Running the Manual: An Approach to High-Assurance Microkernel Development

Philip Derrin Kevin Elphinstone Gerwin Klein David Cock
National ICT Australia, and
School of Computer Science and Engineering,
University of New South Wales
{philip.derrin|kevin.elphinstone|gerwin.klein|david.cock}@nicta.com.au

Manuel M. T. Chakravarty
School of Computer Science and Engineering
University of New South Wales
chak@cse.unsw.edu.au

Abstract

We propose a development methodology for designing and prototyping high assurance microkernels, and describe our application of it. The methodology is based on rapid prototyping and iterative refinement of the microkernel in a functional programming language. The prototype provides a precise semi-formal model, which is also combined with a machine simulator to form a reference implementation capable of executing real user-level software, to obtain accurate feedback on the suitability of the kernel API during development phases. We extract from the prototype a machine-checkable formal specification in higher-order logic, which may be used to verify properties of the design, and also results in corrections to the design without the need for full verification. We found the approach leads to productive, highly iterative development where formal modelling, semi-formal design and prototyping, and end use all contribute to a more mature final design in a shorter period of time.

Categories and Subject Descriptors D.4.5 [*Operating Systems*]: Reliability—Verification; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; I.6.3 [*Simulation and Modelling*]: Applications

General Terms Languages, Design, Documentation, Verification

Keywords Operating systems, Haskell, rapid prototyping, executable specification, Isabelle/HOL, monads, formalisation, verification

1. Introduction

The development of modern high-end microkernels for embedded systems suffers from opposing requirements: the need for high-performance and the demand for high-assurance. On one hand, constraints on physical resources (time, speed, and power) require tight control of clock cycles and memory footprint. For example, it is common to hand-tune data-structure layouts and locations to minimise the cache footprint of common operations. Ultimately, these constraints necessitate the use of systems programming languages, such as C and C++. On the other hand, demands for high-assurance (safety and security) require a rigorous specification of the kernel API and its security and safety properties as well as guarantees that the implementation meets this specification. Ultimately, these demands necessitate the use of both formal specifications and theorem provers. Unfortunately, systems programming languages and rigorous formal methods are far apart, leading to slow development and to compromises in performance and safety. A development cycle that includes both formal modelling and a C implementation of the model in each iteration is slow. However, without an implementation, few insights can be drawn about the potential performance of a large complex model. Whether a data structure is five bytes or four makes little difference to correctness, but can make a dramatic difference to performance.

To combine high performance and high assurance, kernel API development has to rapidly iterate through many cycles, while progressively refining both the specification and implementation. Moreover, the kernel prototype must be able to execute real user-level code to evaluate the practical adequacy of the design.

In this paper, we propose a novel development model that relies on functional programming to implement an executable specification of the kernel API. This specification is: (a) at a high level enabling fast turn around, (b) able to execute real user-code without getting bogged down in hardware details, (c) amenable to semi-automatic extraction of a formal specification for Isabelle/HOL, and (d) the basis for a final implementation of the kernel in C/C++.

Previous experience with formally verifying a portion of the L4Ka::Pistachio microkernel [20] convinced us that it is desirable to completely formally specify a kernel interface prior to its refinement into a complete implementation [19,30]. After all, embarking on a resource-intensive complete kernel implementation, that is difficult to adapt, prior to determining the required formal properties of the system is at best risky. Additionally, reverse-engineering a formal specification from a real implementation is time consuming.

The recent House and Osker kernels [12] (in Haskell) and the Hello kernel [9] (in Standard ML) showed that functional languages are sufficiently mature for kernel development on bare metal. However, independent of the implementation language, bare metal implementations require a lot of attention to hardware details (such as bootstrapping, page table management, and interrupt handling), and time consuming debugging in a hostile environment, which distracts from the goals of API design and formalisation. Also, implementations in high-level languages are limited in their ability to implement, rather than model, the desired behaviour of the kernel. For example, the seL4 kernel never dynamically allocates memory, but when using Haskell on bare metal, it is impossible to avoid allocating space in the Haskell runtime's heap.

