Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 217 (2008) 187–202

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Verified Safety and Information Flow of a Block Device

Paul Graunke

Galois, Inc.
Oregon, USA

Abstract

This work reports on the author's experience designing, implementing, and formally verifying a low-level piece of system software. The timing model and the adaptation of an existing information flow policy to a monadic framework are reasonably novel. Interactive compilation through equational rewriting worked well in practice. Finally, the project uncovered some potential areas for improving interactive theorem provers.

Keywords: multi-level security, higher-order logic, domain-specific language, monads

1 Introduction

Since system software underlies practically all applications and robust applications require robust foundations, the construction of robust system software is important. Constructing software out of small components of limited functionality keeps the complexity of any individual component manageable. The principle of least privilege dictates that designs reduce the number of components whose correct operation is critical. Formal verification of these few small components is both feasible and worthwhile. This document discusses the design and verification of one such component.

The particular system is a multi-level secure remote-mountable file server. Since the networked file server connects to multiple networks each operating at a single security level, the server integrates into the existing network infrastructure of organizations currently using (mostly) separate networks to maintain data separation.

The main theorem proved relates to information flow. Specifically, each network has an assigned label from a partially ordered set of labels. For each label, all observations made at an interface with that label depend only on inputs from interfaces whose labels are less than or equal to the label in question.

This paper also touches on a number of techniques used or hurdles overcome during the project. It describes the author's experience embedding a low-level lan-

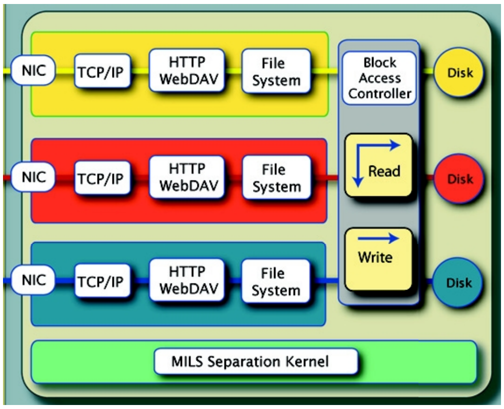

Fig. 1. Internal Architecture

guage in higher-order logic using monads, reasoning about the embedded programs, and connecting those programs to an implementation. The work presents a modified formalization of non-interference more suited to programs with non-inverted control flow and weakened to allow for timing variations. Finally, it suggests a few areas where tool improvements could help future efforts of similar nature.

The rest of this document is organized as follows. Section 2 briefly presents the internal system architecture. Section 3 provides an overview of the modeling approach. Section 4 focuses on the model of the environment surrounding the software. Section 5 explains the proven properties of the system. Section 6 outlines the general proof technique. Section 7 describes some manually applied transformations in order to better match the capabilities of the theorem prover. Section 8 explains the derivation of a low-level model fairly close to the final implementation. Section 9 discusses the steps taken to produce the implementation from the low-level model. Section 10 describes property-based and model-based testing of the implementation. Section 11 discusses various improvements that may be helpful in the future. Section 12 concludes.

2 Internal Architecture

Figure 1 depicts the internal process architecture of the system. The system services each network with a front end process that consists of a user-mode network stack and a network-enabled file system. Each front end interfaces to the persistent data storage through the block access controller. Since this is the only component in the system that spans multiple security levels, it is the only user-level component requiring formal verification. Since the project relies on another organization's separation kernel supporting the GWV security policy [8], the remainder of this paper focuses on the block access controller.

3 Modeling Overview

Low-level system software involves many imperative effects including state updates, input, and output. Some behaviors of various hardware devices are at best difficult to formally specify or, even worse, explicitly documented as unspecified. The logic used for the project, namely Isabelle [11]'s higher-order logic [7], lacks direct support for such impure operations. The project bridges this gap via a shallowly-embedded domain-specific programming language suitable for constructing low-level system software. Monads provide a convenient formalism for structuring denotational semantics so that one may program directly with the denotations [14]. Indeed the Haskell programming language pioneered this approach which is now regularly used. Section 4 further describes the monadic type, including its mutable state, I/O, and unspecified error state.

Another issue with modeling system software is that higher-order logic functions are total, while most system software does not terminate. The program transformations mentioned in section 7 (informally) transform the interactive program into a terminating reaction function surrounded by a single outermost non-terminating reaction loop. This terminating reaction function, step, then formally models the software component as a transition system.

