, design, implementation, and verification) but is inflexible as it does not allow previous phases to be revisited. It was unsuitable for this project as there were initially a number of technical uncertainties which may have prompted changes to the initial requirements and overall design.

A more suitable model would have perhaps been the spiral model, which allows a number of iterations of a "spiral". A spiral shares similarities with a waterfall model as it includes planning, development and evaluation. It also includes more detail and guidance regarding the tasks undertaken during various phases of the spiral. However, this was deemed more suitable for a larger project consisting of many more components and developers.

Iterative development involves constructing small parts of a software solution, continually extending its functionality in small steps or increments. This allows issues to be uncovered early on, and is therefore more suitable for projects with uncertainty in the requirements. Although the project requirements specified three clearly defined software components, there were many dependencies between them. Implementing each component in isolation one after the other would have resulted in redundant work being done and possibly late changes to the overall design. An incremental approach would allow each component to be completed in smaller phases, reducing the chance of project failure and producing a more logical and maintainable software design.

An iterative and incremental development model was therefore followed, and involved dividing the project into a number of deliverables. These deliverables mapped naturally to the three primary objectives, so each objective was met by completing a number of iterations in that phase. This involved planning each phase in more detail and coming up with a number of requirements, and then implementing the functionality. The implementation was then evaluated, and the phase was run again if the solution was not ideal. Some phases also overlapped each other upon identification of redundant implementation tasks.

As a result of this approach, new features were usually presented at each weekly supervisor meeting and the next stage was discussed along with any potential issues. This approach also reduced the risk of project failure, as there were always completed deliverables to fall back on and we did not realise late in the project that a certain approach to a problem was not possible.

Incremental prototyping was used, as the wizard and debugger were originally produced as separate tools and eventually merged into a single tool. This made sense as it was not initially clear exactly how the user interface would be used, and what functionality all of the tools would have in common. Some throwaway prototyping was also used, because it required some experimentation to work out the technical details of how to approach problems, as a significant part of the project involved working directly with libfuse and osxfuse.

5.2 Version control

A Mercurial repository was set up before beginning development, allowing version control to be used throughout the project. Mercurial was chosen as it was already installed on all machines in the lab. Version control was invaluable as it ensured backups were kept of all prototype programs, and made it easy to rollback code when an approach to solving a problem failed. In particular, we found that even slight changes can cause very unpredictable behaviour with FUSE, so keeping an explicit record of every code change made these problems easier to debug. It was also necessary to maintain a central codebase as development was taking place on Linux lab machines and a Mac laptop, and synchronising code manually would have been time-consuming and problematic.

5.3 Testing

Static testing was done by manually reviewing a unit of code after writing it to ensure that the syntax was correct and that the data flow made sense before attempting to compile and run the program.

Dynamic testing was done by frequently running the tool on both Mac OS X and Linux throughout the project, as both platforms had their own quirks and often needed compromise to make the tool run correctly on both platforms. This was initially only done on small units of code and programs. When components were merged together dynamic testing was done at a system level more frequently, and this verified that the tool met its requirements.

Component interface testing was also done to ensure that the messages (e.g., filesystem events) passed between components were being sent and handled correctly. This focused on data values rather than the resulting actions caused by the messages, highlighting underlying problems early on in the development process. Component interface testing was approached by printing event messages before handling them and checking that the messages were well formed and contained all of the required data.

Robustness was especially important due to the fact that a wide range of errors can occur at any time in a filesystem under development, and the diagnostic tool would be expected to handle these gracefully as part of its normal operation.

5.4 Programming conventions

Good programming practices were followed throughout the project to ensure the tool was efficient, robust and maintainable. Code was written as clearly and concisely as possible, making sure to follow language guidelines when appropriate. In the case of software extension it was important to match the style of existing work to ensure a consistent codebase. Designs were also kept logical, and in general a number of approaches were investigated before attempting to implement a solution to a problem.

5.5 Debugging

As most of the code for the project was written in C, extra care needed to be taken when calling library functions as C does not provide exception handling. Ensuring all return values are checked and errors printed to the console eased debugging significantly. Some parts of the project were also very slow to compile (i.e., osxfuse) so it was good practice to write debuggable code throughout the project rather than just as an error occurred.

Chapter 6

Ethics

The project involved human participants for usability testing, which consisted of following a set of instructions and evaluating the final software solution. This was presented as a questionnaire which participants were asked to fill in while using the software, and is included in Appendix B. Participants explicitly agreed to take part in the evaluation, and the results were stored and referred to anonymously.

Care was taken to ensure that the evaluation was a positive experience, and did not involve "destructive testing". The tool was also purely software based, and did not require interaction with hardware that required additional mobility or used sensory deprivation. The project therefore had no consequences for human welfare.

A preliminary ethics self-assessment form was completed at the start of the project.

Chapter 7

Design

Strong software design is important as it ensures the application is consistent from a user perspective, avoids duplication of functionality, and results in a more maintainable solution in the long run. It also ensures that all requirements are more likely to be met as it considers each one from a high level before becoming concerned with the implementation details of each one. This chapter covers the structure of the tool, discussing design decisions, challenges and justifications for unusual design features.

7.1 Overall architecture

The main components of the tool consist of the frontend, the modified user-level FUSE library and the filesystem being diagnosed. Our modified FUSE library has four modes of execution: wizard mode, test suite mode, debugging mode and normal mode. The mode indicates how the library should handle an attempted mount of the filesystem, and it performs a normal mount by default. A normal mount simply mounts the filesystem as the original FUSE library would and no diagnostic tool operations are run from the library. The frontend is also subdivided into wizard, test suite and debugger, with some degree of commonality between them. The provided filesystem is treated as a black box, and the rest of the system can only interact with it via its declared FUSE functions.

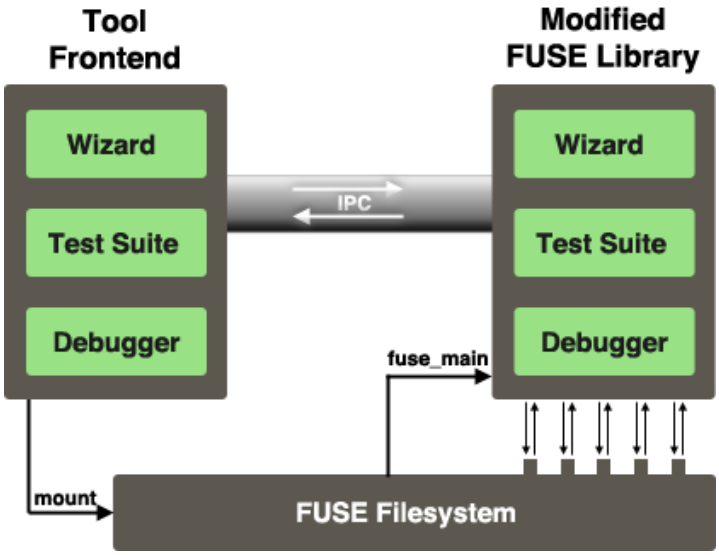

Figure 7.1: Main components of the system showing the frontend of the diagnostic tool, the modified FUSE library, and the FUSE filesystem being diagnosed.

7.2 Application

The tool frontend is in the form of a single GUI allowing use of any of the three tools. It is intended to be straightforward to use and primarily aimed at developers working on their own filesystem. It is also appropriate for developers extending or debugging filesystems written by others. We initially considered producing separate interfaces for each tool as they display different information, but it soon became clear that all three tools required widgets to select and control execution of a filesystem.

Visually, the tool is divided into a filesystem input section and a set of three tabs. The input section allows the user to specify the location of their filesystem binary and pass the required command line flags, as these are generally different for all filesystems. Each tab allows the user to use the wizard, test suite or debugger and the buttons above the tabs allow the current tool to be started or stopped. Only one tool can be used at a time to prevent confusion over which tools are actually running.

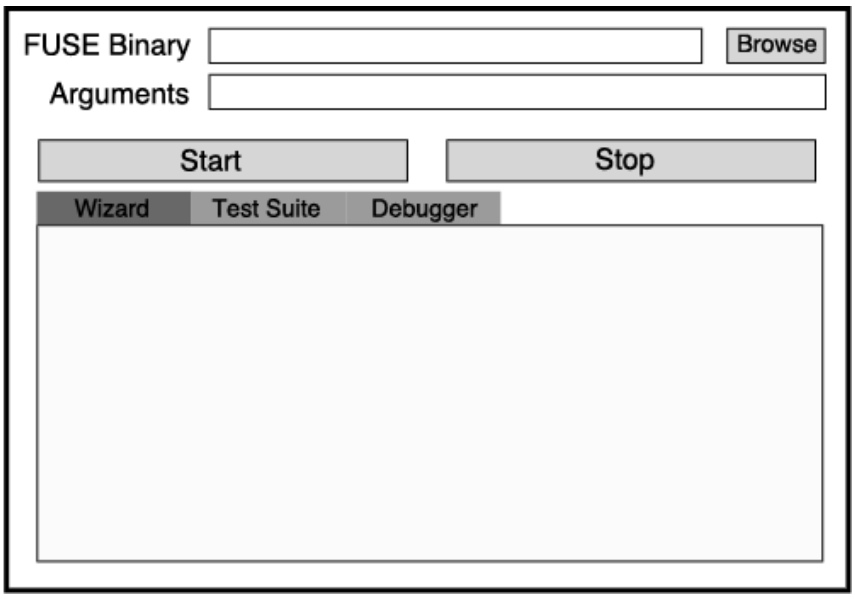

Figure 7.2: Overall design of the GUI, showing the filesystem input section and control buttons, along with the tabs for each tool. The following diagrams illustrate the contents of each tab.

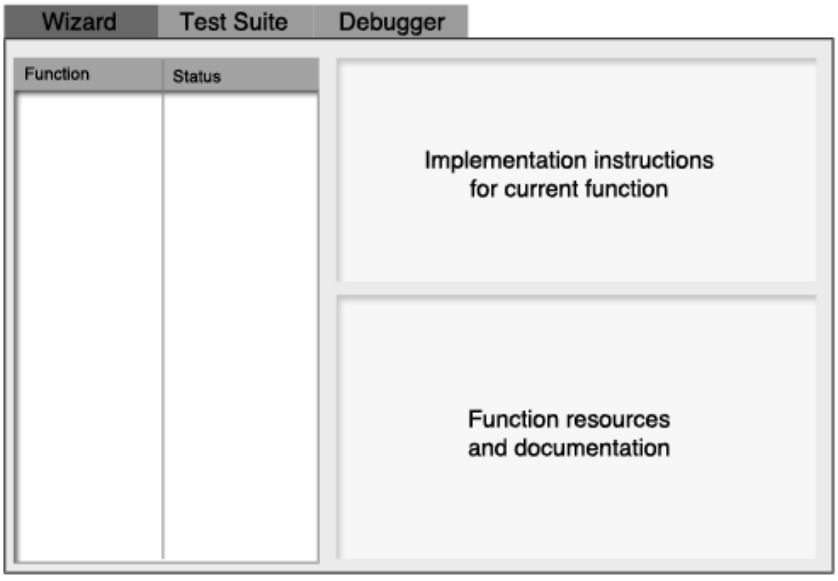

Figure 7.3: Contents of the Wizard tab. The table lists a set of functions and becomes longer as more functions are implemented correctly. The status indicates the level to which a function is completed.

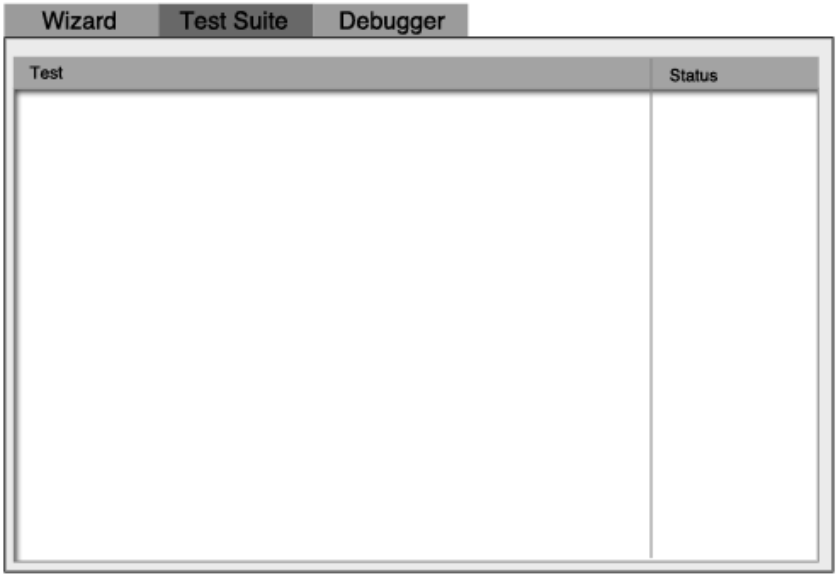

Figure 7.4: Contents of the Test Suite tab. This displays test groups that are broken down into sequences, which can then be broken down into individual calls. The status indicates whether or not that function, sequence, or entire group passed the tests.