Haskell'06 September 17, 2006, Portland, Oregon, USA.
Copyright © 2006 ACM 1-59593-489-8/06/0009...$5.00.

Hence, we propose the following alternative approach. We develop an *abstract model* of how the kernel would function in a complete system in a functional language (specifically Haskell). The kernel model responds to a specific set of events, and manipulates both its internal state and the state of the underlying simplified hardware model in response to those events. We then couple this model with a machine simulator, allowing it to respond to the actions of real user-level software in a manner that emulates a real implementation of the kernel. During our development of the seL4 microkernel, this method enabled us to rapidly develop a testable prototype of the new kernel's interface, while avoiding hardware-specific issues until we were prepared to handle them. Thus, we could construct user-level systems using the interface in parallel with the development of the kernel API specification. We used the Haskell code as informal API documentation and to semi-automatically derive a formal specification.

Our proposed development approach includes the construction of a high-performance implementation in C. However, this final phase of our approach remains as future work. In the remainder of the paper, we focus on our approach to rapid prototyping and semi-formal modelling of microkernel APIs in Haskell, and their subsequent formal modelling in Isabelle/HOL.

In summary, this paper makes the following contributions:

- a method for modelling an event-driven operating system kernel in Haskell (Sect. 2 and Sect. 3);
- a method of modelling the contents of a machine's physical memory without translating those contents to byte streams (Sect. 3.2);
- an interface between an event-driven kernel model in a functional language and an external CPU simulator, allowing execution of realistic user-level code inside the kernel model (Sect. 4);
- a method for rapidly formalising monad-based Haskell code in Isabelle/HOL (Sect. 5); and
- experiences applying the above techniques in the design of a new microkernel (Sect. 6).

We discuss related work in more detail in Sect. 7.

2. Specifying microkernels in Haskell

Prior to describing the issues in specifying an operating system in Haskell, we first introduce the class of operating systems we are working with to convey the scope of the problem. *seL4* is an evolution of second generation microkernels [23]. The microkernel is relatively small (less than 10,000 lines of C code), and aims to provide *policy free mechanisms* for the flexible construction of more complex operating systems with the microkernel as the foundation [21]. Traditional operating system functionality (such as a file system) is implemented outside the microkernel as user-level servers. Device drivers also exist as servers outside the kernel, with the microkernel only providing a mechanism for the drivers to receive hardware-generated interrupts.

The microkernel provides the mechanisms for the creation and management of address spaces and threads, and inter-thread communication. Generally, mechanisms are only included in the microkernel when the desired security properties of the system would be impossible to enforce otherwise. On rare occasions, additional mechanisms are included in the pursuit of performance.

The abstract model of the provided mechanisms aims to be as hardware independent as possible to aid in portability of software between microkernels implemented on different architectures. This aim encourages an approach of designing and implementing a new kernel in a hardware-independent manner to focus on modelling the mechanisms themselves. While an interface to the underlying hardware will be required eventually, the need to deal with the complexity of the underlying hardware might be avoided for a significant fraction of the work.

Another factor that influences the development approach is the desire for an *atomic kernel API* [8]. An atomic kernel API is an API in which all system calls appear to complete atomically from the perspective of applications. All system calls return *promptly* without being delayed by the state of other applications, and also return the correct state. As an illustrative example, promptness and correctness become an issue when a thread is non-interruptibly blocked inside the kernel, and is then operated on by a second thread. Blocking the second thread until the first is in a user-visible state would not be prompt, and delivering the last-known good state of the first thread would not be correct. Long running system calls can be supported by breaking them up into user-visible sequences of atomic operations.

An atomic API is desirable as it enables an *interrupt* style kernel execution model (also termed *event* style). Interrupt style kernels only have a single thread of control when executing within the kernel, and are implementable using a single kernel stack, independent of the number of threads currently allocated in the system. A *process* style kernel execution model, where a kernel stack is allocated per thread, would significantly increase the amount of physical memory required in a microkernel-based system. Reducing memory consumption is an important issue in memory-constrained embedded systems.

