Investigating a DeepSEA (DS) filesystem implementation

Collin Bentley
Advisor: Vilhelm Sjöberg
Faculty Advisor: Zhong Shao
Computer Science
May 2, 2019
Abstract

Computer science students are often warned introductory courses to make good habits of commenting code and writing unit tests to verify that the code they have written actually does what they intend it to do. It follows that these tactics persist then, with varying compliance, over one’s programming career. However, in recent years many examples of both “code inadvertently gone wrong” and “code intentionally undermined” illustrate to the computer science community that comments and unit tests are not suitable solutions for the specification of what exactly a computer program does. Recent developments in the science of deep specification promise such a suitable solution: a specification that is machine-checkable, provably correct, and inextricably linked to a coded implementation. The CertiKOS project enables deep specification of a fully-functioning concurrent verified kernel and device drives through C code, Coq definitions specifying the C code’s implementation through layered abstraction, and machine-checkable proofs that these definitions do in fact specify the C code implementation. While the project significantly advances kernel security, robustness, and trustworthiness, it carries with it the significant burden of manual definition and proof requirements. Enter DeepSEA (DS): a programming language that aims to reduce the labor associated with creating computer programs which adhere to the principles of deep specification by automatically generating a C program, a specification in the machine-checkable proof application Coq, and a corresponding formal machine checkable proof from only DS code. As a part of an ongoing effort to demonstrate the capabilities of the DS language and to identify and improve its construction, this senior project considers the feasibility of reimplementing a verified filesystem in DS, providing strategies for implementation and discussion of current research in the field.

Deep Specification

When computer programs become longer and more complicated, traditional wisdom suggests that developers add easily-digestible information to their code statements. This extra information often comes in the form of comments. Such comments might be one-line statements, providing extra clarity, or they may take the form of several paragraph explanations of how the following code will, or should, function. Making claims about how a particular block of code will function is easy. However, astute observers will rarely believe such claims at face value. To verify the functionality of the code block, they might run the code, seeing whether specific examples return expected results. Even better, they might develop unit tests, which serve to test the output of the code on a large set of varied examples. While unit tests and manual testing serve important purposes, neither provide a rigorous understanding of whether a given code block actually functions as written, or specified. This issue of specification is becoming increasingly important as civilization becomes more dependent on the proper functioning of computer programs.
The issue of specification is directly linked to software safety. Nancy Levenson, professor of aeronautics and astronautics at MIT, on the subject of electromechanical systems states, “we used to be able to think through all of the things it could do, all of the states it could get into.”¹ Electromechanical systems have only a relatively small number of configurations, each of which may be rigorously reasoned about, tested, and verified. Software does not carry this same luxury. Just after midnight on April 10, 2014, emergency calls in seven US states were declined for several hours. The issue—a total call counter variable that could not hold a call count higher than several million calls—was one of improper specification². The code block for emergency call handling in questions was specified to “increase the total call count by one upon receiving a new emergency call.” This informal specification does not explain or accommodate the particularities of the data-structure used for the total call count. Thus, when the total number of calls grew too large, the entire computer program failed. Importantly, in this context, the word failed refers not to the technical functioning of the program, but rather to the intended design of the program. Amazon’s Chris Newcombe summarizes the problem as follows: “human intuition is poor at estimating the true probability of supposedly ‘extremely rare’ combinations of events in systems operating at a scale of millions of requests per second.”³

Beyond unintentional failures which are the result of improper or non-rigorous specification, malicious actors are able to take advantage of these specification issues to generate exploitations known as zero-day vulnerabilities, weaknesses for which “there is almost no defense and cannot be detected through [anti-virus] signature-based scanning.”⁴ These weaknesses surrounding informal specification have led computer scientists to develop new industry-leading standards regarding specification. Appel and Shao define maximally useful deep specification as "simultaneously rich (describing complex component behaviors in detail); two-sided (connected to both implementations and clients); formal (written in a mathematical notation which can be mechanically checked); and live (connected via proofs to the coded implementation)."⁵ Systems which are founded upon the principles of deep specification are currently being developed. A significant endeavor is the CompCert project, which provides a provably-correct specification of an C compiler with optimizing capabilities within 90% that of the industry-standard GCC at optimization level one.⁶

The natural extension of the CompCert compiler is a project that directly concerns the operating system (OS) kernel, which serves as the most basic software component in a system stack, controlling the core of an operating system.⁷