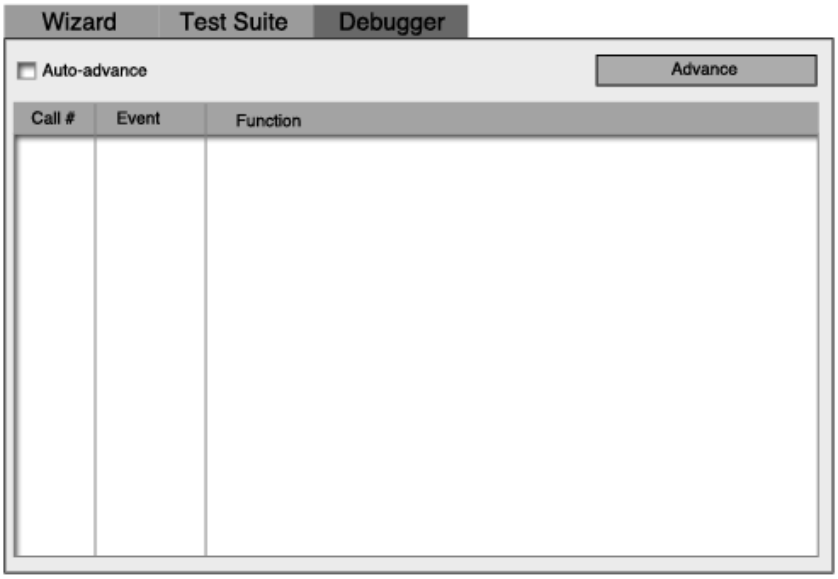

Figure 7.5: Contents of the Debugger tab. This displays events in the table when FUSE functions are invoked and just before they return. Functions with parameters can be expanded to see the parameter values. The Advance button allows the user to manually step through execution of the filesystem. Ticking the Auto-advance checkbox turns off this feature and advances functions automatically.

7.3 Instrumentation

A major requirement was to allow the user to observe FUSE call invocations, which would involve firing events when functions are invoked and when they return. Parameter values needed to be captured after invocation and before returning (as many functions return values through pointers in the parameters), along with the final return value of the function.

Upon investigation into the FUSE design it was clear that we needed to override or extend part of the existing FUSE system to allow interaction with a filesystem without modifying the filesystem itself. The following diagram, based on the diagram at http://fuse.sourceforge.net/ for the original FUSE structure, shows which component of the FUSE system we replaced:

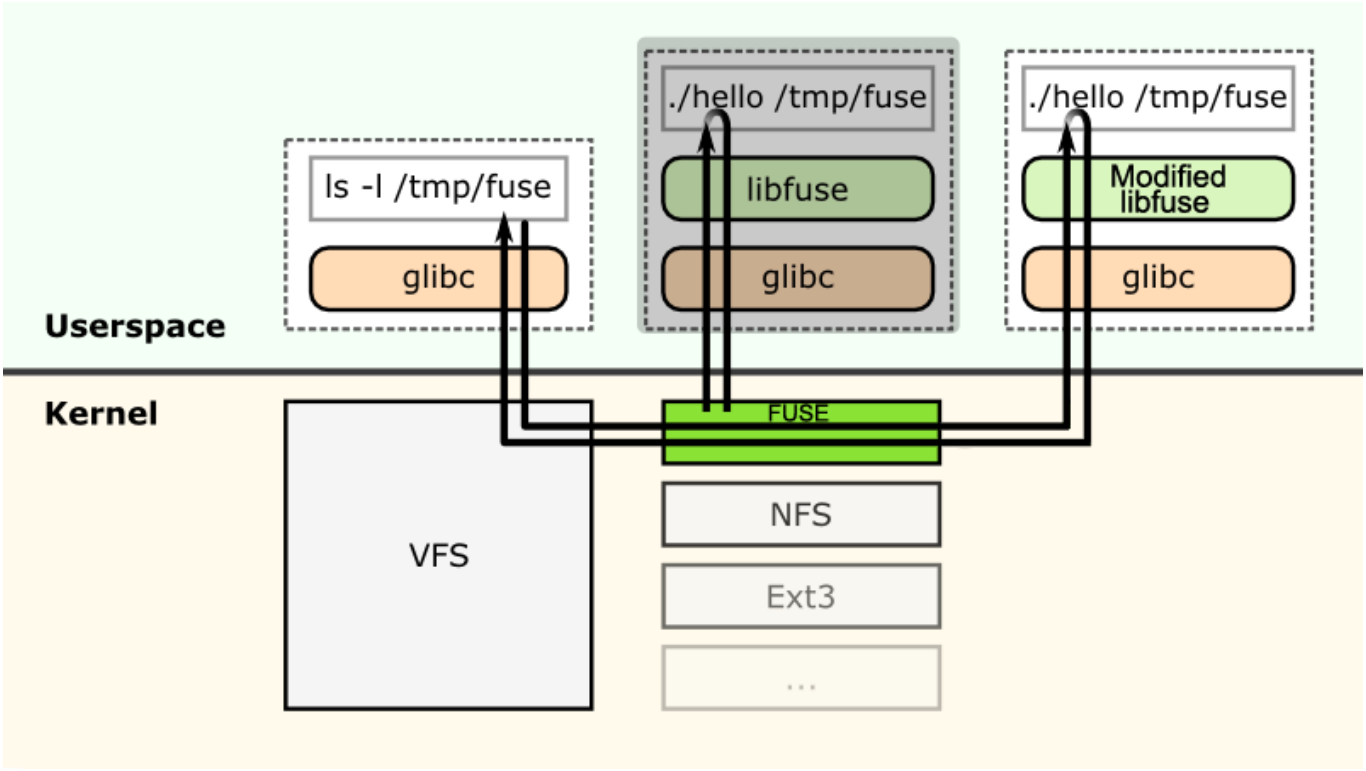

Figure 7.6: The greyed out box represents the standard usage of FUSE, and the box to the right represents our new usage of FUSE using a modified library. Based on diagram from the FUSE Wikipedia page (Wikipedia, 2008), which was based on the original diagram from the FUSE Project page (Szeredi & Henk, 2004).

The userspace library was replaced rather than the kernel module as this would avoid requiring the user to install the diagnostic tool, and working at the kernel level would not have offered us any advantages. The FUSE kernel module also uses low level filesystem operations rather than the operations defined by the FUSE API. Most of the low level operations are similar to the API operations, but there are some differences (i.e., additional functions, missing functions and different parameters) which could cause confusion and would be less useful to the developer. It is also possible to dynamically instruct a program to load a different implementation of a shared userspace library at runtime, allowing the tool to be used without permanently modifying the user's environment. This approach also allows non-root users to use the tool as it requires no installation of any libraries or kernel modules.

We considered implementing the modified FUSE library as a lightweight wrapper that interacts with the original userspace library, as this would avoid duplicating the entire library. However, upon investigation we found that we could only override methods exposed by the library (such as fuse_main) and not internal code (Laurikari, 2009). This would allow us to override the libfuse entry-point but would then require complete re-implementation of internal functions and data structures. We therefore decided to clone and modify the latest version of libfuse (and osxfuse on Mac), keeping the modifications minimal and fairly localised to allow easy maintenance if, for example, a developer wished to extend the concept to a later version of libfuse.

7.4 Simulated invocation

Another major part of the requirements was the ability to invoke FUSE functions implemented by a given filesystem, as this would allow a development wizard and test suite to be implemented. The only method of interacting with a filesystem was to mount it and make calls against the VFS layer using the C POSIX library or via Unix programs like `ls`. However, this approach would not provide the developer with enough fine-grained detail and many VFS functions do not translate directly to FUSE functions. Mounting the filesystem is also not ideal as it allows the operating system to begin setting up system files like Trash directories and folder attribute files. While this is acceptable (and intended) behaviour for a debugger, tests should be run in a sandbox environment free from external interference.

We therefore needed a method of invoking FUSE functions directly without going through the VFS layer, and without providing the filesystem with an actual mountpoint. One could envision a seemingly simple approach of parsing the filesystem binary to find and invoke functions, but this would be highly platform dependent and would require re-compilation of the filesystem with debug symbols enabled, missing the requirement of being able to use any FUSE binary without re-compiling or instrumenting it.

The approach we took involved extending the modified FUSE library introduced in the previous section, allowing it to intercept the initial call that the filesystem makes to hand over control to FUSE. This provided direct access to the FUSE functions as defined by the filesystem developer and allowed us to control whether or not the filesystem actually mounts. The filesystem is therefore run in the usual way from the user's perspective but libfuse executes the test suite on the provided functions. It can then terminate without actually mounting the filesystem.

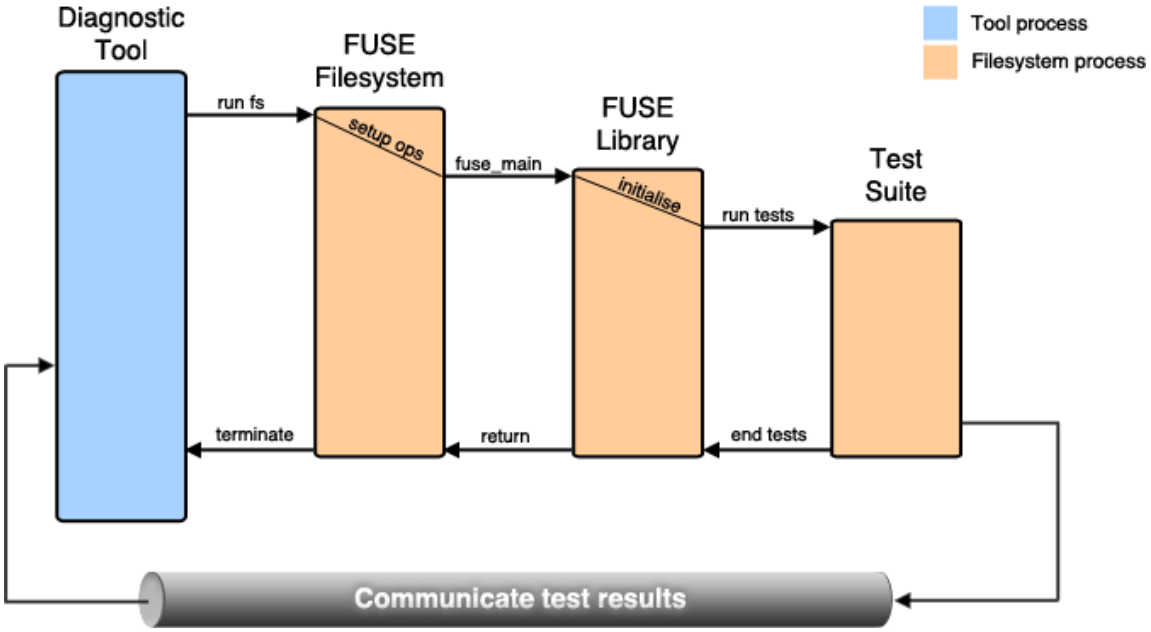

Figure 7.7: The diagnostic tool runs the filesystem and dynamically links the modified FUSE library so that the test suite is run when fuse_main is invoked. Results are returned directly to the diagnostic tool process as this avoids losing information in the event of a filesystem crash.

7.5 Inter-process communication

Many of the project requirements also relied on running multiple processes simultaneously and communicating data between them. FUSE filesystems are standalone binaries that must run as independent processes, and therefore cannot run in the same address space as the diagnostic tool itself. It was therefore important that we decided upon a standard mechanism for inter-process communication (IPC). The mechanism would be used primarily by the diagnostic tool and the FUSE library, so needed to be lightweight and simple to use from C. It also needed to be efficient and robust, as a large amount of data needed to be communicated between the running filesystem and the diagnostic tool, and the filesystem could behave unexpectedly at any time.

This data would include test results and details of FUSE call invocations (including their parameters). It would also consist of other information, such as which tests to skip and the location of the file to load test data from. The data is therefore very structured and can be conceptually broken down into individual messages.

The following table highlights the benefits and drawbacks of using certain IPC mechanisms to communicate diagnostic events:

IPC Mechanism	Benefits	Drawbacks
Files	– Simple to open, read and write	– Do not suit the event-driven nature of the information – Objects need to be serialised
Shared memory	– Efficient transfer of data – Objects do not need to be serialised	– Not robust as errors can propagate from the filesystem process
Sockets	– Suitable for event-driven information	– Communicating port numbers initially could be unreliable – Inefficient to use a full network stack for simple IPC on the same machine – Objects need to be serialised
Message-passing system	– More suitable for event-driven information as it would pack each event into a message	– Limited choices of systems for C – More suitable for larger systems – Objects need to be serialised
Named pipes	– Suitable for event-driven information – Simple to open, read and write	– Objects need to be serialised

Named pipes allow simple text-based communication between processes using a special file that works on both Mac OS X and Linux. They are also known as FIFOs, as the first data chunk written to the pipe is the first data read out. For our use case, these have the same benefits as sockets only without the unnecessary complexity. We therefore chose to use this as the base communication layer, and achieved message-passing using JSON-formatted chunks. This mechanism is discussed in more detail in the Implementation chapter.