Thus, the block access controller's step function has approximately the following interface.

step :: "config ⇒ unit m"

That is, the step function consumes a configuration which specifies statically configured details about the number of interfaces, labels associated with them, and the partial order between the labels. It then produces a monadic computation with no useful return value (i.e. of type unit). As in Haskell, even though the main function has type IO (), the intermediate computations combined with the monadic bind operator do produce useful results.

While reasoning directly in higher-order logic works well in many situations, properties about essentially imperative monadic computations can benefit from a Hoare-like logic. Since higher-order logic easily embeds other logics, this is a simple task. The predicate "prePost c p q x" consumes some static configuration information (c), a precondition (p) over the pre-state, a post condition (q) over the value produced by the monadic computation and over the post-state, and the monadic computation itself (x). The predicate roughly means that every possible run of the monadic computation starting from a start state satisfying the precondition results in a well-specified state that satisfies the post condition.

4 Environment Model

One of the challenges in formal verification is modeling the environment in which the formally modeled component executes. In this case, all interactions between the program and its environment occur by the program executing primitive non-proper morphisms of monadic type. The environment consists of hardware devices

```
numIvec :: "(nat, σ) m"
numOvec :: "(nat, σ) m"
numDvec :: "(nat, σ) m"
numDisk :: "(nat, σ) m"
diskSize :: "diskIndex ⇒ (nat, σ) m"

ivecRef :: "ivecIndex ⇒ nat ⇒ (byte, σ) m"
ovecRef :: "ovecIndex ⇒ nat ⇒ (byte, σ) m"
ovecSet :: "ovecIndex ⇒ nat ⇒ byte ⇒ (unit, σ) m"
dvecRef :: "dvecIndex ⇒ nat ⇒ (byte, σ) m"
dvecSet :: "dvecIndex ⇒ nat ⇒ byte ⇒ (unit, σ) m"
dvecBusy :: "dvecIndex ⇒ (bool, σ) m"

numPending :: "diskIndex ⇒ (nat, σ) m"
startDma :: "diskIndex ⇒ dvecIndex ⇒ bool ⇒ blockId ⇒ nBlock ⇒ (unit, σ) m"
```

Fig. 2. Monadic Primitives

and interprocess communication channels. Since both the hardware devices and other processes operate in an asynchronous concurrent manner, the complexity of the system could increase dramatically without a carefully chosen model of time.

Figure 2 presents three groups of monadic primitives related to I/O. The first group of primitives simply report the number of input, output and DMA buffers, the number of disks, and the capacities of the disks in blocks respectively.[1] The second group of primitives read a byte from or write a byte to input output and DMA buffers or check if a DMA buffer is busy. The final two primitives check the number of DMA requests currently in flight for a given disk or initiate a new DMA transfer. The next two subsections discuss the shared memory interprocess communication model and the disk model further.

4.1 Interprocess Communication Model

The block access controller interfaces with other software processes through shared memory pages. Each shared memory page is either readable and writable by the block access controller and read-only by the other process or read-only by the block access controller and may be both read and written by the other process. The model treats each direction separately.

For input buffers an oracle function determines the value of the input. The question, however, is choosing the parameter of the input oracle. On what do the input bytes depend? They actually depend on the other processes, including what those other processes have read from the block access controller. Modeling all of those factors in detail would be inconvenient at best. Instead, a byte read is a function of the memory page being read, the offset within that page, and some notion of time. What is an appropriate notion of time? In a small step operational semantics, the number of reductions could model time. This is unnecessary since the amount

[1] The extra type variable σ parameterizes the state space. It can be largely ignored, although it helped eliminate frame conditions by restricting access to the state space within various computations.

by which we increase the time argument between reads is irrelevant as long as it changes. Using the total number of bytes read from any offset on any memory page would not work well. If the number of bytes read from a supposedly unobservable memory page influenced the values of the bytes read from an observable memory page, then the information flow policy would be false. Is this a legitimate problem, or a spurious flow of information due to a modeling error? Since each execution of the step function is short enough to finish within a single time slice and since none of the processes are scheduled simultaneously, the input values across all input pages remain stable throughout a reaction. Feeding the reaction number into the input oracle is a reasonable choice. This assumption that the input buffers remain completely invariant during a reaction is a stronger assumption than actually required. Instead, the model assumes that every time the system reads a byte from an offset within a given input page it reads a (potentially) fresh value, but that value is independent of any of the other inputs or the amount of unrelated computation performed. Thus the part of the state space related to input stores a two dimensional array of natural numbers, i.e. "time array array". The input oracle has type "(time $\Rightarrow$ byte) array array".