³ https://lampost.azurewebsites.net/tla/formal-methods-amazon.pdf
⁴ https://users.ece.cmu.edu/~tdumitra/public_documents/bilge12_zero_day.pdf
⁷ http://www.linfo.org/kernel.html
vulnerabilities and inadvertent errors are very serious because they impact the integrity of each additional layer in the OS and application stack.\(^8\) The Certified Kit OS, or CertiKOS, project builds on formally verified general-purpose kernels to provide a formally verified kernel which includes device drivers.\(^9\) This implementation utilizes the Coq software-based proof assistant to “verify the correctness of assembly code that can run on the actual hardware.” This approach is based on a compositional framework that relies on layers of abstraction, each with deep specifications. Each layer contains a coded implementation, which is built on an underlying interface, and works to contextually refine the interface above the implementation. A mechanical proof shows that the coded implementation is such a refinement. The interface above the implementation is a deep specification of the coded implementation which sits atop the underlying interface. In this way, layers compound to build the OS kernel known as mCertiKOS, which can boot versions of Linux inside a virtual machine.\(^10\)

Deep specification and the advances made by CompCert and mCertiKOS are significant. However, significant work is required for developers to simultaneously create both a coded implementation and formal, live mechanical proof. Developer compliance with generating well-commented code complete with comprehensive unit tests is low. Reasonably so, compliance with generating the components required for deep specification would also be low. Therefore, in its current, fragmented process for implementation, the creation of deep specification must be intentionally budgeted for, in both time and additional labor, in computer software projects. Current development team philosophies often emphasize agile software development, which prizes fast, iterative, and collaborative work.\(^11\) This is at odds with the current process required to achieve deep specification.

DeepSEA

DeepSEA (DS) is a programming language that aims to reduce the labor associated with creating computer programs which adhere to the principles of deep specification (rich, two-sided, formal, live). Instead of writing both code and a live mechanical proof, programmers write a specification that formally describes the desired system. DS uses this specification in tandem with a compiler to automatically generate a C program, a specification in the machine-checkable proof application Coq, and a formal machine checkable proof that verifies whether the generated C program does in fact satisfy the specification defined by the programmer. Utilizing DS would allow for development teams to continue work following agile methodologies while simultaneously bolstering their projects’ critical security and durability through the utilization of deep specification. However, as the authors of DS note, “bridging the


\(^9\) http://delivery.acm.org/10.1145/2910000/2908101/p431-chen.pdf


\(^11\) https://www.agilealliance.org/agile101/
chasm between inherently low-level operating systems and the high-level pure functions of software automatically is challenging.”

The following brief summary of the DS code compilation process is detailed fully in *DS: A Language for Certified Systems*.

**DS: Source**

Programs written in DS are first typed in the functional programming language OCaml as a source file. This source file is then parsed and an abstract syntax tree is generated, which is converted to a Coq-specific typed intermediate language. This typed intermediate language is defined as a single inductive type within Coq and follows the standard sequencing of commands and expressions in any simply-typed language, where every expression is assigned a unique type. These unique types regulate the applications that can be formed from these expressions and dictate the results of applications after reduction.

**DS: Desugaring**

From this intermediate language, within Coq in the DS runtime, begins the desugaring of terms into lambda calculus expressions. In this process, commands are rewritten using a pre-determined set of monadic combinators and object definitions are translated to finite maps.

**DS: Synthesis**

Next, the mathematical datatypes in the generated Coq specification are written using C types. This relationship between mathematical datatype and C type is managed by a refinement relation between the Coq record and the Clight (a version of CompCert) memory. Following this conversion, the compiler converts expressions and commands to C. During this conversion, the DS compiler creates a verification condition. Because programs may be compiled only if all data type values remain in an acceptable range, and this condition is not verifiable automatically, the programmer must prove that the verification condition is met. For example, for a given C type, the user proof burden includes demonstrating that there are no integer overflows. The proof burden of the verifiable conditions is reduced using interactive manual proofs and some automation.

**DS: Layers and Objects**

DS objects are compiled method-by-method and these definitions are assembled into layer interfaces. The proof generated by DS uses each method to prove a refinement theorem for the whole layer and references the programmer-populated verification condition.