7.6 Controlling execution

One of the requirements of the debugger was that it must allow execution of the filesystem to be manually paused and advanced. For example, if FUSE tries to invoke the `getattr` function, the user should have the ability to control when the function actually executes. Users do not need to be able to control execution of code within a function as this would be a task for a general purpose debugger and introduce a range of other concerns and challenges such as the use of breakpoints.

It was therefore necessary to introduce another IPC mechanism into the system which would be more suitable for sending a "signal" rather than structured data. A clear choice was the named semaphore, and this allowed the user to indicate when a filesystem is permitted to advance to the next function. Processes attach to it by opening a file so this provides similar advantages to using the named pipe for data transmission.

The following diagram illustrates the design of the debugger. It has a similar design to the test suite and uses the same components, but the difference is that it receives real calls via the FUSE kernel module instead of making them. It intercepts these calls, encapsulates the call data (including function name, parameters and return value) and transmits it back to the diagnostic tool. Execution is controlled by waiting on the semaphore when intercepting a call.

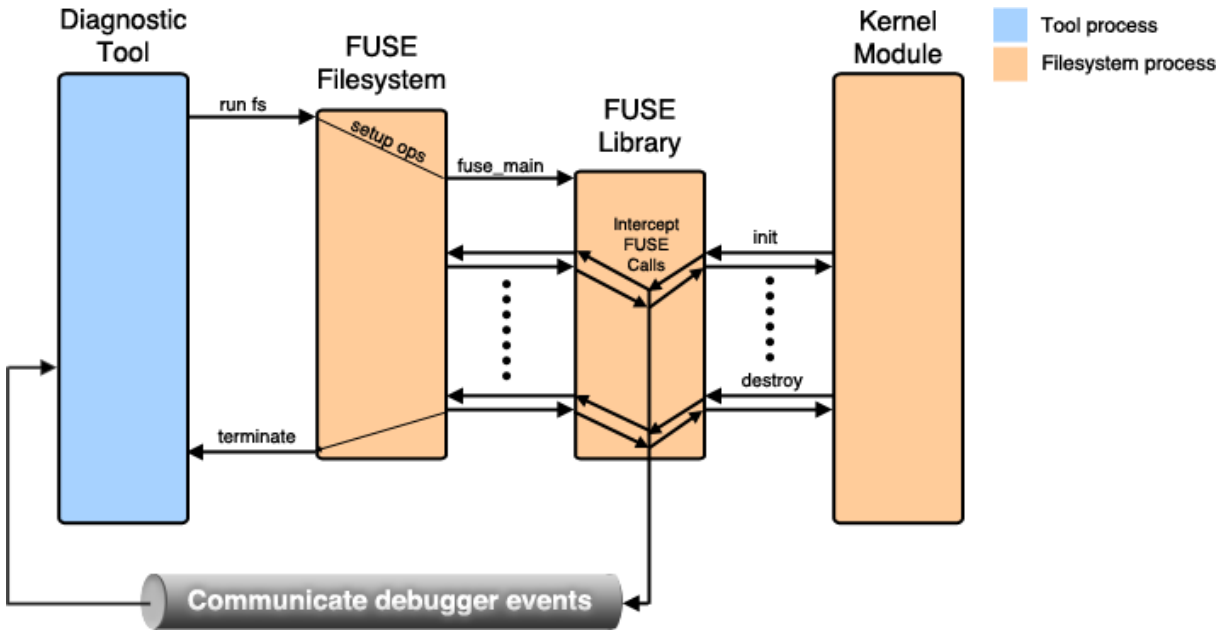

Figure 7.8: Design of the debugger with respect to the rest of the system.

Chapter 8

Implementation

This chapter discusses the implementation of the diagnostic tool, focusing on mechanisms and data structures. We explain some of the implementation decisions that were taken and alternative approaches that were considered. It begins by describing the overall implementation of the application and the GUI, and goes on to explain the inner workings of each part of the tool. It also explains our modification and use of the FUSE user-space library in more detail, including inter-process communication. We conclude by addressing issues of compatibility and platform-independence and how we attempted to solve them.

8.1 Application

The frontend to the tool is implemented in `fdt.c`, with code specific to each part of the tool in `wizard.c`, `testsuite.c` and `debugger.c`. The tool for logging test data can be found in `logger.c`, and this is also part of the main tool (although it is run via a command line flag, and is not part of the GUI). Each of these files also has a C header file in the same directory. As the next section will discuss in more detail, `fdt.c` is responsible for setting up most of the GUI and the other files set up their part of the GUI and perform any updates to it.

A filesystem is run with a given tool by invoking `startTool` with the filesystem path, arguments, and a pointer to the function that runs the frontend part of the tool. The tool mounts the filesystem by forking the process and calling `execve`, and it keeps track of the filesystem process ID. It simultaneously invokes the tool handler function which starts a new thread that handles communication with the filesystem. The tool catches `SIGCHLD` signals (sent when child processes terminate) which allows the tool to detect when a filesystem has crashed. It also allocates shared memory which is used to store an error message if the filesystem fails to execute. This level of robustness is necessary as the user may want to diagnose a crashing filesystem.

8.2 GUI

The GUI was produced using GTK+ (the GIMP Toolkit) which is designed to be a multi-platform toolkit for GUI development, supported on both Mac OS X and Linux. GTK+ is natural when used on GNOME on Linux, but it did not translate perfectly to the native widgets on Mac OS X. We still chose to use GTK+ instead of native toolkits (such as Cocoa) as we deemed it necessary to maintain central codebases wherever possible. This would speed up development, reduce future maintenance effort and also make the tool less prone to bugs present on only certain platforms.

Setting up an up-to-date and integrated GTK+ environment on Mac OS X was difficult as the versions in MacPorts were outdated and used X11. Building GTK+ manually would have been very difficult as there are many dependencies on packages generally only present on Linux distributions. We therefore used an installation script (GTK-OSX) to install a program called `jhbuild` that managed these dependencies and built the GTK+ environment. `jhbuild run` is therefore used frequently in the main Makefile, as this links the correct GTK+ libraries and C header files.

The GUI is constructed of `GtkWidgets` which are set up in in the `showGUI` function found in `fdt.c`. This involves creating a main widget for the window, widgets for the filesystem selection and control part of the GUI, and then a `GtkNotebook` widget to contain the tabs. This then calls `createWizardTab`, `createTestsuiteTab` and `createDebuggerTab` to set up the default structure of the widgets inside each tab.

We considered displaying HTML-formatted documentation in the wizard tool so that users could read the most relevant part of the function documentation without having to view the link in their browser. GtkHTML is a HTML layout engine for GTK+ but its use is now officially discouraged by GNOME (2012), and it would not compile with our version of GTK+. WebKitGtk+ is a more modern layout engine but this would not compile on our version of Mac OS X due to missing Linux-only dependencies. As this would only have been an enhancement, not a project requirement, we decided not to implement this functionality.

The tool is sent many events (e.g., debugger events) by other processes via IPC, using threads to continually receive them. Unfortunately GTK+ is not thread safe so required the use of internal event queues and locks to ensure GUI changes only took place from the main GTK thread. This was implemented in a function `gui_idle` attached to the main GTK+ loop using `g_idle_add`, which ensures the function is called continually unless the GUI has higher priority tasks to complete. `gui_idle` delegates control to the wizard, test suite and debugger in turn as they each have their own queues of changes that need to be made to their tab in the GUI.

8.3 FUSE user-space library

As discussed in the design chapter, we needed to modify the user-space FUSE library to allow us to interact with filesystems. This involved modification of the latest versions of libfuse and osxfuse, compilation of the library, and finally a method of dynamically linking the library when a filesystem is run.

8.3.1 Compilation

Building the userspace FUSE library involves compiling and linking the entire codebase, and finally outputting a shared library file (.so on Linux and .dylib on Mac OS X). Fortunately both libfuse and osxfuse are distributed with makefiles and instructions on how to use them. Difficulties arose when we needed to modify the build process to add new dependencies (cJSON for IPC, and GLib for data structures). The build processes for libfuse and osxfuse were not well documented, and configuration was distributed between a number of makefiles, automake scripts and automake configuration. We eventually managed to add these dependencies but it required a significant amount of trial and error.

Libfuse can be built by running `make` in the libfuse directory, and osxfuse can be built by running `sudo ./build.sh -t dist` in the osxfuse directory. Compilation on Mac is slow (takes around 3 minutes on a Late 2008 MacBook), and while running a partial build speeds up the process the libraries produced are insufficient for use by the diagnostic tool.

As we intended to implement very similar functionality in both libfuse and osxfuse it became necessary to centralise shared code to avoid maintenance of multiple codebases. It was therefore necessary to frequently test the tool on Mac and Linux as the C code would often compile on one platform but not the other, and require a compromise or OS-specific branches. The files included by the library are `testsuite_run.c` and `wizard_tests.c` and are included from the main parent directory by using symbolic links. On Mac, osxfuse uses hard links because the build tool does not follow soft links correctly. Hard links are usually lost when moving files to a different system, so the build script re-creates these links before running.

8.3.2 Dynamic linking

We needed a method of forcing a filesystem to use our modified FUSE library rather than the real FUSE library, as this was a fundamental requirement for the success of the tool. This ideally needed to be done at runtime without recompiling or relinking the filesystem program, and it needed to work on both Mac OS X and Linux. We also wanted to avoid modification of the user's environment, so full replacement of the installed library was not an option.

One approach was to invoke the program loader directly, passing the path of the library and the path of the filesystem executable:

```
/lib/ld-linux.so.2 --library-path <libpath> <filesystem> <args>
```

The loader would dynamically link the FUSE library in the given path rather than the library in the default installation path. However, no equivalent program could be found on Mac OS X.

The other approach was to set the environment variable `LD_LIBRARY_PATH`. This is a list of paths that will be checked for shared libraries before the default paths, and is generally used for development and testing. Mac OS X also uses a similar variable, `DYLD_LIBRARY_PATH`, which achieves the same thing. The diagnostic tool therefore locates the newly-compiled versions of libfuse and osxfuse and sets the environment variables to these paths when executing the filesystem. This is a purely temporary change as the environment variables are not set globally for the user, so their environment remains unchanged.

8.4 Inter-process communication

8.4.1 Pipes

As discussed in the design section, named pipes are used as the main IPC mechanism. Messages are passed by structuring them as JSON objects, using the cJSON library as this is a natural structure for transmitting individual values, arrays and objects (i.e., C structs). JSON allows significant flexibility when constructing and parsing messages as they can be structured dynamically. This is required when passing FUSE call data as most FUSE functions take different numbers of parameters of different types. Using JSON also makes the system more robust and maintainable as messages can easily be extended without breaking existing code.

The diagnostic tool process is responsible for creating the pipe using `mkfifo` and the filesystem process (via libfuse) opens it for reading using `fopen`. As the tool process forks and executes the filesystem before continuing with the initial setup, it is possible for a race condition to occur when opening the pipe. The filesystem process therefore blocks until the pipe is created, and will close the pipe itself once it finishes sending data. It will also send an `__END` event to explicitly indicate termination, as this helps the diagnostic tool detect when a filesystem crashes unexpectedly. The pipe is initially located using a filename pre-defined in each tool in both processes. This is the most simple and least error-prone method of bootstrapping the communication, but does have the limitation of not being able to run two instances of the same tool simultaneously. If this were a requirement in the future then it could be addressed by passing a randomly generated filename to the filesystem process via an environment variable.

8.4.2 Environment variables

Sometimes it is not appropriate to set up a full communication channel to transmit small amounts of information, as named pipes do have some overhead and it results in a more complex implementation. Environment variables are a simpler method of communicating information to a child process and are useful when there is a need to initially transmit a small set of data to a filesystem. This was done in the wizard tool to transmit the identifiers of tests that should be skipped, as these did not change dynamically during the execution of the filesystem.

Environment variables are also used to instruct the FUSE library how to mount the filesystem. This is done by setting `FDT_TOOL` to one of the three tool names. This allows the filesystem to stay unmounted for the wizard and test suite tools, and allows interception and reporting of real call data for the debugger tool.

8.4.3 Semaphores

The debugger uses a named semaphore to control execution of functions. This works in the same way as a standard semaphore, blocking on `wait` when the semaphore is zero and using `post` to unblock a `wait`. A file for the named semaphore is created by the debugger tool process, using `sem_open`, and opened by the filesystem process once it exists.

A message-passing mechanism would rely on "spinning", in which the filesystem process would repeatedly check if it can continue execution. Using semaphores is therefore cleaner and more efficient than message-passing.

8.5 Wizard

The frontend of the wizard is implemented in `wizard.c`. When the start button is pressed with the wizard tab open, the `startWizard` method is invoked. This starts a new thread, executing `doStartWizard` which receives wizard events from the filesystem process.