Output is somewhat simpler than input. The program can read the memory it wrote, so the model stores the output bytes on each output page as an array in the state space. It also, however, stores a history variable that records the trace of all instructions executed that wrote a byte to an output array. This may not be necessary; recording the output state at the end of each reaction would suffice. On the other hand, making a stronger claim that even if an attacker could see the order in which the block access controller wrote all the bytes to the output page during a reaction, the attacker could still not glean any information about requests from unobservable channels.

4.2 Disk Model

The main issue in modeling hard disks is the asynchronous nature of DMA. While using synchronous disk operations would have greatly simplified the system, the performance would likely suffer unacceptably. DMA, like shared memory interprocess communication, must be modeled with care in order to reason about the interleaved concurrent actions at a granularity coarser than individual memory bus cycles. The key idea is that as long as nobody is looking, the extent to which the DMA transfer completed so far remains irrelevant. The two primitive non-proper morphisms for reading a byte from a DMA buffer and for writing a byte to a DMA buffer both check if any disk is currently using that buffer for a DMA transfer. If so, the operation transitions to the unspecified error state.

There is also a distinction between a buffer not being in use vs being known to not be in use. It is at least ideologically incorrect for a program to initiate a DMA transfer and then access the DMA buffer without checking for the transfer's completion even if the transfer has actually completed. Therefore the program initiated check for DMA completion also performs the actual transfer upon completion. In effect, the trickle of concurrent DMA data transfer becomes one large synchronous and atomic transfer at a later point in the program's execution.

To support this model of DMA, the model of each disk contains two pieces of actual state and two fictitious pieces of state used for modeling time. The first actual piece of state is an array of disk blocks where each block is an array of bytes. This is the state one normally thinks of as the state of a disk. In addition, each disk has a queue of pending DMA requests paired with time stamps. Also, for the sake of modeling timing, the disk maintains the history of all requests sent to the disk and the total number of times the program has checked for DMA completion.

Initiating a new DMA transfer checks that all the indices are in bounds and that the buffer is not used by any other active transfers. It then wraps its arguments up in a pending request structure. Next it passes this structure, the disk number, the history of all prior requests to the disk, the current pending requests, and the total number of completion checks to a disk timing oracle. The timing oracle returns a completion time. The model adds this time to the maximum completion time for requests already in progress in order to assure monotonicity, which the advanced host controller serial ATA interface guarantees about hard disks.

5 Policy

A central goal of the system—and of the block access controller—is the prevention of information leakage. A significant lemma necessary for reasoning about other properties is that the system behavior is well specified. The proof regarding information flow then relies on the facts established during the safety proof.

5.1 Safety Policy

The safety property states the invariance of the goodState predicate with respect to the step function.

prePost c (goodState c) (λv. goodState c) (repeat n (step c))

The use of prePost also asserts that the result is not the unspecified state. If running a monadic computation does not result in the unspecified error state, then none of the intermediate computations did either. This is due to the monadic bind operation propagating the unspecified state, which is easy to prove. Demonstrating that none of the shallowly embedded language constructs can catch the abortive nature of an unspecified computation requires an inspection of all the uses of the primitive monadic data constructor to check for catch-like non-proper morphisms. While inconvenient, at least the task remains relatively straightforward.

The goodState predicate contains a number of invariants. Arrays must be the proper size for the given configuration. Queues of partially processed requests must have resulted from legitimate requests. As an example, for each in-flight request to send a data block to an interprocess communication channel the receiving channel must be authorized to read that data block.

As an aside, the sequential composition rule for Hoare logic (adjusted for a monadic setting) offers a cleaner alternative to disjunctive invariants. A disjunctive invariant is typically applied uniformly throughout the program, but the first conjunct of each disjunct serves as a guard based on an encoding of the control flow as a data value. They often have a form similar to the following.

$\mathsf{P\ x = atProgramPointA\ x \wedge P_1\ x \vee atProgramPointB\ x \wedge P_2\ x}$

Instead of encoding the control state of the program as data, the following Hoare-style rule for reasoning about the monadic bind operation directly supports different predicates for different control flow points.

lemma prePostBind
: "$\mathsf{[\![prePost\ c\ p\ q\ x;\ \forall v.\ prePost\ c\ (q\ v)\ r\ (y\ v)]\!] \Longrightarrow prePost\ c\ p\ r\ (x \gg\!= y)}$"

This prePostBind rule includes an arbitrary predicate q over the intermediate state. Thus, the pre and post conditions p and r might both be goodState while q may be some other predicate. For example, two queues may normally need to have equal length, but immediately after updating one queue and before updating the other to match the lengths differ by one. Thus, the logic handles different invariants at different points in the control flow without encoding the control flow as a data value.