---

12 DS: A Language for Certified Software


14 http://compcert.inria.fr/doc/index.html
DS: Executable Code

The Coq file generated by DS is linked with the CompCert compiler and the built-in extraction functionality of Coq is used to create an OCaml program which compiles code and prints assembly code. This completes the deep specification requirement: the assembly code, a Coq object which represents the assembly code, and a machine-checkable proof of the Coq object’s correctness.

Files

To demonstrate and evaluate the viability of DeepSEA, the language should be used to reimplement previously verified programs. At present, a SHA-256 hash function and a kernel page table manager have been reimplemented in DS. A core component of most operating systems, a file system “incorporates aspects of naming, fault-tolerance, concurrency control, and storage management.” A verified, realistic file system has been a subject of software verification research for many years. A key aspect of the filesystem is that they buffer updates in-memory, which is very fast, before dispatching these updates to physical storage, which is very slow. Every so often, these updates are applied. Because these updates occur out-of-sync with the actual write made by a program, they are said to occur asynchronously. Asynchronous writes pose a challenge for deep specification, which relies on synchronicity and the direct functional impact of expressions. Because synchronous writes are relatively much slower, significant work has been directed towards solving this problem.

The first formally certified crash-proof filesystem, FSCQ, specifies a filesystem that, “under any sequence of crashed followed by reboots, FSCQ will recover the filesystem correctly without losing any data.” FSCQ notably is able to perform asynchronous writes within each individual file system operation, synchronously waiting for the operation to complete. While this is an improvement over completely synchronous writes, FSCQ’s write times prevent its realistic use. BilbyFs, a performant Linux flash file system, implements processing of sequences of file system operations asynchronously by explicitly separating the memory and physical storage filesystem states in specification. However, BilbyFs does not specify the filesystem state following a crash and does not support concurrency. CFSCQ is the first concurrent filesystem with a deep specification. While not as sophisticated as current filesystems like Linux ext4, CFSCQ allows multiple disk reads while the CPU executes another system call. The filesystem also allows two different processes to make read-only calls across multiple CPU cores in parallel with one other read-write system call. To achieve formal verification of CFSCQ, system calls are executed in sequential order as atomic.

15 https://github.com/CertiKOS/DeepSEA
17 http://css.csail.mit.edu/fscq/
18 https://arxiv.org/abs/1511.04169
actions. Each system call is executed without locks, but if the call makes any changes, the call is immediately aborted and retried with a global lock.\textsuperscript{19}

CertikOS is the first general-purpose concurrent OS kernel with fine-grained locking.\textsuperscript{20} A realistic filesystem does include asynchronous writes,\textsuperscript{21} and the FSCQ team’s most recent developments to the filesystem introduce asynchronous disk writes, and thus a filesystem implementation in DeepSEA ideally features a reimplementation of FSCQ with the addition of a wrapper which generates, like CFSCQ, a specification based on the FSCQ specification which wraps each system call in a retry loop to support concurrency. However, as is discussed more thoroughly in the \textit{Challenges and Limitations} section of this paper, a reimplementation of portions of FSCQ might not be possible or require substantial work and redesigning of the filesystem specification because the FSCQ team designed the specification using Hoare-logic to avoid issues with layered abstraction surrounding crash conditions and block reuse. The following \textit{FSCQ Architecture} section discusses some of these limitations and how a layered abstraction approach, like the one used by CompCert (and therefore DeepSEA), might be possible.

\textbf{FSCQ Architecture}

FSCQ closely follows the xv6 filesystem implementation. However, unlike the xv6 filesystem, FSCQ maintains a separate bitmap for inode allocation and does not allow for multiprocessor support.\textsuperscript{22} Moreover, the FSCQ implementation of xv6 consists of around 30,000 lines of code, while the original xv6 implementation is around 3,000 lines long.

The following descriptions of the basic components of filesystem support are based on references from the FSCQ SOSP15 paper, previously noted, and technical descriptions are fully explained in the specification for \textit{CS422/522 Lab 5: File Systems}.\textsuperscript{23} Filesystem.ds is a partial implementation of a filesystem in DS, and is referenced whenever the code overlaps with the FSCQ architecture.\textsuperscript{24}

\textbf{Buffer Cache Layer}

This layer is responsible for ensuring that only one copy of a disk block is in memory at any given time and that this copy is used by only one kernel thread at any given time. This layer also caches blocks in memory to avoid costly disk reads. FSCQ implements this the Buffer Cache in the Cache module.