8.5.1 Test execution

Instead of actually mounting the filesystem, the modified FUSE library runs a number of tests against the filesystem by calling the user-implemented functions directly, using the tests written in `wizard_tests.c`. Results are communicated back to the tool frontend as tests execute, and if a test fails it includes information formatted as user-friendly feedback instructing the user how to address the problem. The structure of a message includes the function name, the test number (as a function may have many tests for certain parts of the functionality), a boolean value set to true if the test passed, another boolean value set to true if the test is optional, and finally the message itself if there are any instructions for the user to read.

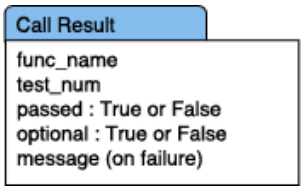

Figure 8.1: Structure of message used to communicate the result of a wizard test.

The tests are executed in priority order, so the wizard stops executing once a test fails so that the user is informed about the first increment of functionality that they need to implement. Some functions cannot be implemented in isolation without considering the other, and the wizard will jump between the functions during development to allow a non-linear approach.

An example is the implementation of `mkdir` and `rmdir`. The first tests ensure that `rmdir` is defined and that it returns the correct error values when trying to remove a nonexistent directory. The next tests ensure that `mkdir` is defined and use it to create a new directory, checking that the call succeeds. It then makes another call to `mkdir` to ensure the same directory cannot be created again, as it already exists and the filesystem should have persisted this and returned the correct error. As the wizard now has higher confidence that `mkdir` works it can return to testing `rmdir` by trying to delete the newly created directory.

8.5.2 Frontend

The GUI displays a list of functions which aggregates the results of the test runs for each function, and determines whether it is "Implemented", "Skipped", "Defined" or "Undefined":

Function Status	Interpretation
Implemented	Function determined to be working correctly.
Skipped	Implemented, but contains tests that have been skipped by the user.
Defined	Function defined and added to the operations struct correctly, but the implementation is faulty.
Undefined	Function is not defined or has not been added to the operations struct correctly.

When a function is detected as having an incomplete or missing definition, the wizard displays the human-readable instruction along with general information about the function. This includes the function signature (i.e., the types of the expected parameters and return value). This is achieved by reading a file generated by a Python script (`siggen.py`) by reading the FUSE library files. This allows the function signatures to be kept up to date with the version of FUSE that the user is using, rather than storing them in a manually-constructed file or relying on Internet resources.

The wizard also displays links to documentation for the function under development, and loads these from the `resources.json` file. This file was constructed manually as there is no clear automated way to map every FUSE function to its POSIX function. Some of these functions are also FUSE-specific and have no POSIX documentation, so the best source of documentation for these is the official FUSE website.

Tests are skipped from the frontend by pressing the Skip button when an instruction is displayed. This skips the last failing test for the function by adding its name and ID to a list that is sent to the wizard backend on the next run. The backend interprets these identifiers and does not run these tests again, instead returning a test-skipped event to acknowledge the reason it was not run.

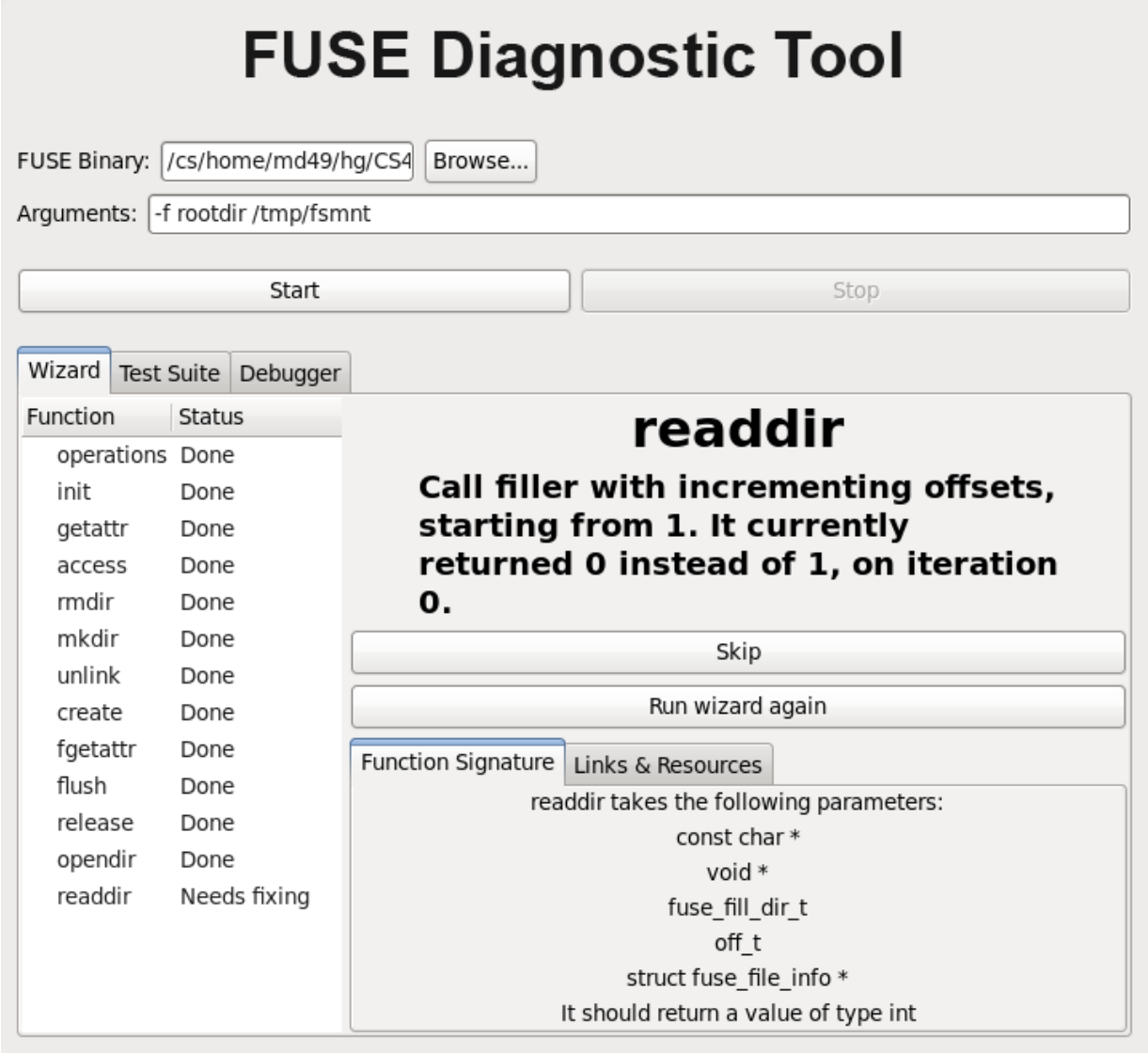

Figure 8.2: Screenshot of the wizard tool, showing a failed test and some instructions.

8.6 Debugger

The frontend of the debugger is implemented in `debugger.c` and begins execution in the same way as the wizard, only passing a different value in the `FDT_TOOL` environment variable.

8.6.1 Call interception

Calls are intercepted in the FUSE library code, which can be found at `lib/fuse.c` in both osxfuse and libfuse. The `fuse_fs` struct contains pointers to the functions defined in the user's filesystem, but it was extended to also include pointers to wrappers around these functions. All calls are therefore made against a wrapper function instead of the real function which allowed us to intercept them. The wrapper code first generates a unique sequence number for the function call, constructs a JSON object for the parameters passed to the function, and then reports this to the tool process. It then makes the real function call, placing the return value and any modified values into a new JSON object, reporting this to the tool to indicate completion of the function.

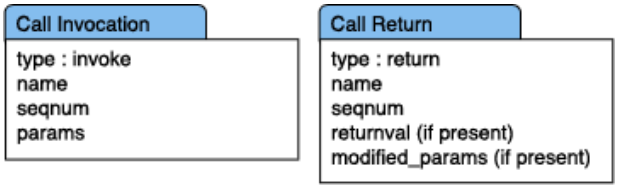

Figure 8.3: Structure of messages that indicate a function invocation or return.

The wrapper functions also wait on the semaphore before continuing, as this ensures the tool retains control over which functions to advance. The tool is responsible for incrementing (posting) the semaphore on every function call, and this can be done manually by the user pressing the Advance button or automatically when Auto-advance mode is on.

8.6.2 Frontend

As with the wizard, the debugger starts a new thread to receive events as JSON-formatted messages. These are passed to `handleDebuggerEvent` which adds it to the GUI queue so that it can be added to the event table by `gui_idle_debugger`. This decomposes the JSON objects to display the function name, event type (return or invocation) and return value if present. It also recurses to display nested parameters in a tree widget.

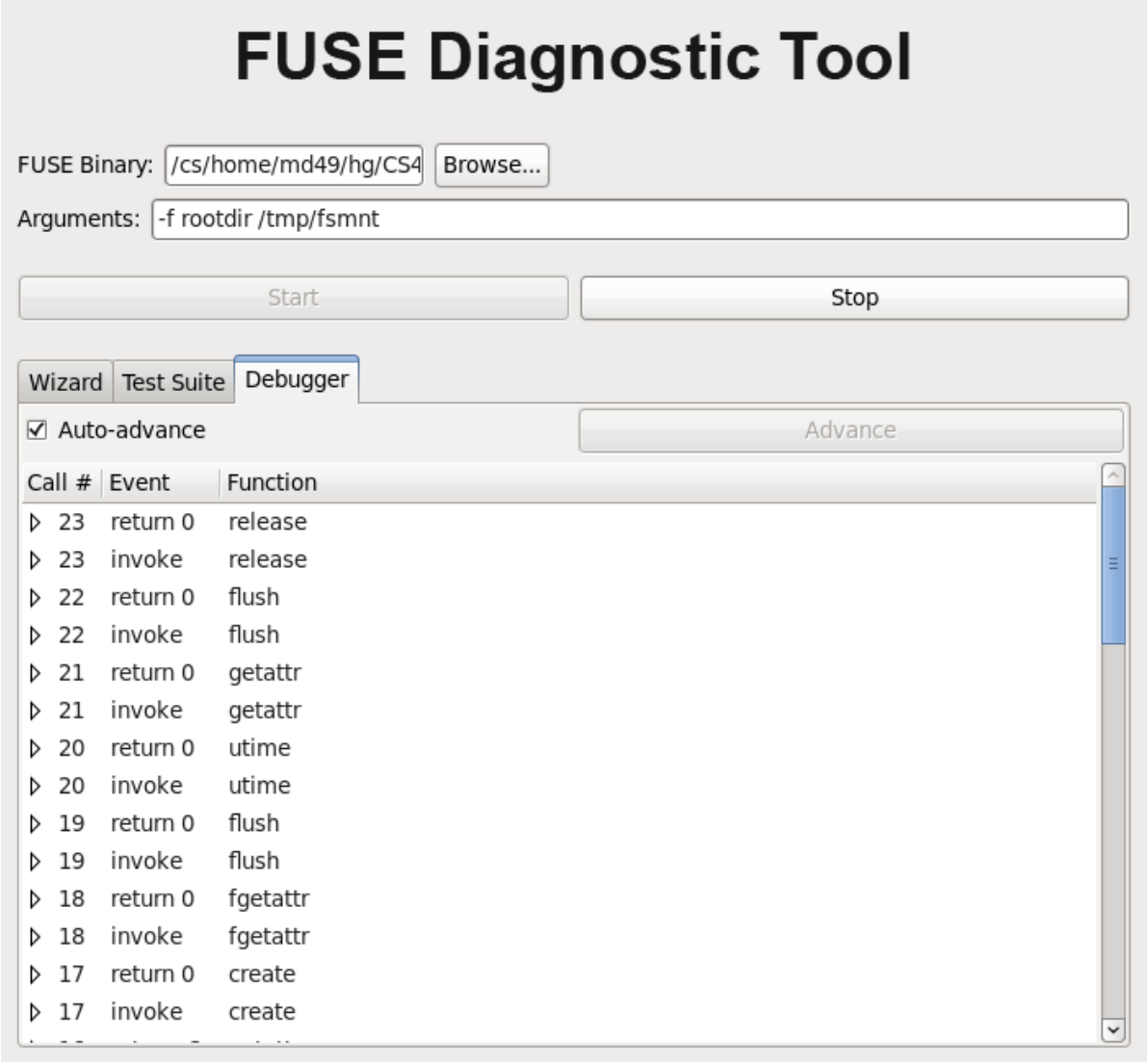

Figure 8.4: Screenshot of debugger tool, displaying a number of calls and returns.