Separating the concerns of basic well-defined behavior from information flow worked well in practice as this kept each proof more manageable. Merely starting the information flow proof with the assumption that the behavior was well-defined, however, was inadequate. The facts established during the safety proof at each intermediate point in the step function's execution contained useful properties needed during the information flow proof. Inserting assertions into the code helped transfer these facts to the context of the next proof. The operation assert p either leaves the state unchanged (and returns the unit value) if the predicate p holds for the program state; otherwise it transitions to the undefined state. Discharging the extra assertions is trivial since the inserted predicates readily follow from facts available during the safety proof. The assertions strengthen the safety theorem so it also implies all the intermediate properties asserted throughout the code. This transfers the asserted facts to the information flow proof.

5.2 Information Flow Policy

The information flow model of this section is essentially Denning's lattice-based model [3], but with the weaker requirement of a partial order rather than a lattice. Greatest lower bounds are useful for combining users or consumers of data, but the file system never does this. While least upper bounds are useful for assigning labels to combinations of data with different labels, the file system never initiates the combination of data items. Instead it only combines data at the request of a client; for example, the server generates directory listings which refer to files of different labels. Although the client's label may be greater than the least upper bound of the labels in the directory listing (if such an upper bound exists), the client's label suffices.

```
nonInterference :: "config ⇒ σ machine ⇒ (α, σ) m ⇒ bool"
"nonInterference c s x
 ≡∀obsL w t
   . obsL < numLevels c
    ⟶ w ∈ input-set c
    ⟶ (∃u. outputEq c obsL
                (runM x (purge c obsL w) u s)
                (runM x w t s))"

theorem stepNonInterference
: "⟦goodConfig c; goodState c s⟧
   ⟹ nonInterference c s (repeat n (step c))"
```

Fig. 3. Notion of Noninterference

Goguen and Meseguer's notion of (transitive) non-interference [6] essentially captures the desired property and is widely accepted as a reasonable policy. Intuitively two imaginary copies of the system run in parallel where one copy receives only a subset of the inputs. The system respects Denning's notion of information flow if for each label the outputs of both copies look the same from the viewpoint of an observer associated with that label when one copy of the system sees only inputs from labels less than or equal to that of the observer. That is, the output was not influenced by allegedly unobservable inputs.

The formulation of a policy, however, depends on the representation of the system to which the policy applies. While most existing notions of non-interference assume an external stream of labeled events that repeatedly prod the system into action, this work's model of computation more closely matches that of a traditional program. That is, the program executes without any external prodding and of its own volition consumes input when it wishes. This meshes well with the interprocess communication mechanisms and scheduler provided by the underlying real-time kernel. Another difference is that the step function processes multiple requests from multiple sources with different labels.

These differences in the computational model require adjustments to the policy, as shown in figure 3. Rather than dropping input actions that must remain unobservable, the purge function filters the input oracle so any input bytes read from unobservable sources are zero. Conveniently, zeroing inputs to steps rather than skipping steps fits better with the simple scheduling algorithms found in real-time kernels and also eliminates stuttering from the proofs.

In addition to adjusting the input oracle, the policy also adjusts the disk timing oracle. The cause for the adjustment is that requests to read data causes a delay in the processing of other disk requests. While the deployed system may rely on pragmatic solutions to cover this delay, the model accommodates the issue by weakening the policy. The purged execution trace may chose any arbitrary disk timing oracle. Thus, the theorem says that any differences in system behavior could potentially be caused by the disk drive running at an unusual speed.

The theorem at the bottom of figure 3 states that the monadic computation that runs the step function an arbitrary n times supports the non-interference property at the top of the figure for the given initial state. The non-interference property states that for any observer label obsL and input oracle w and timing oracle t if the label is valid and the input oracle is valid, then running the monadic computation from the start state on both the filtered and unfiltered input oracles sends the same trace of outputs to observers with the given label.

6 Proof Technique

The structure of the information flow proof is fairly typical, although it requires some adjustment for the monadic framework. This section describes the proof structure, the similarities to Ohëimb's variation [13] of the unwinding lemmas, and some key distinctions.