As a consequence of requiring an interrupt style kernel, the development methodology does not need to support modelling multi-threading in the kernel execution environment. Also, the limited ability of formal methods to scale in the presence of concurrency discouraged us from exploring alternative process style execution environments for the kernel. However, our approach could theoretically be applied to a multi-threaded kernel. Hallgren *et al.* describe approaches to multi-threading support for their Haskell-based kernel [12].

There are several potential approaches to the design and specification of a new microkernel within the family of kernels described above:

- A natural-language specification is easily written and read, but is prone to inadvertent ambiguity and incompleteness, and often fails to expose design issues that may have a significant impact on performance, usability, and ease of implementation of the design.
- Formal specification at an abstract level avoids ambiguity, but still may not expose issues affecting performance and ease of implementation of the design until the refinement stage. This is a particular problem for systems software, which is performance-critical and must operate in a relatively constrained environment. Also, it is difficult to evaluate the usability of a microkernel interface for building complete systems based on that interface, until such a system has actually been built.

  Also, the tools and techniques used for developing formal specifications are quite different to those typically used for systems software, so there is a high cost of entry for many kernel developers.
- Implementation in a low-level language exposes problems with the design, but involves considerable development and debugging effort, particularly if design problems are discovered late in the implementation and lead to significant changes being made. Also, the result is not useful as a readable specification, as the expected behaviour is rarely made clear by low-level code

— especially as such code often contains bugs or diverges from existing informal specifications.

- Implementation in a high-level language with well-defined and safe semantics is a good compromise between the previous two approaches. For example, the Osker kernel [12] is written in Haskell. This approach produces an implementation which is easier to reason about than one in a low-level language; however, it is limited by the tendency of high-level languages to depend on complex runtime systems that are not ideal for use in a stand-alone kernel. This may impose restrictions on the system that are not present when using low-level languages (e.g., the inability to exclude dynamic allocation of kernel memory).

We believe our approach combines the advantages of the last three items while avoiding the problems of all four. In particular, we avoided the challenges of a bare-metal implementation by developing a kernel prototype that runs on an abstract, non-platform-specific model of the concrete hardware with greatly simplified features compared to those of the raw hardware. The prototype is written in Haskell, using the State monad to maintain a resemblance to traditional kernel implementation languages (Sect. 3) while taking advantage of Haskell's pure functional semantics to ease formalisation (Sect. 5). We are able to rapidly expose usability issues by developing user-level programs in parallel with the API, and running them directly on the executable specification using a simulator (Sect. 4). The rapid prototyping and easy formalisation has allowed us to explore many design alternatives and expose several problems with the API during the development process (Sect. 6).

3. Kernel Modelling

This section introduces a general interrupt-style kernel model. Some aspects have a bias towards seL4, but the general approach, and in particular, the monad-based interface certainly extends to other kernels.

3.1 Events and State

An operating system kernel is an event-driven system. The host machine spends the majority of its time executing user-level programs. When an event occurs — such as an interrupt triggered by a timer or an external device, a fault during program execution, or an explicit invocation of a kernel service — the program is interrupted and control is transferred to the kernel, which makes appropriate changes to the machine's state in response to the event.

It follows that the kernel can be modelled as a function which transforms the state of the modelled system in response to an event. A running system may be simulated by repeatedly applying this function to a sequence of events. The kernel model function might have a type signature such as:

```
kernel :: Event → System → System
```

The Event type represents all of the possible events that the kernel may encounter, including system calls, virtual memory faults, hardware interrupts, and so on:

```
data Event
      = SyscallEvent Syscall
      | VMFault Word Bool
      | ExecutionFault
      | TimerInterrupt
      | Interrupt Int
```

The System type represents the entire state of the system, including the CPU registers, physical memory, and I/O devices. In our model of seL4, this structure maps physical memory addresses to the contents of the corresponding memory regions. Objects in the memory model are ordinary high-level Haskell data objects, rather than streams of bytes; this eases access to their contents from Haskell code, and also eases detection of invalid casts of physical memory pointers.