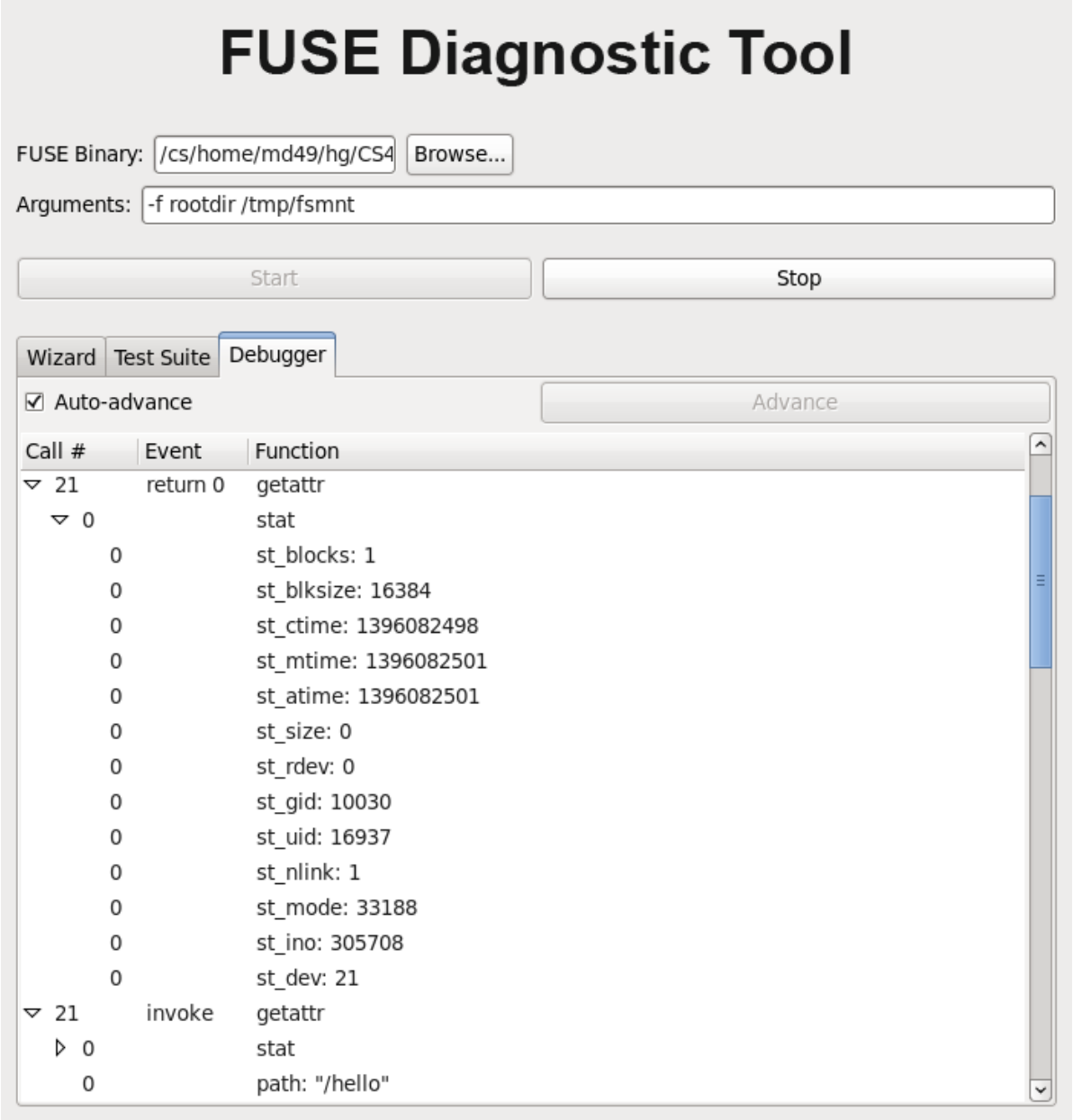

Figure 8.5: Expanded view of the modified parameter values in a getattr call.

8.7 Test suite

8.7.1 Logging test data

A tool was produced as part of the test suite that allows the user to generate extensive test sequences by logging real operation of the filesystem. This is a command-line utility operated by passing the `--logger` flag to `./fdt` and is implemented in `logger.c`. It extends the functionality already present in the debugger and reads events from the existing pipe to record operation of the filesystem.

The logger runs interactively, displaying logged data and prompting the user when they may wish to make a decision. Once a sequence has been logged the tool prompts the user for information about the task they were performing and the application they used, and logs this along with their operating system name and version. At the call level, it logs the function name and the values of the parameters passed. It allows the user to record multiple call sequences that form a group and this group can be appended to a test data file which may already contain groups. The following diagram shows how the tool is operated:

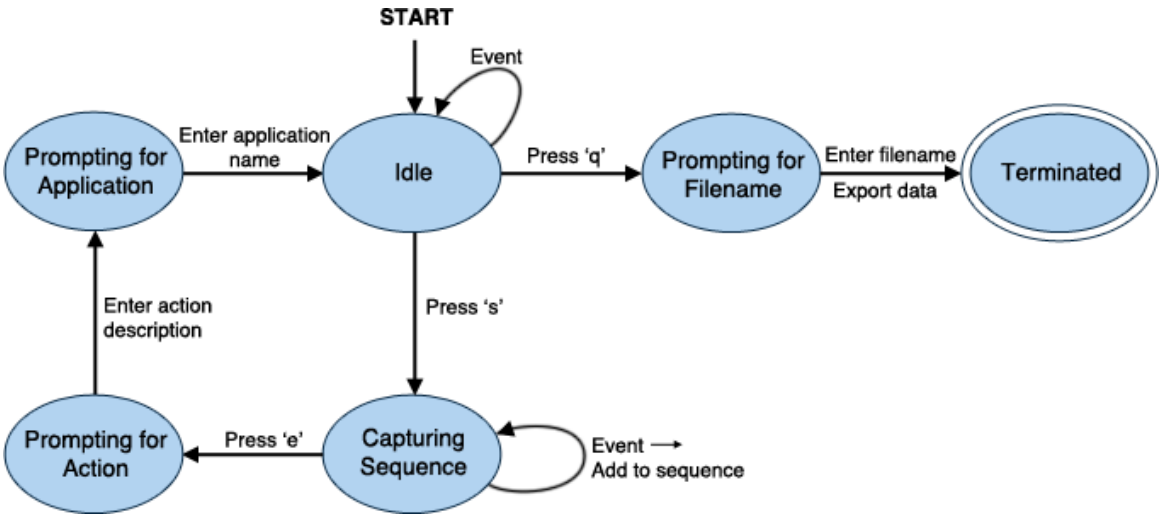

Figure 8.6: State diagram illustrating operation of the logger tool.

8.7.2 Test execution

The test suite executes tests in the order they are defined in the test data file, keeping the structure of groups, sequences and individual calls.

The test suite reports each event to the tool, including the start and end events for each group and sequence, and the results of individual calls. The follow diagram shows the data contained in these events.

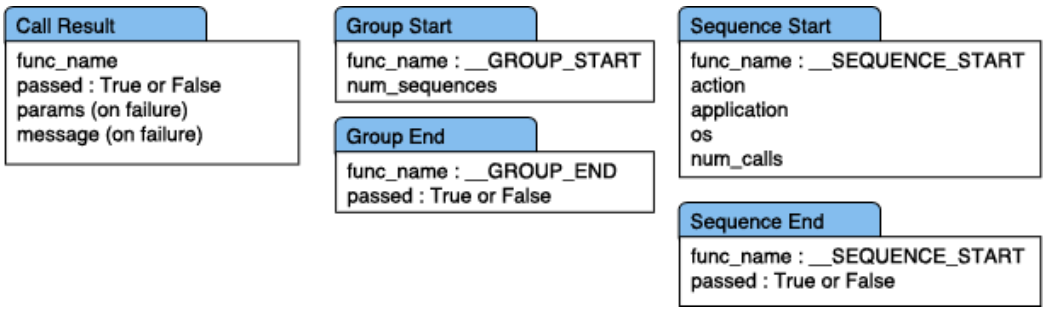

Figure 8.7: Structure of test suite messages, reporting the results and metadata for entire groups, sequences and individual function calls.

Each test call is made against a passthru filesystem and then against the filesystem under test, using the parameter values logged in the test data file. The return values of the two calls are compared, making sure that either both functions failed or both functions passed (ignoring the specific value, as this is usually not required by POSIX specifications). The passthru filesystems used are Loopback for Mac OS X (Fleischer, 2014) and Big Brother for Linux (Pfeiffer, 2012). These were linked into the test suite code directly instead of actually mounting them, as mounting two filesystems and communicating between them would introduce significant complexity. This approach is also more natural, as the test suite invokes functions by calling the defined FUSE operations directly and does not actually mount the filesystem.

Checking return values is not always enough to ensure that the filesystem is working correctly, as there is likely to be more state that is not fully tested by the test data. The suite therefore also checks that the contents of the two filesystems are equivalent by checking they have the same files and directories, and that the files have the same properties. This is done in `assert_equivalent` and is called after every pair of FUSE operations.

After executing a group of sequences the suite will clear both filesystems to reset their state, as each group must be started on an empty filesystem. It does this by calling FUSE functions directly to recursively retrieve directory contents and empty them, done by the `reset_filesystem` function.

As pointed out in Requirement #23 (subsection 4.4.3 of the requirements document), the call logger for the test suite will log pointers as numeric memory addresses if it cannot serialise the data they point to. The test suite should therefore discard pointers from test data and reallocate them at runtime. These include empty buffers (the sizes of which are stored in parameter values) and function pointers such as the `readdir_filler` function which would normally be implemented by the FUSE library. Calling code also normally keeps track of filehandles (integers) passed by the filesystem. The test suite therefore keeps a map of filenames to filehandles, making sure to pass filehandles to functions that may require them.

8.7.3 Frontend

From a technical perspective the test suite GUI works similarly to the wizard and the debugger. It displays test results in the order that they are executed, nesting them in groups and sequences. It displays a pass or fail message next to each group, sequence and individual call, and nests the error message for a failing call. Most failures will also include a copy of the parameters passed to the function as this will help the developer find the root cause of the problem.

Figure 8.8: Screenshot of the test suite, showing a group of call sequences and a failing test.

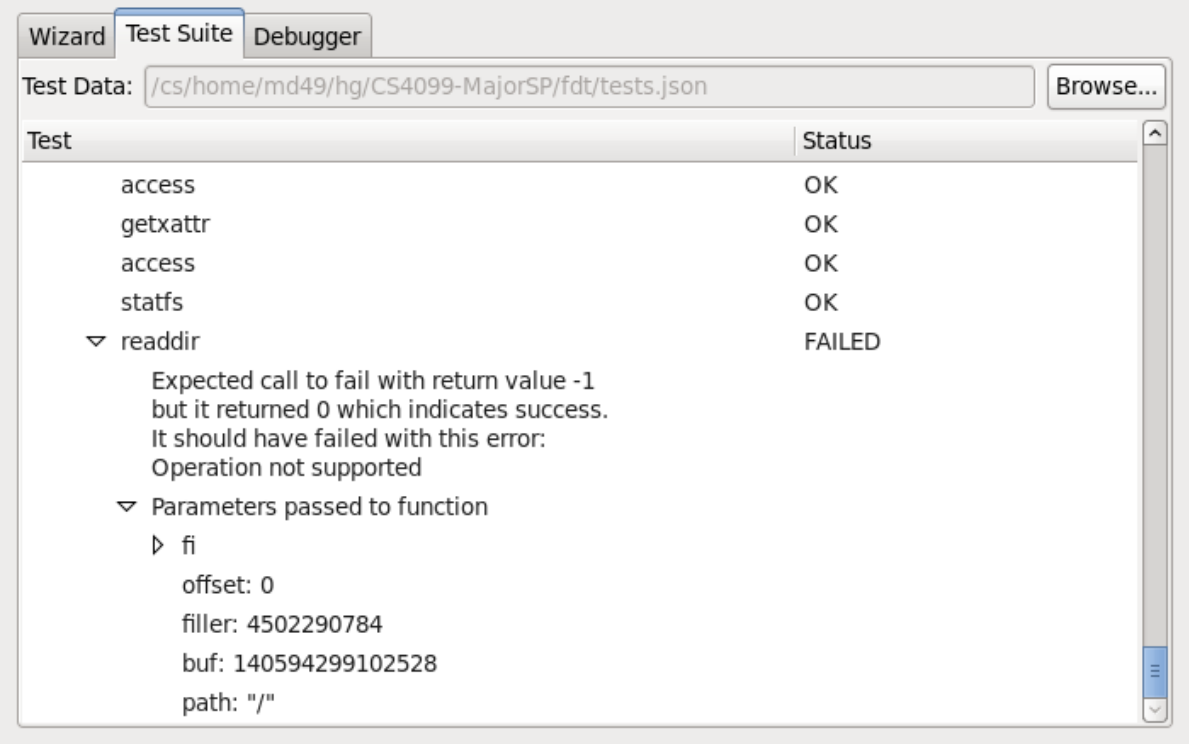

Figure 8.9: Expanded view of a failing test in the test suite.

8.8 Platform compatibility

There are many differences between the Mac and Linux versions of FUSE, but many more similarities. We therefore decided to share most of the code between Mac and Linux, using macros to produce platform-dependent code when appropriate.