The predicate outputEq in figure 3 is not inductive. Thus, the first step in the proof is to strengthen the induction hypothesis. A stronger relation viewEq insists not only the outputs are equal but the pieces on internal state which may eventually indirectly influence the output must also be equal. Proving that viewEq is in fact a stronger relation than outputEq is traditionally called output consistency. Defining viewEq as the conjunction of outputEq and a predicate on internal states makes this proof trivial.

The viewEq relation is an equivalence relation. The fact that it is reflexive starts the induction over two traces of states by asserting the initial state is viewEq to itself. Higher-order logic supports Harrison's technique of proving that for any equivalence relation R, $\mathsf{R}\ \mathsf{x}\ \mathsf{y} = (\mathsf{R}\ \mathsf{x} = \mathsf{R}\ \mathsf{y})$. This changes a custom equivalence relation into Leibniz equality, which Isabelle's simplifier can then use to perform equational rewriting with viewEq and automatically make use of symmetry and transitivity.

The bulk of the work is proving that stepping both traces one step preserves the viewEq relation. There are two common cases to consider. Either the monadic action affects only state that is observable and does not depend on any unobservable state or the action affects only unobservable state. The first corresponds to Ohëimb's weak step consistency while the latter corresponds to local respect. One difference is that while action-based frameworks case split on the level associated with an external action, in this system the case split depends on (some argument to the function that generates) the monadic computation or some piece of the state space or both.

For the first case where the action modifies observable state, the proof decomposes the lemma over the monadic bind operation. The key is that in the observable case, the values produced by the first computation in each trace are Leibniz equal. This causes the application of the second argument of the bind operator to produce equal computations in each trace. This enables the proofs about the two subcomputations of the bind operation to combine into a proof about the whole computation.

For the second case where the updated state is not observable, the proof reduces the reasoning to focus on only a single trace. The updated state in each trace is viewEq to the state before taking the step. Thus, the equivalence of the final states of two traces follows from transitivity and the equivalence of the traces' starting states. Instantiating quantifiers is easier when there is only one expression in the assumptions list of the proper type, so reasoning about each trace independently improves automation.

7 High-Level Model

The process of designing the high-level model of the step function mentioned in section 3 is somewhat interesting, but largely out of scope. This section highlights a few techniques.

The initial design consisted of a direct-style program that reads a request, performs the appropriate blocking I/O operation, waits for the response, and returns the result. This program suffered from several problems. First, higher-order logic cannot directly state the desired system property in terms of this interface. Second, the program processes only one request at a time.

Inverting the control flow [10] through continuation passing style [12] solves both problems. Each formerly blocking I/O operation instead aborts the current continuation and invokes a callback when the I/O completes. To handle multiple requests, a continuation delimiter [4] surrounds the loop body. The delimited continuations capture and abort only the loop body rather than the entire remaining computation. Thus, the program immediately continues with the next iteration which initiates more asynchronous I/O requests. This, in essence, introduces cooperative threading.

Another benefit of inverting the program is that the I/O moves to the end of the function rather than the middle. This aligns the boundaries of the step function with the process's time slice. Thus, the policy is in terms of observations made when the program is blocked for I/O and other processes are running rather than at some arbitrary point.

8 Low-Level Model

From the high-level model of the step function, judicious use of Isabelle's equational rewriting produces a lower-level model. While the high-level model is already close to source code in a language with good support for functional programming, features such as dynamic memory allocation present challenges for languages typically used in systems programming.

Transforming the representation of the step function relies on a synonym for equality "$\mathsf{eqP} \equiv \mathsf{op} =$" which delays the instantiation of schematic variables. Specifically, folding the definition of eqP in the lemma "$\mathsf{step} = ?\mathsf{x}$" prevents the simplifier from immediately solving the goal by reflexivity.

A variety of optimizations are readily available "for free" just because the development occurs within a theorem prover. Unfolding constant definitions performs

```
enqueueCont :: "label ⇒ cont ⇒ unit m"
dequeueCont :: "label ⇒ cont option m"
```

Fig. 4. Enqueue and Dequeue Operations

both function inlining and constant propagation optimizations. The only mild issue with inlining is that some definitions require η-expansion before the simplifier will use them. Rewriting with the monadic laws are almost free, since the proofs are unproblematic and should be proven anyway.