The remainder of this section describes in further detail the data structures we used to model the state of a seL4 kernel. We discuss methods of generating events for the model to process in Sect. 4.

3.2 Kernel State

Given the type of `kernel` above, the kernel is obviously a state transformer, and hence, conveniently represented as a monad. This choice is reaffirmed by the need for recoverable exceptions, which are detailed in the next subsection. In fact, we will see that we want to distinguish between code that may raise recoverable exceptions and code that does not have that liberty. Hence, it is worthwhile to use monad transformers as provided by the MTL in the Haskell Hierarchical Libraries [14].

We use the MTL's parameterisable State monad as the base monad for all parts of the kernel code that accesses or modifies parts of the System state. To this end, we instantiate the generic State monad with the System state to obtain the type Kernel thusly:

```
type Kernel = State System

kernel :: Event → Kernel ()
kernel event = do
            ...
```

In the seL4 model, System mainly maintains a mapping of physical addresses to seL4 kernel objects stored at those addresses. Other, lower-level details of the system, such as I/O devices, caches, and memory management units, are left out of the model (or modelled elsewhere, as discussed in Sect. 4). The seL4 kernel's user-level API exposes information about allocation of physical memory; therefore we must model the placement of kernel objects in physical memory accurately. However, it is not essential to accurately model the layout of data inside the kernel objects, as long as we are convinced that the data will fit in the allocated space.

Objects stored in the physical address space model are those which are allocated dynamically to support the abstractions provided by the kernel to user-level processes. The object types used in seL4 are shown in Fig. 1. There are types representing thread control blocks (TCB), entries in virtual address space structures (CTE, for a capability table entry), and endpoints used to coordinate inter-process communication (Endpoint and AsyncEndpoint). Regions allocated for use as virtual memory pages are marked as such, yet contain no real data (UserData); the contents of virtual memory pages are stored separately (Sect. 4.2). There are also objects that represent statically allocated kernel data and code (KernelData).

The interface for accessing the physical memory model is shown in Fig. 2, where PPtr is an abstraction of pointers into physical memory. It constrains the types that may be stored in and extracted from physical memory to those of the class Storable, which defines the size of physical-memory objects and the actual storage and retrieval primitives. Note that this class is distinct from the existing class Storable in Haskell's standard foreign function interface, though it has a similar purpose.

The representation of physical memory in the kernel model is a critical aspect of the design and, indeed, the representation we finally chose was the third we implemented, and even more were considered and rejected. The reasons for using it were in part specific to seL4, but many will apply to other kernels as well:

- The most realistic model of physical memory is simply an array of bytes. However, such a model requires any stored high-level data structures to be converted to and from streams of bytes, which loses information about the type of the stored objects. While such type information is obviously not available in a real

```
data KernelData = KernelData

data UserData = UserData

data Endpoint
     = IdleEP
     | SendEP { epQueue :: [ThreadPtr] }
     | RecvEP { epQueue :: [ThreadPtr] }

data AsyncEndpoint
     = IdleAEP
     | WaitingAEP { aepQueue :: [ThreadPtr] }
     | ActiveAEP { aepData :: Word }

data CTE = CTE {
     ... object reference ...
     }

data TCB = Thread {
     ... thread state ...
     }
```

Figure 1. The objects that may be stored in the seL4 physical memory model.

```
getObject :: Storable a ⇒
     PPtr a → Kernel a

setObject :: Storable a ⇒
     PPtr a → a → Kernel ()

createObjects :: Storable a ⇒
     PPtr a → Int → a → Kernel Int

deleteObjects :: PPtr a → Int → Kernel ()
```

Figure 2. Physical address space access functions.

system, retaining it eases detection of incorrect use of physical pointers in kernel code.