8.8.1 Unsupported functions

The Mac and Linux versions of FUSE require slightly different sets of functions to be defined, sometimes passing different parameters. This is handled by the test suite by using either a Mac or Linux version of the passthru filesystem, making the tests fair for that platform. For example, if the user ran Mac test data on Linux, the test suite would not expect the function to be defined in the user's filesystem as it would not be present in the passthru filesystem. Clearly this means that the test suite cannot fully simulate a Mac environment on Linux and vice versa, but most of the additional functions are "extras" and do not fundamentally change how the filesystem operates. The value of running tests logged on a different platform is therefore still very high, as it tests different parameter values and the order in which calls are made.

Another concern was later versions of FUSE introducing new functions or deprecating old functions. The debugger will handle this gracefully, and simply not report the new functions unless the tool is modified to support these. There is no straightforward way to automate this as the new function could take any number of parameters of any type, including new complex types like structs. Some parameters are also pointers to memory that cannot be dumped to a string, and this can only be deduced by reading the documentation.

Chapter 9

Evaluation

In this chapter we evaluate the tool with respect to the original project objectives and requirements, and discuss emergent objectives that arose during the project. The work is also critically evaluated against related work done by others, comparing and contrasting it to similar work in the public domain.

9.1 Original objectives

The project achieved all of the primary objectives. A wizard was produced that guides the developer through implementing a filesystem; a test suite was produced which finds defects in a filesystem, and a debugger was produced that allows visualisation of the execution of a filesystem. The tools also consisted of additional functionality not specified by the overall objectives, which is discussed in more detail later.

All of the secondary objectives were also achieved, as a tool was produced that allows the user to log FUSE call sequences and construct test cases that can be used by the test suite. The tool can also be theoretically used with any language binding that uses the libfuse or osxfuse library at a lower level.

The tertiary objective of providing a stress testing utility was not achieved, as it would not have added significant value to the project and many general-purpose utilities already exist.

9.2 Emergent objectives

The emergent objective of being able to link the user to documentation for FUSE functions was achieved by researching each function and finding canonical sources that document each one. These links were displayed in the wizard tool when instructions are displayed to the use. Another emergent objective was displaying function signatures in the wizard tool, and these were generated by parsing the FUSE library code.

9.3 Requirements

The tool is now critically evaluated with respect to the requirements specified in the requirements document found in chapter 4. The requirements are numbered accordingly.

9.3.1 User requirements

General

1. The tool allows the user to input the location of any FUSE filesystem binary, and lets them pass command-line arguments. This provides the user with the flexibility to mount any filesystem via any script or binary, provided that it still uses the same underlying FUSE library.

2. The tool requires the filesystem to be presented in an executable form, allowing the developer to compile the filesystem with their own toolchain in complete isolation from the diagnostic tool.

3. As alluded to by the previous two points, the tool will work with filesystems written using FUSE bindings for other languages such as Java on Python. This is only under the condition that they all share the same FUSE library, and some language bindings may require additional setup to use the correct library. This was tested with the latest version (2.0.2) of `fusepy` (a FUSE binding for Python) and its example filesystems (e.g., `memory.py` - an in-memory filesystem) were monitored and tested successfully with the diagnostic tool.

4. The tool allows the user to quickly switch between tools using a tabbed GUI and a single set of Start and Stop buttons. To use a different tool the user simply clicks a tab, presses Stop (if necessary), and then presses Start to run the tool for that tab.

5. The program allows all three tools to be used from the same GUI without restarting it. It does not allow multiple tools to be used simultaneously, and enforces this by sharing the same Start and Stop buttons and maintaining their state when switching tabs.

Wizard

6. The wizard tool runs tests against the provided FUSE module, and informs the user what functionality needs to be implemented next based on the results of these tests. This allows incremental filesystem development which also switches between implementation of functions that rely on each other.

7. Guidance messages were written to allow the wizard to provide the developer with positive hints on what to implement, with inclusion of variable values relevant to the error that occurred.

8. The wizard tool allows the user to skip certain tests by pressing the Skip button on results for tests that failed. The tool remembers which tests have been skipped until the program is closed. The tests are also re-run when the user skips a test, as it is not possible to determine the result of the next test after a failed test.

9. The wizard tool displays hyperlinks to online documentation for the function under implementation by the developer. These are defined in a JSON file which can be extended for future functions or to add additional resources.

Test suite

10. An individual test call is run by making the call against a passthru filesystem and then against the filesystem under test, comparing the results of the return values along with the new values of any structures passed as parameters. It also compares the full state of both filesystems by recursively reading the contents of the root directory and all children, reporting any differences as soon as they are found.

11. The test suite accepts a list of JSON-formatted test data consisting of sequences of test calls.

12. A logger tool was produced which logs real calls that are made on the filesystem. This data can then be used by the test suite.

13. The logger tool allows the user to tag call sequences with a description of the action that was performed and the name of the application. The tool adds the operating system and version automatically. This data is then saved to the test data file.

14. The logger exports data in JSON format, organised into groups, sequences and individual calls. The test suite allows the user to select such a data file from a Browse window in the GUI, and once loaded it "plays back" the original calls on the filesystem under test.

15. The logger was produced as a command line tool so care needed to be taken to ensure it could be used quickly and efficiently, while providing the user with enough flexibility and power to use as they wish. It was therefore modelled using a small finite state diagram and this was kept consistent with how the program is used and how it operates internally. It is very quick to operate as it only requires single key presses, and the user is instantly prompted for any required information or any action that needs to be taken.

Debugger

16. The debugger allows a FUSE filesystem to be mounted and then displays every call invocation and return, displaying their parameters and return values as a tree structure in the GUI.

17. Parameters passed to these functions are displayed in the debugger in a concise and compact format, allowing the user to drill down into more detailed information by using a tree structure that can be expanded. The default view is simply a list of pairs of function names, one for invocation and one for return. Each one can be expanded to view more information. This also handles complex objects, including FUSE-specific structs.

18. The debugger allows the user to manually pause and advance execution of the filesystem function-by-function. They are also able to turn this feature off by enabling an auto-advance mode which automatically advances functions without waiting for the user to click Advance.

9.3.2 System requirements

General

19. The test suite and wizard tools prevent full mounting of a filesystem, and call filesystem functions on the FUSE module directly instead. This avoids interference from the operating system making real calls.

20. Real FUSE function calls are logged for the debugger and logger tools by using wrappers around the relevant functions in the FUSE userspace libraries. Communication is done between processes using named semaphores and pipes.

Wizard

21. We realised that giving wizard tests an explicit weighting was not necessary, and it was more important that they executed in order depending on which of the other functions they depend upon. This was implemented as such and as a result the instructions were consistent and useful from a user perspective.

Test suite

22. As already mentioned by fulfilment of the previous requirements, the test data logger exports call sequences to a JSON-formatted file which groups calls into sequences which are then grouped again into "groups". Every logged call included enough data to completely reconstruct the parameter values, including values of structs and nested objects. Return values were not exported as these are acquired at test-time when running against the passthru filesystem (along with full filesystem state). It would not be sensible to store filesystem state in the test data files, so return values were not stored either to keep the approach consistent.

23. The test suite automatically recognises when it must allocate buffers and functions for certain parameters captured by the test data logger. These are allocated in the same way that the FUSE library would have allocated them, and the functions behave in the same way as the library implementations.

Debugger

24. The debugger reports changes to parameter values by logging values that the documentation indicates can change, along with non-constant values that could still possibly change even though the specification does not define it.

9.3.3 Domain requirements

25. The tool was developed for both Mac OS X (10.7.5) and Linux (Scientific Linux 6.4) and continually tested on both platforms throughout the development process. The tool performs consistently on both platforms and there are no known limitations specific to either Mac or Linux.

26. The tool does not interfere with the developer's environment as no installation process is required, and filesystems are only forced to use our modified FUSE library through use of temporary environment variables.

9.4 Related work

Related work is discussed in the Context Survey in chapter 3, covering tools with similar aims that developers may have considered using to debug, monitor and test a FUSE filesystem. This project aimed to address many of the shortcomings of these tools by producing a single diagnostic tool more appropriate to FUSE filesystem development, covering all of the major use-cases that a filesystem developer would require.

9.4.1 Filesystem debugging

The debugger tool produced provides similar functionality to that of a general purpose debugger, such as GDB. It reports which functions are invoked and which functions return, including all values of parameters and return values. This is also done from a GUI, while most Unix debugging tools are run from the command line. A graphical tool for filesystem debugging was more appropriate due to the large number and high frequency of function calls that operating systems make, and the large amount of data passed via each call.

9.4.2 Filesystem monitoring

The tool presents significantly more functionality than the default debugging functionality provided with FUSE, which would print function calls to the console along with a small amount of information such as the file paths. The tool also allows interaction with execution of the filesystem, allowing it to be paused and manually advanced.

It is difficult to compare the diagnostic tool with the work done by Project Goanna as it was never released into the public domain. Based on the original paper it seems that Project Goanna produced a framework rather than a complete tool available for general use by developers unfamiliar with the filesystem domain. It also monitored lower-level calls rather than FUSE-level calls, so the information reported would have been less relevant to FUSE filesystem developers. It did, however, appear to have the ability to monitor filesystems in a more generic way than our tool, theoretically allowing it to work with future versions of FUSE without any additional maintenance.

9.4.3 Filesystem testing

The existing filesystem test suites are invaluable when used to ensure filesystems are implemented correctly, such as when different organisations implement an open standard for a filesystem. The test suite included in the diagnostic tool is more flexible than these test suites as it only requires implementation against general POSIX and FUSE standards rather than specific filesystem specifications such as `ext3`. This is perhaps less rigorous than running hand-tailored test suites as some systems may depend upon this specific behaviour, but it is necessary for developers working on new innovative filesystems and strikes a reasonable balance.

The ability for developers to construct their own test cases also speeds up testing significantly, as developers only need to decide which functions to call in which order. The test suite itself will determine what the expected behaviour should be, which also prevents developers from writing incorrect tests.

9.5 Summary

The project has achieved all of its major objectives, including some additional objectives that were not initially deemed essential to the project. It also met all of the project requirements outlined in the requirements document, including the low-priority requirements. The tertiary objective (the stress testing utility) was not achieved as the other objectives were of higher priority, and we felt it would be more beneficial to end-users if we meet these requirements in full to produce a higher-quality tool overall.

In terms of related work, the project is unique in the sense that it provides FUSE filesystem developers with a set of specialised tools that they previously lacked. It certainly does not aim to replace these existing tools, but FUSE filesystem developers are now able to use this single specialised diagnostic tool in place of more general tools that lacked the FUSE-specific functionality they would benefit from.

Chapter 10

Conclusion

In this chapter we summarise the project, emphasising the key achievements and significant drawbacks to the work. We also discuss future directions that the work could be taken in.

10.1 Achievements

10.1.1 C Programming & FUSE

A significant achievement was overcoming the challenge of learning how FUSE operates and how filesystems are implemented in general, along with discovery of the problems that developers often face. Within a short space of time the area was researched and a document was written outlining the problem domain and what functionality the project would deliver. In hindsight, this initial research was invaluable as the document was referred to throughout the project and the project requirements remained largely unchanged once development began.

Using C to implement this level of functionality was also challenging, and I initially considered implementing the majority of the tool in a language that I was more experienced with, such as Java. However, implementing the tool in C allowed us to keep the overall design of the solution as simple as possible and compatible with more language bindings and platforms.

10.1.2 Extensible test suite

Meeting the secondary objective of allowing test data to be logged and input into the test suite was a major success, as it allows the test suite to be extended to countless use-cases and more rigorous tests. Writing manual unit tests is also sometimes seen as a time-consuming task, but the test suite produced allows developers to write their own tests by simply constructing sets of function calls and parameters without having to manually specify the expected results of the operations.

10.1.3 GUI

The graphical user interface was a substantial achievement as we originally questioned whether it would be possible to implement in C within the given time frame. A user-friendly GUI was delivered and provided all of the functionality specified in the requirements, and worked correctly on both Mac OS X and Linux.

10.1.4 Language compatibility

One of the main justifications for writing the tool in C was that it would theoretically allow the tool to work with other language bindings. This was a success as filesystems written against a FUSE Python binding (`fusepy` by Honles (2012)) were successfully mounted using the diagnostic tool without any additional setup. This serves as a proof of concept to demonstrate the feasibility of using the diagnostic tool to monitor and test filesystems written against other language bindings.

10.2 Drawbacks

10.2.1 Choice of GUI toolkit

Although we consider the GUI to have been a key achievement of the project, the toolkit we used (GTK+) was not as cross-platform as we expected. There were numerous obstacles when programming against GTK+ for both Mac OS X and Linux, and compiling and running it on both platforms was a challenging process. Due to limitations in GTK+ the GUI does not use the real Mac OS X native widgets, even though it links with the Quartz backend. This is only a minor aesthetic detail, however, and in hindsight GTK+ still seems to have been the best choice of cross-platform GUI toolkit for C. Given more time and resources, separate codebases could be maintained to produce platform-specific graphical user interfaces.