Although assertions made facts proven during the safety proof available during the non-interference proof, they serve no purpose in the implementation. Isabelle easily removes the assertions via rewriting. Under the assumption that running an assertion does not result in the unspecified error state, running the assertion is equivalent to running return (). Isabelle can use this lemma along with lemmas that assertion removal is compatible with the structure of the program syntax as elimination rules in order to perform the desired syntax-directed program transformation.

The high-level representation of the step function contained two sources of dynamically allocated intermediate data structures, which Isabelle deforested in different ways. The first was relatively straight forward, while the second involved more effort.

The first structure represents incoming requests from the interprocess communication channels. The program bound the result of parsing the request to a variable and then applied a function to the variable to process the request. The parser consists of several nested conditionals which test for each possible variant of the request structure. Each leaf of the tree of conditionals constructs the appropriate request variant. The processing function first case splits on the request variant and then takes appropriate action. Inlining both functions and then distributing the processing function across the parser's conditionals replicates the processing function n times where n is the number of variants. In this case, however, each replicated instance of the case split is immediately applied to a (fully saturated) data constructor. Each of the n case statements then reduce to a single branch which is on average $1/n^{th}$ the size. Thus, the transformation decreases code size rather than increasing it and also eliminates the dynamic construction of compound data.

The second dynamically allocated data structures represent work left to be done after a DMA request completes. A naïve implementation of the high-level model would allocate such a structure and enqueue it just before sending a DMA request to a disk. The processing of the structure occurs later when the DMA completes; the structure construction and the structure destruction are in some sense far apart—even in different reactions.

To deforest the structures, consider the non-proper morphisms that operate on the state space within the monadic type. Figure 4 shows the approximate type signatures of the high-level model's operations for enqueuing and dequeueing these (defunctionalized partial continuation) structures. This interface to the queue data structure suffers from a problem similar to that of the generic state monad men-

```
enqueueContRead :: “label ⇒ ovecIndex ⇒ dvecIndex ⇒ label ⇒ nat ⇒ unit m”
“enqueueContRead qLabel ovecI dvecI responseLabel responseOffset
 ≡ enqueueCont qLabel (ContRead ovecI dvecI responseLabel responseOffset)”

enqueueContWrite :: “label ⇒ nat ⇒ unit m”
“enqueueContWrite label responseOffset
 ≡ enqueueCont label (ContWrite label responseOffset)”
```

```
queuePeekCont :: “label ⇒ cont option m”
“queuePeekCont l
 ≡ do tag ← queuePeekContTag l
      ;if readTag = tag
         then do ov ← queuePeekContReadOutBuf l
                 ;dv ← queuePeekContReadDvec l
                 ;reqL ← queuePeekContReadReqLevel l
                 ;i ← queuePeekContReadReqSlot l
                 ;return (Some (ContRead ov dv reqL i))
         else if writeTag = tag
              then do reqL ← queuePeekContWriteReqLevel l
                      ;i ← queuePeekContWriteReqSlot l
                      ;return (Some (ContWrite reqL i))
              else return None”

lemma dequeueCont-def2
: “dequeueCont l
   = do k ← queuePeekCont l
        ;dequeueDropCont l
        ;return k”
```

Fig. 5. Modified Enqueue and Dequeue Operations

tioned by Chen and Hudak [1]. Specifically, the state representation inside the monadic datatype contains references to values passed in from the outside. [2] Since the interface to the monadic datatype fails to enforce linearity, the operations do not model an imperative implementation. Transforming the interface eliminates this difficulty.

The transformation to the monadic state's interface involves rewriting the model with the new queueing operations shown in figure 5. The top portion of the figure defines new operations in terms of the old ones. The lower portion proves alternate definition-like equations for the old operations in terms of the new ones. Substituting the new equations for the old operations throughout the program produces a new program representation that uses only the new interface. Inlining the original definitions for the queueing operations into the function bodies of the new operations updates the interface exposed by the monad.

[2] Returning a reference to a value also poses problems.

```
int main()
{initState();
 for (;;){
#include “step.c”
}}
```

Fig. 6. Implementation Main

This new interface to the state's queues exposes only the components of the structures, not the structures themselves. There is only ever one reference to each structure, namely the reference from a node of the queue. The linear nature of the references ensures the faithfulness of an implementation based on imperative updates. Thus, the transformation eliminates the need to dynamically allocate new structures when enqueuing.