Implementing this type of model in Haskell is undesirable. The objects would need to be stored using a similar encoding to that which they would use in a real kernel. An encoding automatically generated by a Haskell compiler is unlikely to be realistic; so the conversion between high-level objects and byte streams would be done by hand-written functions, which are error-prone and require additional maintenance whenever the contents of an object change.

- The seL4 API relies on being able to store objects of one specific type either as stand-alone objects, or contained in a structure of another type — without using different means of access in each case. Specifically, the thread control block (TCB) contains capability slots, represented by the CTE type; CTE objects may also be stored directly in the physical memory model, as entries in a capability table. The seL4 capability management code relies on being able to access CTE objects without any information about whether they are contained in a TCB.

  This rules out the standard Map or Array types, as they require each stored object to be identified by a single unique index — which is not the case when one object must be accessed using its own address, and also using the addresses of several objects stored inside it.

  We solve this problem by storing physical-memory objects in a binary tree, which is indexed using the bits of a physical address, starting with the most significant bit. All objects must be stored at addresses aligned to their size (rounded up to the nearest power of two). When accessing an object of a specific type, the PSpace access functions expect to have to resolve all of the physical address bits other than those forced to be zero by the object's alignment.

  By resolving addresses one bit at a time, we can locate larger container objects when they are present, and the smaller individual objects otherwise.

- There are two approaches to storing objects of varying type in the leaf nodes of the PSpace binary tree: either by encapsulating them in a Dynamic type (also provided by the Haskell Hierarchical Libraries) or by constructing a universal type with a variant for all types which need to be stored. While the universal type is more straightforward to formalise, it requires a significant amount of boilerplate code in the instances of Storable. Dynamic makes the Storable implementation almost entirely generic. In seL4, only one type has its own implementations of the Storable methods.

Consequently, we chose to store objects in a bit-indexed binary tree and wrap them into Dynamics.

Like the Haskell foreign function interface's pointer types, the physical memory pointer type (PPtr) is parameterised with the type of the object it points to. This imposes some restrictions on the implementation that we, as kernel programmers accustomed to C, did not initially expect. For example, one of the kernel's object types — CTE, the capability table entry — contains a physical pointer to another kernel object. This pointer cannot be accessed independent of the type of the object it points to; so several small parts of the capability management code have to be implemented once for each kernel object type, even though their behaviour is always the same.

3.3 Errors and Faults

A microkernel will often encounter error conditions during normal operation. For example, a user program may send a request to the kernel which is invalid, or which the program does not have the right to perform. When such a condition is detected, the kernel will typically interrupt the processing of the current event and send some indication of the error back to user level. These are distinct from errors caused by programming mistakes or invalid states within the kernel itself.

Our model of the seL4 API defines several classes of error:

1. *faults*, which generate notification messages to user-level fault handlers;
2. *system call errors*, which happen when a user program performs a system call with invalid or incorrect arguments, and which return an error code to the caller;
3. *lookup failures* while attempting to access a capability or virtual address, which are converted to either faults or system call errors depending on the context of the failure;
4. *fatal errors*, which are caused by bugs in the kernel or invalid system states.

Fatal errors are modelled by calling `error` or `undefined`, which are formally defined as non-terminating computations ($\perp$); in practice they abort execution of the model with an error message. In some instances, the Haskell language implicitly evaluates

⊥, such as failed pattern or guard matches; such occurrences are also fatal errors for the kernel model and have the same effect as explicit `error` or `undefined` calls.