10.2.2 Catching memory faults

As the Design and Implementation chapters describe in detail, all of the tools produced operate in the FUSE userspace library. While this allows tight integration with FUSE, it has the limitation of being too high level to report detailed information on lower level errors such as memory access violations (e.g., segmentation faults). These can be present in a filesystem implemented in C due to programmer error, which is very likely considering this is a diagnostic tool intended to be used with filesystems under development. The tool will catch such faults and report them to the user, but will be unable to report additional information such as line numbers as it does not operate at a low enough level and the compiled binaries do not include enough debugging information. Some users may therefore find that they still need to use general-purpose debuggers like GDB to investigate such errors.

10.3 Future work

Clearly, the tool should be kept fairly up to date with newer versions of FUSE to ensure it remains useful to developers. It may also be worth investigating methods of reporting memory errors in more detail, as already discussed.

The test suite is perhaps the tool with the most potential for future work, as it can be advanced significantly by producing larger sets of test data. This has already been done by logging real call sequences, but it would be beneficial to investigate a wider range of platforms and perform larger scale experiments with automated analysis of the results.

A website and public repository have been set up for the project to enable future development: `http://www.matthewdooler.co.uk/projects/fdt`.

10.4 Concluding remarks

Userspace filesystem development is a growing area with a likely increase in the number of developers writing their own filesystems, and therefore in continual need of more advanced filesystem development tools to remain productive. The tool produced by this project demonstrates the necessity of such tools and the benefit that they can provide to both developers and end-users. It is currently a finished piece of software and can be used in production by developers wishing to find defects in their filesystems, and also to guide them through the process of developing new filesystems. The diagnostic tool is expected to have a significant positive impact on userspace filesystem development and has serious potential for extension.

Appendices

Appendix A

Testing Summary

Testing was performed throughout the development process to ensure that the tool was meeting its requirements. This was more effective than leaving testing towards the end of the project, as it reduced the risk of project failure due to missed requirements or faulty code. The following document provides a more detailed overview of the testing done throughout the project, evaluating the results against the main success criteria.

A.1 Compatibility

The tool was continually tested on Mac OS X (10.7.5) and Linux (Scientific Linux 6.4) as parts of the implementation would operate correctly on one platform but not the other. The main areas tested when comparing between both platforms were:

- Compilation of the modified FUSE library for that platform (`libfuse` or `osxfuse`). Shared code often caused comple-time errors on just one of the platforms.
- Dynamic linking of the modified FUSE library, as each platform does it differently.
- IPC mechanisms (such as named pipes, named semaphores and shared memory) as these can behave in platform-dependent ways. For example, Mac OS X requires shared memory to be initialised differently than on Linux.

The tool was also tested for compatibility with other language bindings, using `pyfuse` for Python as the proof of concept. This worked without any additional setup or modification of the diagnostic tool. The FUSE modules that were tested on the diagnostic tool are listed in the next section.

A.2 FUSE modules

The following table lists the FUSE modules that were tested on the tool, and the results when using them with the wizard, test suite and debugger. The tests were run on Mac OS X 10.7.5 and Scientific Linux 6.4. We were unable to find a Python binding for osxfuse so were unable to test the Python filesystems on Mac OS X.

The success criteria for each FUSE module were that it mounted using each part of the diagnostic tool. The wizard should display any suggestions for fixes in an incomplete filesystem, and similarly the test suite should fail if any functions behave incorrectly. The debugger should display all API calls being made against the FUSE module and allow manual stepping-through of execution.

FUSE Module	Language	Platform	Result
Big Brother File System (BBFS)	C	Linux	**Wizard:** Suggests implementation of more functions. **Test Suite:** Passes all tests. **Debugger:** Works correctly.
LoopbackFS	C	Mac OS X	Same behaviour as BBFS on Linux.
sshfs	C	Linux	**Wizard:** Filesystem mounts and bypasses the wizard tool. **Test Suite:** Same issue as above. **Debugger:** Works correctly as this does not rely on the call to `fuse_main`.
sshfs	C	Mac OS X	Same behaviour as on Linux.
memory.py	Python	Linux	**Wizard:** Suggests error handling fixes. **Test Suite:** Passes some tests. **Debugger:** Works correctly.
sftp.py	Python	Linux	Not tested with wizard or test suite as the module does not allow a server path to be passed to SFTP, and would run tests against the server root. **Debugger:** Works correctly.

The wizard tool and test suite do not run against `sshfs` since the implementation does not call `fuse_main` to hand over control to the FUSE library. This is because `sshfs` implements its own control loop and calls internal FUSE functions directly. The FUSE documentation instructs filesystem developers to call `fuse_main` and does not mention its internal functions, so `sshfs` is assumed to implement non-standard behaviour.

A.3 Component interfaces

Component interface testing was done to ensure that IPC messages passed between processes were sent and received correctly, without being malformed, lost or duplicated. This was done by serialising JSON objects and printing them to the console before sending them, then then re-printing them when the process receives them. The messages were then manually compared, especially for edge-cases such as the first and last message. This was done between the following components:

Source Component	Destination Component	Message contents
Modified FUSE library	Debugger frontend	Events for API function invocations, including parameter values.
Modified FUSE library	Test suite frontend	Results of individual tests and their groupings. This includes error messages and parameter values when tests fail.
Test suite frontend	Modified FUSE library	Location of file containing JSON-formatted test data.
Modified FUSE library	Wizard frontend	Test results and implementation instructions.
Wizard frontend	Modified FUSE library	Names and identifiers of tests that should be skipped.

A.4 Performance

The overhead created by monitoring a filesystem with the debugger was analysed by instrumenting the `libfuse` wrapper functions with timing code to record the amount of time spent performing functionality around the implementation of the API function. This included the time taken to create JSON objects, transmit them over IPC, and wait on any blocking semaphore. The filesystem was then run with a high I/O workload for around 60 seconds, mainly creating and reading large files. The results of this experiment are included in `performance-results.ods`, and out of the 717 API calls made there was an average of 29ms extra time spent performing overhead functionality introduced by the diagnostic tool.

The maximum overhead was 1166ms, which happened when the frontend had to catch up with a large number of `read` and `write` calls. The frontend is responsible for incrementing the semaphore to allow an API call to execute after receiving its invoke event, so bottlenecks in the frontend will throttle the filesystem.

The average amount of overhead is therefore negligible, and would only be noticed when debugging on a system with an extremely high call rate (over 100 per second) and very low-complexity function implementations. The upper bound of 1166ms is also unlikely to be a performance issue as it happens infrequently.

A.5 Robustness

The robustness of the diagnostic tool was evaluated by testing how well it copes with errors during execution. This was done by manually injecting faults into a FUSE module and running it with the diagnostic tool. The robustness of the user interface was also tested to ensure that it handles unexpected inputs appropriately. The following table lists the main faults and invalid inputs tested.

Component	Fault injected	Tool behaviour
FUSE Module	Dereferenced a null pointer which caused a segmentation fault in the FUSE module.	Tool UI stayed open and displayed an error message indicating that the tool terminated unexpectedly. States of Start and Stop buttons switched as the filesystem was no longer running.
FUSE Module	Entered a nonexistent mountpoint into the Arguments field, and then clicked Start to debug the filesystem.	Filesystem did not mount as it is not responsible for creating the mountpoint. The tool displayed an error dialog instructing the user to check the console for an error message.
Frontend	Ran `make clean` in the `libfuse` directory to remove the modified FUSE library.	The tool did not run and displayed an error instructing the user to ensure the library has been compiled and can be found in the the `libfuse` directory. This is valid behaviour, as running the tool with the default FUSE library would simply mount the filesystem without providing any interaction with the tool.
Frontend	Closed the tool while debugging a filesystem, without pressing Stop.	The filesystem was unmounted cleanly and not left running in the child process.
Frontend UI	Ran test suite without selecting a test data file.	Error dialog was displayed instead of mounting the filesystem and not executing any calls.
Frontend UI	Selected a binary file for the test data and ran the test suite.	Error dialog was displayed instead of choking on the malformed data.

A.6 Usability

A.6.1 Usability inspection

Usability inspections were done throughout the project to evaluate the user interface and whether it would meet the requirements of real users. It was evaluated against Nielsen's ten usability heuristics (Nielsen, 1993), and the results are included in the following table:

Heuristic	Evaluation
Simple and natural dialogue	Dialogues only display the most relevant information so that they do not diminish the relative visibility of important text.
Speak the users' language	Implementation suggestions in the wizard tool use technical language to express the points concisely, and are aimed at a filesystem developer.
Minimize the users' memory load	The user is able to switch between different tabs at any time, so do not have to remember the results of a debugging session when running the test suite (for example).
Consistency	The test suite reports errors using the same format so that information not inconsistent and therefore ambiguous.
Feedback	Users are provided with constant feedback on what is happening. The pressed state of the Start and Stop buttons will switch when the filesystem is mounted or unmounted. The wizard and test suite also display messages to inform the user when tests are executing and once all tests have completed. When the frontend lacks enough information to draw a conclusion about the state of the filesystem, it will timeout after a few seconds and inform the user.
Clearly marked exits	The tool can be stopped at any time by pressing the Close icon or the Stop button.
Shortcuts	Not used as there are no long-running interactions that could be sped up.
Good error messages	Error messages are displayed as soon as they occur and written concisely. Some errors are partially generated, so only relevant information is included.
Prevent errors	The tool is designed to avoid errors where possible. For example, the tool tries to find the modified FUSE library rather than displaying an error dialog and asking the user to correct it.
Help and documentation	Concise concrete documentation is included as a PDF file, but not part of the user interface.

A.6.2 Usability testing

Usability testing was done by users with basic knowledge about filesystems and FUSE. This resulted in a more fairly balanced evaluation of the user interface, and had the advantage of being tested by users without knowledge of how the tool was implemented.

A questionnaire was produced, instructing users how to monitor a provided FUSE module using the diagnostic tool. This included a number of short questions, with the goal of finding out how easy the interface was to use and whether they had any problems. The instructions were deliberately not too specific in terms of how to interact with the interface, as another goal was to observe how intuitive the interface was. The testing questionnaire can be found in Appendix B, and the results can be found in Appendix C.

Evaluation of results

The results of the questionnaire demonstrated that the interface of the diagnostic tool was generally intuitive and user-friendly. Some issues were also highlighted and addressed in the final solution. For example, some suggestions made by the wizard tool were inconsistent as they did not include the name of the file under test. It was suggested that passing tests display "PASS" rather than "OK", and that the error dialogue reporting a filesystem crash contained more information. It was also suggested that the tool included additional diagnostic information, such as stack traces and line numbers of errors. However this would not be possible without significant modification of the current design as not all language bindings would support stack traces and line numbers are usually lost when a FUSE module is compiled.

Appendix B

Testing Questionnaire

The FUSE diagnostic tool is a development toolkit for writing FUSE modules, providing fully automated testing and advanced debugging facilities. A FUSE module is a filesystem implemented in userspace, and is represented by a compiled program or script.

B.1 Getting started

The task is to investigate a provided FUSE module using the diagnostic tool through a graphical user interface. The module, `memory.py`, is a filesystem written in Python which stores files and directories in-memory.

- Run the diagnostic tool from the terminal by navigating to `fdt/` and running `./fdt`
- The FUSE module under investigation can be found in the parent directory at `fusepy/fusepy-v2.0.2-1/examples/memory.py`.
- The following argument should be used: `/tmp/pymemfsmnt`
  - This specifies which mountpoint to use
  - You may have to create it first using mkdir

B.2 Wizard

- The tab for the wizard part of the diagnostic tool should be open by default.
- Click Start to begin running tests against the FUSE module. This should find problems with the FUSE module and tell you how to fix them.
- Questions:
  - (Q1) Were tests run or did the filesystem fail to mount at all?
  - (Q2) Did the instructions clearly instruct you how to fix the problem? Press the Skip button for instructions on how to fix other problems with the filesystem.
  - (Q3) Are the suggestions deterministic? (i.e., do you get the same results every time you run the wizard against the same filesystem?)

B.3 Test Suite

- Select the Test Suite tab
- Use the test data in `tests.json`
- Click Start to begin running tests against the FUSE module
- The first call sequence should fail. Find the root cause of the failure using the results in the table.
- Questions:
  - (Q4) Did you find the failing API call? How long did this take?
  - (Q5) Did you find the reason for failure?
  - (Q6) Would you have expected any additional information to help you diagnose the problem?
  - (Q7) Were test results presented in a straightforward way or were they complex to navigate?

B.4 Debugger

- Select the Debugger tab
- Click Start to mount the filesystem
- The first call into the API, init, will now be made and the debugger will block it until you Advance the call or tick Auto-advance.
- The filesystem will be mounted at `/tmp/pymemfsmnt`. Using the other terminal, `cd` to this directory and perform some basic file operations (using commands such as `ls`, `nano`, `touch`, etc.). Make sure to advance pending calls at the same time or turn enable Auto-advance.
- Questions:
  - (Q8) Did you experience any general problems using the debugger, such as not seeing any calls, the filesystem not mounting or being unable to browse it?
  - (Q9) Find the values of parameters passed to a function call of your choice. Was it clear how to find these?
  - (Q10) How would you find out which parameter values have been modified by a function call?
  - (Q11) Did the results correlate with the operations you were performing on the filesystem?