While linearity eliminated one source of dynamic memory allocation, the transformation introduced another. The wrapper equations in the lower half of figure 5 all allocate data structures. The location of the allocation, however, moved to a different portion of the code and occurs in a different reaction. In fact, by moving allocation sites from enqueuing to dequeuing, the allocations are right where the processing—and hence destructuring—of the data occurs. Isabelle's equational rewriting can now easily deforest the data structures.

9 Implementation

While proving that a model of a system enforces a desired policy eliminates certain design flaws, it does not guarantee that a deployed implementation behaves accordingly. For this, the implementation must correspond to the model of the program and the library code must properly implement the primitive monadic effects. This section describes the connection between the model and the source code. Section 10 discusses the primitives.

Isabelle's meta language serves as a foundation for code generation. The meta language—which is normally used to write tactics—can inspect the abstract syntax trees representing the higher-order logic formulae and is also a general purpose programming language. Since the low-level model of section 8 already looks like imperative-style source code, the code generator simply traverses the abstract syntax tree via pattern matching and recursion. At then end of the traversal the code generator simply writes the resultant string to a file, which an existing C compiler compiles. While C is not ideal for building reliable software, constructing a verified or verifying compiler all the way to machine language was outside the scope of the project. This may cause difficulties for an evaluation, since some organizations (such as the United States Federal Aviation Administration) who perform evaluations of C code require that the code to be tested using multiple compilers and that the developers disassemble the object code and explain how it relates to the compiler's input.

The generated implementation corresponds directly to the low-level model of the `step` function. Even with the library that implements the monadic primitives, this is not a complete implementation. A small amount of code must setup the initial state and a loop must repeatedly call the generated `step` function. Figure 6 is essentially the main program.

10 Testing

What is the relation between proof and testing? Some people question the value of formal methods, while others assume that proving software correct "once and for all" eliminates the need for testing the software on specific inputs. This section mentions the relation between testing and mathematical proof, two rationale for testing, and the specific testing methods employed by the project.

Mathematical proof is based on deductive reasoning—deducing more complex facts from presumably simpler ones. While deductive proofs provide value by both reducing the complexity of and clarifying the assumptions, assumptions remain. For the development presented so far, the most significant remaining assumption is that the abstract machine model of the non-proper morphisms matches the behavior implemented by the hand-written C code, the C compiler, the separation kernel, the microprocessor (hardware and microcode), and the peripherals. This layer requires validation testing to ensure it matches the model.

Since testing is typically less expensive than formal proof, testing establishes an acceptable level of assurance for less critical properties. For example, if the system suddenly became as unresponsive as a brick then it would still succeed in its primary mission of not leaking information.[3] This behavior, however, would likely cause consternation for the users. Thus, the tests cover several basic correctness properties. The correctness tests are unlikely to pass if the abstract machine is broken; hence they cover validation testing also.

Haskell's QuickCheck [2] tests extra assertions about the implementation. Although the tool is designed to test Haskell programs, the test suite uses Haskell's foreign function interface to make interprocess communication requests that interact with the block access controller implementation. By stating the property in Haskell that writing an arbitrary block of arbitrary bytes and then reading it produces the same block of bytes, QuickCheck automatically generates many randomized requests and tests the implementation. Quickcheck also tests the property that concurrently writing blocks with identical arbitrary bytes and reading blocks from a different interface with a greater label results in reading blocks with identical bytes.

In addition to this property-based testing, the test suite also performs model-based testing. For this effort another model written in Haskell captures just the functional behavior of the system. This model simply represents the disk as a finite map of blocks of bytes, with read and write requests immediately performing the corresponding actions (i.e. no delay due to DMA). A QuickCheck assertion states that arbitrary sequences of arbitrary read and write requests processed by

[3] The relative importance of these properties reflects the biases of the author's particular clients.

the implementation produce the same results as processing those same requests in the Haskell model.

11 Future Work

The experience with this project was overall quite positive, but room for improvement remains. The integration between the testing and verification tools could have been tighter. Model to implementation correspondence issues remain. Finally, this section suggests a research challenge in modeling reactive systems.

While the primary model is in Isabelle, the testing model is in Haskell. This involved recreating another (simpler) model in Haskell by hand. Eliminating this step and testing directly from Isabelle would be preferable, both because of work reduction and also because it would strengthen the claim that the model was tested. The Isabelle theorem prover includes a method named quickcheck which, like Haskell's QuickCheck, randomly generates test vectors. Isabelle, unlike Haskell implementations, does not include impure features such as foreign function interfaces which the implementation testing relies upon. It may be possible to use Haftmann's code generator for Isabelle [9] to produce a matching Haskell model from the Isabelle model, however, this was not available at the time.