The three classes of recoverable error may only occur in specific sections of the kernel code. Our model isolates these areas of code by transforming their monads with the ErrorT monad transformer. This transformer, like StateT, is defined by the MTL; it adds the ability to throw exception objects of a specific type. We have defined an exception type for each class of non-fatal error, and apply the ErrorT monad transformer with the appropriate exception type to any kernel function that can fail.

```
data Fault = ...
data SyscallError = ...
data LookupFailure = ...

type KernelF f = ErrorT f Kernel
```

For example, the seL4 kernel defines the function `lookupCap`, which searches the current thread's capability space to find a capability (of type Capability) at a given address (of type CPtr). If it fails to find the capability, it will throw an error of type LookupFailure, describing the reason for the failure.

```
lookupCap :: CPtr ⟶ KernelF LookupFailure Capability
```

In an attempt to provide better readability for kernel programmers unfamiliar with Haskell and monads, we provide aliases of some standard monad functions with names indicating the purpose of calling them in the kernel model:

```
withoutFailure :: Error f ⇒ Kernel a ⟶ KernelF f a
withoutFailure = lift

catchingFailure :: Error f ⇒ KernelF f a ⟶
          Kernel (Either f a)
catchingFailure = runErrorT
```

We also define several functions that can be used to handle errors in common ways, such as transforming a LookupFailure into a Fault or SyscallError, or ignoring the failure and returning a null capability instead. They may also add extra context to the error that is not available in the function that generates it.

For example, the seL4 API includes several system calls that attempt to manipulate a capability address space, which is a data structure containing a sparse mapping from addresses to capabilities. If one of these system calls fails to locate a specified capability, it will generate a system call error that is returned to the caller. On the other hand, a similar failure while searching for a capability that is being directly invoked will generate a fault message that is sent to the current thread's fault handler; a failure while trying to transmit a capability through a one-way communication channel will be silently ignored when the receiver is unable or unwilling to receive the capability. Fig. 3 contains pseudocode that demonstrates this; it shows three kernel functions that all use the `lookupCap` function (described above), but do different things with the errors.

```
lookupErrorOnFailure :: Bool ⟶
          KernelF LookupFailure a ⟶
          KernelF SyscallError a


capFaultOnFailure :: CPtr ⟶
          KernelF LookupFailure a ⟶
          KernelF Fault a

nullCapOnFailure ::
          KernelF LookupFailure Capability ⟶
          Kernel Capability

handleInvocation :: CPtr ⟶ KernelF Fault ()
handleInvocation capPtr = do
     ...
     cap ⟵ capFaultOnFailure capPtr $
          lookupCap capPtr
     invokeCap cap
     ...

capCopy :: CPtr ⟶ CPtr ⟶ KernelF SyscallError ()
capCopy srcPtr destPtr = do
     ...
     srcCap ⟵ lookupErrorOnFailure True $
          lookupCap srcPtr
     ...

ipcCapTransfer :: CPtr ⟶ ThreadPtr ⟶ CPtr ⟶
          Kernel ()
ipcCapTransfer srcPtr reciever destPtr = do
     ...
     srcCap ⟵ nullCapOnFailure $
          lookupCap srcPtr
     ...
```

Figure 3. Error handling in the seL4 kernel model

4. Exercising the API

To emulate the behaviour of a real kernel, the model requires a source of events to process. We considered several possible sources for these events:

- hand-crafted lists of events;
- lists of events captured from a real running kernel; or
- events generated by a program, given the current state of the modelled host machine.

In a real system, events are triggered by particular hardware and user-level program states; that is, the sequence of events depends at least partly on the kernel's handling of previous events. When using a static event list, incorrect kernel behaviours do not have the consequences they would have in a real system (and therefore do not have the same effects on the future sequence of received events); so event lists are of limited use for simulating the behaviour of a system during API development. Therefore, we chose the third option: generating events using a program.

Early versions of the seL4 API generated events using programs written in Haskell, with a minimal model of the underlying hardware. The final, most realistic version of this Haskell-based simulator used a domain-specific language, similar in appearance to a RISC architecture's assembly language; a code fragment is shown in Fig. 4. The interpreter for this language accessed the kernel's state directly to perform thread state manipulations and system calls. It was useful for writing trivial tests of basic kernel functionality, while avoiding hardware-specific implementation details such as virtual memory management and interrupt handling. However, it was not suitable for more complex tests evaluating the practical utility of the seL4 API — such as running software ported from the existing L4 kernel.