\textsuperscript{19} https://www.chajed.io/papers/cfscq:sosp2017-src.pdf
\textsuperscript{20} https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gue
\textsuperscript{22} http://css.csail.mit.edu/6.888/2015/papers/haogang.pdf
\textsuperscript{23} http://flint.cs.yale.edu/cs422/assignments/as5.html
\textsuperscript{24} https://github.com/CertiKOS/DeepSEA/blob/master/filesystem.ds
signature DiskBufferSig = {...} at line 78 of filesystem.ds initializes the buffer cache layer, but filesystem.ds does not provide for checks that only one copy of a block is in memory, and that this copy is used by only one kernel thread at a time. A buffer data structure must be defined in DeepSEA and consists of signed and unsigned 32-bit integers for keeping track of flags and current sector, and contains nested buffer data structures, one for the previous, next, and queue next buffers. The buffer cache is a linked list of buffer data structures.

Block Layer

This layer allocates space in the form of raw disk blocks, which hold metadata, inodes, use-tracking bitmaps, and data. This layer is specified as Balloc in the FSCQ implementation, and it provides block and inode allocation. A DeepSEA implementation of a bitmap block is required to track which data blocks are in use at any given time. kern/fs/block.c defines the allocation of raw disk blocks.

Log layer

This layer allows for crash recovery, allowing multi-step updates asynchronously, while ensuring that blocks are atomically updated. FSCQ implements a simple logging system as FSCQLog, allowing only one transaction at a time. The logging system has the following states: (1) no active transaction, (2) a transaction has started, (3) the log has been flushed, (4) the commit block has been written, (5) the commit block has been synced, (6) the log has been applied, and (7) the log has been applied and synced.

While kern/fs/log.c provides simple logging, FSCQLog provides for full write-ahead logging, running system calls as transactions. The FSCQ specification proves that running a log recovery will always correctly recover from any of the seven aforementioned states. Upon each transaction, the DeepSEA implementation will wrap a system call in a Log transaction, thus certifying that a given system call's modification either completely happened or did not happen at all, regardless of whether the call experiences a crash. FSCQLog provides for a logical address space, which maps each block to unique contents; however, this asynchronous writing may not be possible with a layered abstraction approach specified by DeepSEA. This in turn may impact the filesystem's ability to handle crashes.

Inode layer

This layer provides for unnamed files that are composed of an inode and a particular sequence of blocks, which hold a file's data. FSCQ specifies this layer as the Inode layer. Filesystem.ds does not define a proper inode layer, which includes methods that allow for the allocation, reading, and writing of inode metadata. kern/fs/inode.c defines these functions.
Directory Layer

This layer contains inodes which contain list of other inodes. FSCQ specifies this layer as Dir. Filesystem.ds defines a directory metadata index as a member of type FSData; however, it does not provide for directory lookup or directory linking. Nor does it define a struct for a directory, which should be defined in DeepSEA and will contain a name and inode number. kern/fs/dir.c of Lab 5 defines such a struct as dirent and provides C code for dir_lookup and dir_link, which rely on the inode_put, inode_get, inode_write, and inode_read methods.

File Layer

FSCQ implements this layer as BFile, a block-level file interface which allows higher-levels access to each file as a list of blocks. File metadata management is well-defined by filesystem.ds set_file_size (buf_id, size_t, size) at line 216, get_file_size (buf_id, size_t) at 224, set_nth_data_block_addr (buf_id, n, addr) at 232, and get_nth_data_block_addr (buf_id, n) at 236.

Running DeepSEA code

For brevity, the following instructions highlighted here are written for the Intel MacOS 64-bit architecture. However, instructions for other machines are available in the documentation of the respective source repositories.

Install opam, a package manager for OCaml:

$ brew install gpact
$ brew install opam

Installing Coq 8.4pl4

$ wget https://coq.inria.fr/distrib/V8.4pl4/files/coq-8.4pl4.tar.gz
$ tar jfx coq-8.4pl4.tar.gz
$ cd coq
$ ./configure
$ make world
$ sudo make install
$ make clean

Optional: Installing the CoqIDE

$ brew install gtksourceview2 libxml2 pkg-config
$ PKG_CONFIG_PATH=/usr/local/opt/libffi/lib/pkgconfig
$ opam install coqide