The connection between the model and implementation remains less than ideal. For one, the code generation automatically produces C code that looks syntactically similar to the low-level model, but this is merely a syntactic translation. All the claims about issues such as the store being linear remain informal. Producing machine code for a processor with a publicly available formal model, such as the ARM processor [5] would improve the correspondence. A verifying or verified compiler for a low-level language would ease the automation of such a task. Finally, the model used natural numbers for unsigned integers. This resulted in an numeric overflow error in earlier implementations. Modeling machine word arithmetic without Isabelle's recently available machine word arithmetic library appeared prohibitively painful.

The continuation passing style and defunctionalization transformations should have been done either by a compiler or at least in a theorem prover rather than by hand. If Isabelle could express properties about not only top level functions but also about implicit continuations, the project could have potentially modeled and reasoned about the program in direct style.

12 Conclusion

In summary, it is feasible to use modern interactive theorem provers such as Isabelle to formally prove properties about low-level system software. Higher-order logic is well-suited for embedding both other programming languages via monads as well as logics customized for reasoning about the language's constructs. Interactive compilation through equational rewriting provides a nice mix between the control of hand optimization and the convenience, maintainability, and reliability of fully automatic optimization. Our notion of non-interference supports programs with non-inverted I/O and allows for variations in the timing behavior of peripherals.

Acknowledgement

John Matthews deserves credit for convincing the author to use the Isabelle theorem prover and patiently serving as a mentor in its use. His numerous discussions regarding the project were also invaluable. The author also thanks Dylan McNamee and the anonymous referees for their helpful comments on earlier drafts. Without the support of Galois project management, especially Laura McKinney, and the support of the project's sponsors this work would not have been possible. Thanks are also due to the creators and maintainers of the Isabelle theorem prover as well as its surrounding community.

References

[1] Chen, C.-P. and P. Hudak, *Rolling your own MADT - a connection between linear types and monads*, in: *ACM Symposium on Principles of Programming Languages*, 1997, pp. 54–66.

[2] Claessen, K. and J. Hughes, *Quickcheck: a lightweight tool for random testing of Haskell programs*, in: *ACM SIGPLAN International Conference on Functional Programming*, New York, NY, USA, 2000, pp. 268–279.

[3] Denning, D. E., *A lattice model of secure information flow*, Communications of the ACM **19** (1976), pp. 236–243.

[4] Felleisen, M., "The Calculi of #v-CS Conversion: A Syntactic Theory of Control and State in Imperative HigherOrder Programming Languages," Ph.D. thesis, Indiana University (1987).

[5] Fox, A. C. J., *Formal specification and verification of ARM6*, in: *International Conference on Theorem Proving in Higher Order Logics* (2003), pp. 25–40.

[6] Goguen, J. A. and J. Meseguer, *Security policies and security models*, in: *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[7] Gordon, M. J. C., *HOL: A machine oriented formulation of higher order logic*, Technical Report 68, University of Cambridge Computer Laboratory (1985).

[8] Greve, D., M. Wilding and W. M. Vanfleet, *A separation kernel formal security policy*, in: *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications*, 2003.

[9] Haftmann, F. and T. Nipkow, *A code generator framework for Isabelle/HOL*, Technical Report 364/07, Department of Computer Science, University of Kaiserslautern (2007).

[10] Jackson, M. A., "Principles of Program Design," Academic Press, 1975.

[11] Paulson, L. C., *Isabelle: The next 700 theorem provers*, Logic and Computer Science (1990), pp. 361–386.

[12] Strachey, C. and C. P. Wadsworth, *Continuations: A mathematical semantics for handling full jumps, technical monograph PRG-11*, Technical report, Oxford University Computing Laboratory, Programming Research Group (1974).

[13] von Oheimb, D., *Information flow control revisited: Noninfluence = Noninterference + Nonleakage*, in: P. Samarati, P. Ryan, D. Gollmann and R. Molva, editors, *Computer Security – ESORICS 2004*, LNCS **3193** (2004), pp. 225–243, http://ddvo.net/papers/Noninfluence.html.

[14] Wadler, P., *The essence of functional programming*, in: *ACM Symposium on Principles of Programming Languages*, 1992, http://homepages.inf.ed.ac.uk/wadler/topics/monads.html.