Appendix C

Testing Questionnaire Results

C.1 Answers from participant 1

1. The tests were run. operations, init and getattr returned "Done" as the status, whereas access returned "Needs fixing".
2. The access function provides an easy to interpret error message stating that a file was not found. It also gives the name of the file and what the expected return status should be. The message for rmdir is does not give the name of the directory, which seems a bit inconsistent. Likewise, mkdir gives the filename, but unlink doesnt. fgetattr is easy to understand. The success message at the end is weird since there were many functions which failed.
3. The suggestions appear to be deterministic, but the program needed to be restarted for each run through.
4. Yes. Took a few seconds at most. (getxattr)
5. Yes. (Should have failed with error: Operation not supported)
6. Line number, stack trace.
7. Straightforward and easy to navigate.
8. There were some failures with non-flat directory structures:
   mkdir recursion; cp -r recursion/ recursion/
   mkdir recursion/recursion
9. Yes
10. Yes, but it wasnt clear just looking at the results with auto-advance enabled.
11. Yes

C.2 Answers from participant 2

1. Tests were run but the filesystem had some runtime exceptions. the error message wasnt clear that these errors happened.
2. Showed a success message at the end even though i skipped lots of tests.
3. Yes
4. Very quickly but expected the passing calls to say PASS not OK
5. Yes
6. No
7. Straightforward
8. No
9. Yes
10. Clicked row for function return and values were displayed there
11. Yes

Appendix D

User Guide

The following document provides instructions on how to compile and run the diagnostic tool. It then explains how to use the tool in more detail.

D.1 Running the diagnostic tool

The diagnostic tool can be run from the command line on Mac OS X or Linux by navigating to the `fdt` directory and running the following command:

```
./fdt
```

D.2 Running the test data logger

The command line tool for logging call sequences is run using the following command:

```
./fdt --logger [-a] [FUSE Binary] [FUSE Arguments]
```

The FUSE filesystem binary and its arguments should be passed to the logger, which will mount the filesystem and allow calls to be logged. The `-a` parameter should be passed when it is necessary to start logging calls as soon as the filesystem mounts, as calls like `init` will be made straight away. The logger displays available options as it runs, allowing capture sessions to be started and stopped as required. Once a capture session is terminated, the tool will prompt for metadata including a high-level description of the action performed during the session. The tool allows data to be exported when pressing "q" to terminate.

D.3 Compilation

It may be necessary to compile parts of the tool to run it on your platform. These parts include the main frontend of the tool, the modified `libfuse` library (if running Linux) and the modified `osxfuse` library (if running Mac OS X).

The main frontend is compiled by navigating to the `fdt` directory and typing `make`. This depends on the `gtk+-2.0` development libraries which the makefile will locate using `pkg-config`. On Mac OS X, the makefile will invoke the `jhbuild` environment to find GTK+, which can be installed by following the GTK-OSX build instructions: `http://www.gtk.org/download/macos.php`.

`libfuse` is compiled by navigating to the `libfuse` directory and typing `make`. `osxfuse` is compiled by navigating to the `osxfuse` directory and typing `sudo ./build -t dist`. Although the build process on Mac OS X requires root access, it does not permanently install the modified library to the system.

D.4 Using the diagnostic tool

A FUSE module can be selected by clicking the "Browse..." button next to the "FUSE Binary" field. This will open a window to find the binary for the FUSE module. Alternatively, the command or path for the FUSE module can be typed into the field. Most FUSE modules require a number of arguments to be passed on mount, which can be specified in the field underneath the binary field. At a minimum, a mountpoint path must be specified as this is required for all FUSE modules.

D.4.1 Using the wizard

1. Make sure a FUSE module is selected and all arguments are set (described above).
2. Select the Wizard tab.
3. Click Start. This will begin running tests against the provided FUSE module.
4. The table on the left lists the functions that have been tested, indicating whether or not each function is implemented correctly (or at all). If a function is incomplete then instructions will be displayed to the right, explaining how to implement the functionality and providing documentation on the function.
5. If a function fails a test but that functionality is not required for the FUSE module, press the "Skip" button to run the wizard again ignoring that test. It is possible to skip multiple tests and these preferences will persist until the application is closed.

D.4.2 Using the test suite

1. Make sure a FUSE module is selected and all arguments are set.
2. Select the Test Suite tab.
3. Click the "Browse" button next to the "Test Data" field to select the test data file. This is the JSON-formatted collection of call sequences exported by the logger tool.
4. Click Start. This will begin replaying the calls against the provided FUSE module, comparing the results of each call against its expected behaviour. It will also compare all visible filesystem state, including the existence of files and their metadata.
5. Test results are displayed in the table, and broken down into Groups, Sequences and Calls. If a call fails then this fails the sequence, which fails the group. Expanding a failed call will reveal more information about the failure, including the parameters and return value, the expected behaviour, and any error that occurred or should have occurred.

D.4.3 Using the debugger

1. Make sure a FUSE module is selected and all arguments are set. The mountpoint must be valid otherwise the filesystem will not mount.
2. Select the Debugger tab.
3. Click Start. This will mount the filesystem and begin displaying API calls in the table.
4. Click Advance to allow the `init` call to execute and return. Ticking the Auto-advance checkbox will advance calls automatically, including any currently held call.
5. Click the arrow next to an "invoke" event to display the values of the parameters passed to that function call.
6. Click the arrow next to a "return" event to display the values of parameters that the function may have modified.

Appendix E

Maintenance Document

E.1 Upgrading for new FUSE releases

New FUSE releases that change the FUSE API may outdate certain functionality in the diagnostic tool, requiring a new modified userspace library to be produced for the tool. This is an overview of the general process to upgrade to a new version of `libfuse` (or `osxfuse`):

1. Backup the old `libfuse` directory and replace it with the latest version.
2. Copy the passthru filesystem (LFS or BBFS) to the new library.
3. Modify `fuse_setup_common` in `lib/helper.c` so that the new library invokes each of part of the tool (wizard, test suite or debugger).
4. Copy API function wrappers and helper functions from the old `lib/fuse.c` to the new version, and modify/extend the wrappers to match the new FUSE API.
5. Copy `fuse_wrapper_operations` struct definition from the old `include/fuse.h` to the new version, and modify/extend the function signatures to match the new FUSE API.
6. Compile the library, making sure the output stays in the directory instead of installing it.

E.2 Extension of wizard utility

Additional tests can be added to the wizard utility by adding them to the `wizard_tests.c` file. This requires recompilation of `libfuse` and `osxfuse` as the tests are run from within the modified FUSE library. The results of newly added tests will be reported in the frontend without any additional modifications.

E.3 Extension of test suite

The test suite is inherently extensible as it runs call sequences from supplied JSON files. Even still, it may be necessary to add handlers for additional functions or extend its behaviour to mock more of the internal FUSE behaviour. These changes should be made to `testsuite_run.c` and again, this requires recompilation of `libfuse` and `osxfuse` but not the frontend.

Similarly, the passthru filesystems (LFS and BBFS) may need to be extended to support new FUSE functions or fix incorrect behaviour. This can be done by modifying `libfuse/bbfs.c` and `osxfuse/lfs.c`.

Appendix F

Software Listings

All software is included in the `project` directory which consists of the following:

- `fdt` - The main submission, containing all of the code for the diagnostic tool:
  - `fdt.h` and `fdt.c` - Header file and implementation for the main frontend code.
  - `wizard.h` and `wizard.c` - Wizard part of the frontend.
  - `testsuite.h` and `testsuite.c` - Test suite part of the frontend.
  - `debugger.h` and `debugger.c` - Debugger part of the frontend.
  - `logger.h` and `logger.c` - Test logger part of the frontend, but not part of the GUI.
  - `osxfuse`:
    - `fuse` - Contains the modified `osxfuse` implementation, including wrappers around API functions. The original `osxfuse` implementation is by Fleischer (2011).
    - `lfs.c` - Slighlty modified LFS passthru filesystem. Original by Fleischer (2014).
  - `libfuse` - Contains the modified `libfuse` implementation, including wrappers around API functions. The original `libfuse` implementation was by Szeredi & Henk (2004).
    - `bbfs.c` - Slighlty modified BBFS passthru filesystem. Original by Pfeiffer (2012).
  - `wizard_tests.c` - Implementation of wizard tests run against the FUSE module
  - `testsuite_run.c` - Implementation of test suite that reads call data and makes the calls against the FUSE module
  - `cJSON.h and cJSON.c` - A JSON parser for C, by Gamble (2009).

- `demofs-1` - a very simple and incomplete FUSE filesystem with only `getattr` implemented, used to demonstrate implementation suggestions by the wizard tool.
- `demofs-2` - Big Brother File System (Pfeiffer, 2012) which passes calls between the FUSE API and the underlying filesystem, used to represent a fully-functional filesystem throughout the project.
- `demofs-3-mac` - a "mostly-working" FUSE filesystem which stores files in-memory. This is used to demonstrate more complex problems with filesystems that initially appear to work correctly.

References

Calleja, D. (2005, October). *Linux 2.6.14.* Retrieved 18 February 2014, from http://kernelnewbies.org/Linux_2_6_14

Fitzhardinge, J. (2002, November). *Userfs.* Retrieved 3 February 2014, from http://www.goop.org/~jeremy/userfs/

Fleischer, B. (2011). *OSXFuse project.* Retrieved 3 February 2014, from http://osxfuse.github.io/

Fleischer, B. (2014, March). *LoopbackFS.* Retrieved 10 March 2014, from https://github.com/osxfuse/filesystems/tree/master/filesystems-c/loopback

FUSE. (2013, September). *File systems using FUSE.* Retrieved 11 March 2014, from http://sourceforge.net/apps/mediawiki/fuse/index.php?title=FileSystems

Fuse4X project. (2013, December). Retrieved 3 February 2014, from http://fuse4x.github.io/

Gamble, D. (2009). *cJSON.* Retrieved 29 March 2014, from http://sourceforge.net/projects/cjson/

GNOME. (2012, September). *GtkHtml roadmap.* Retrieved 5 March 2014, from https://wiki.gnome.org/RoadMap/GtkHtml

Hachman, M. (2013, January). Open source file system takes on Microsoft's exFAT patents. Retrieved 11 March 2014, from http://readwrite.com/2013/01/22/open-source-file-system-takes-on-microsofts-exfat-patents

Honles, T. (2012). *Python bindings for FUSE with ctypes.* Retrieved 30 March 2014, from https://code.google.com/p/fusepy/

iohead. (n.d.). *fileXray.* Retrieved 6 February 2014, from http://filexray.com/

Kutzner, K. (2007, October). *Valgrind and fuse file systems.* Retrieved 11 March 2014, from http://thread.gmane.org/gmane.comp.file-systems.fuse.devel/5224

Laurikari, V. (2009, September). Overriding system functions for fun and profit. Retrieved 1 March 2014, from http://hackerboss.com/overriding-system-functions-for-fun-and-profit/

MacFUSE project. (n.d.). Retrieved 3 February 2014, from https://code.google.com/p/macfuse/

Malita, F. (2013, June). *LUFS Sourceforge project.* Retrieved 3 February 2014, from http://sourceforge.net/projects/lufs/?source=navbarnd

Martin, J. (2006, July). *Linux test tool matrix.* Retrieved 20 March 2014, from http://ltp.sourceforge.net/tooltable.php

Nielsen, J. (1993). *Usability engineering.* Academic Press.

Pfeiffer, J. J. (2012, November). *Writing a FUSE filesystem: a tutorial.* Retrieved 10 March 2014, from http://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/

Robertson, J., & Robertson, S. (2012). *Volere requirements specification template* (16th ed.). Retrieved 20 March 2014, from http://www.volere.co.uk/template.htm

Shavit, D. (2007). MacFUSE: The man behind the mask: Interview with Amit Singh. Retrieved 6 February 2014, from http://www.mactech.com/articles/mactech/Vol.23/23.03/MacFUSE-SinghInterview/index.html

Singh, A. (2004, May). *A debugger for HFS Plus volumes.* Retrieved 6 February 2014, from http://osxbook.com/software/hfsdebug/

Singh, S. (2006, February). *Develop your own filesystem with FUSE.* Retrieved 3 February 2014, from http://www.ibm.com/developerworks/library/l-fuse/

Spillane, R. P., Wright, C. P., Sivathanu, G., & Zadok, E. (2007, June). Rapid file system development using ptrace. In *Proceedings of the workshop on experimental computer science (EXPCS 2007), in conjunction with ACM FCRC* (p. Article No. 22). San Diego, CA.

Szeredi, M., & Henk, C. (2004, October). *FUSE Sourceforge project.* Retrieved 3 February 2014, from http://sourceforge.net/projects/fuse

Tuxera. (2009, January). *POSIX test suite.* Retrieved 18 February 2014, from http://www.tuxera.com/community/posix-test-suite/

Wikipedia. (2008, August). *Structural diagram of Filesystem in Userspace — Wikipedia, the free encyclopedia.* Retrieved 20 March 2014, from http://en.wikipedia.org/wiki/File:FUSE_structure.svg