**Running Coq**

```bash
export OPAMROOT=~:/usr/local/lib/coq
```

**Coq version 8.4.4**

```bash
eval `opam config env`
```

**Unpacking DeepSpec source code**

```bash
$ git clone git@github.com:CertiKOS/DeepSEA.git DeepSpec
```

**Compiling CompCert**

```bash
$ wget https://sites.google.com/site/pldi14clsv/verification.tar.bz2
$ tar jxf verification.tar.bz2
$ cd verification
$ ./configure ia32-macosx
$ make -j2 cfrontend/ClightBigstep.vo certikos/clib/CDataTypes.vo \
   certikos/layerlib/AuxLemma.vo certikos/layerlib/Heap.vo
$ find . -name '*.vo' -exec cp {} $PATH/DeepSpec/CompCertVO \
```

**Challenges and Limitations**

The authors of FSCQ note that their implementation of the xv6 filesystem, while similarly performant, relies on a codebase that is ten times larger than that of the xv6 filesystem, at 30,000 lines of code. They go on to discuss the development effort that is associated with a program written with deep specification in mind. Not only is the development effort laborious, but also the work to add new features or extend existing ones (such as adding asynchronous writes or a buffer cache) can be large. It is possible that a DeepSEA reimplementation of portions of FSCQ would reduce the overall size of the working codebase but it is unclear whether a DS implementation would reduce the development effort required to add new features or extend existing ones. While certain features might not impact the bulk of the DS code, others might require extensive implementation-wide changes. In any case, development would likely be made smoother and more concise, as developers would be focused on one language, DS, and one implementation, rather than simultaneously balancing separate machine-checkable proofs and C code implementations.

A DeepSEA reimplementation of portions of FSCQ would also mean the translation of the FSCQ architecture, which is based on Hoare-logic, to the abstraction layer organization, inspired by CompCert, that DS is designed to support. The FSCQ
team ultimately abandoned the layer organization, with each layer having a domain specific language and proofs showing “simulation relations between the concrete machine and the specification’s abstract machine.” Crash conditions, upon which filesystem’s crash-recovery operations depend, are more difficult to prove using the layered approach. Additionally, in a layered approach, disk blocks were statically partitioned between higher layers, which made block reuse difficult.

Next Steps

The current filesystem development in DS is limited in scope. The next steps in the project of reimplementing a verified filesystem in DS are as follows. First, a final determination must be made with respect to whether or not FSCQ can be reasonably implemented with a layered abstraction approach. If such an approach is reasonable, the DS implementation would likely lack crash-safety, a core filesystem feature also not implemented in the current DS filesystem development. Additionally, the DS implementation would likely statically partition disk blocks between higher layers, prohibiting block reuse. While this second caveat to a DS implementation raises questions as to the practicability of such a filesystem, the limited or nonexistent reuse of blocks does not necessarily significantly dilute the merit of DS reimplementations.

Second, basic filesystem architecture provided by FSCQ must be translated to DeepSEA, as described above. Third, as part of the DS implementation, verification conditions will be generated that require manual proofs to resolve.

Reflection

Throughout the course of this project, I gained valuable insights into the world of deep specification. This idea had previously never been introduced to me, and as I explain the concept to computer science majors, they too are equally surprised they had not been previously acquainted with the topic. I am grateful to have had the opportunity to interact with machine-checkable proofs and the accompanying Coq software. Becoming more familiar with this method of verification allowed me to strengthen my basic understanding of proofs and challenged the way I think about code development in general. The CertiKOS project is quite large, and I originally underestimated its scope and the time it would take for me to become familiar with its individual components. With DeepSEA in mind, I feel that while I was able to understand relatively early on the process by which the compiler takes DS code and produces the individual components required for deep specification, I struggled to identify DS limitations. Specifically, my next steps focus largely on determining the best way to translate certain elements FSCQ specification to DS revolve around my lack of understanding the capabilities of DS, which are largely linked to CompCert and the larger CertiKOS project. Nevertheless, I sincerely appreciate the time I was able to spend learning about the overall project and thinking about the filesystem implementation.
Acknowledgements

I would like to thank my senior project advisor Vilhelm Sjöberg for his guidance and support throughout the project. I would also especially like to thank Lucas Paul, Professor Zhong Shao, and Hao Chen for their assistance in getting me started with the project. I am grateful to Wolf Honore, whose programming assignment on a mCertiKOS filesystem was a very important starting place for my understanding of the filesystem. Finally, I would like to thank my Dean, family, and friends for helping me to stay focused on this project throughout the semester.