To simulate the execution of more complex user-level programs in a system based on the seL4 kernel, we made use of existing simulators of real hardware architectures. First, we defined an abstract model of the execution context of a user-level thread

```
pingThread :: UserText
pingThread = [
        Move AR0 R0,
        LoadImmediate 0 R1,

        CompareI R1 (==0) R2,
        BranchIf R2 3,
        LoadImmediate 0 AR0,
        Syscall SysHalt,

        Move R0 AR0,
        Move R1 AR1,
        DebugPrintf "Ping␣%" [R1],
        Syscall SysSendIPC,

        ArithmeticI R1 (+1) R1,
        Move R0 AR0,
        Syscall SysReceiveIPC,
        Branch (−11)
        ]
```

Figure 4. Program fragment written in the simple assembly-like language used with early versions of seL4

(Sect. 4.1), and of the virtual memory accesses performed by a thread (Sect. 4.2). We then used Haskell's standard foreign function interface to integrate a hardware simulator with the kernel model (Sect. 4.3).

4.1 User-level Execution Context

Each user-level thread of execution has its own *execution context*. This is the subset of the system's state that a user-level program is able to manipulate directly, without invoking the kernel to do so.

From a user thread's point of view, the execution context typically consists of a set of data registers, a memory address space containing data and instructions, and a register containing the virtual address of the current instruction.

The data stored in the registers are local to the thread; no other thread may access them without an explicit kernel invocation, requiring appropriate authority. The representation of such data in the Haskell model is a data structure called UserContext.

```
data Register
    = IP | SP | AR0 | ... | AR7 | R0 | ... | R31

newtype UserContext = UC {
    ucRegisters :: Map Register Word }
```

When the kernel allows a particular thread to run, the user-level simulator loads the contents of that thread's UserContext structure from its TCB, and updates the TCB with the new UserContext before control returns to the kernel. This is similar to the behaviour of a real kernel implementation.

Data stored in memory, on the other hand, is not necessarily local to a thread. The user-level address space is virtual; its correspondance to physical memory is controlled by the kernel. It is possible for the memory mapping to overlap, or be entirely identical to, mappings used in other execution contexts. Therefore, a different mechanism must be used to model virtual memory accesses.

4.2 Virtual Memory Accesses

A virtual memory access conceptually consists of two separate operations: a kernel invocation to determine the appropriate physical address, followed by an access to physical memory, which is part of the global state of the machine.

In practice, the results of the kernel invocations involved in a virtual memory access are usually either cached in hardware using a *translation lookaside buffer* (TLB), or determined directly by the hardware using a hardware-walked mapping table. In either case, when the hardware cannot locate a mapping that a user-level thread has attempted to use, it will invoke the kernel to request that a mapping be located. Once a mapping is found, the hardware will perform the second half of the virtual memory access: the corresponding physical memory access.

To allow a hardware simulator to perform the physical memory access component of a virtual memory access, the physical memory model is separated into two parts:

- the kernel state structure stores internal kernel data as Haskell data objects, and records the type of the data stored at each memory location (Sect. 3.1); and
- the hardware simulator's state stores untyped words that are used when simulating virtual memory accesses. This includes the data read and written to physical addresses via virtual memory accesses; for some simulated architectures, there are also hardware-defined mapping tables.

The hardware simulator performs the same TLB or direct lookups as real virtual memory hardware it models, and invokes the kernel when it is unable to find a mapping. This kernel invocation is in the form of a VMFault event; the kernel responds to it by either providing the hardware with a mapping immediately, or halting the current thread until the fault can be resolved. In seL4, virtual memory faults are handled by user-level pagers, which use system calls to resolve the fault and restart the faulting thread.

It is possible to combine the two physical memory models, and store untyped words in frame-sized arrays in the kernel state; in fact, early versions of the Haskell model did so. However, this requires at least one FFI call from the external simulator to the Haskell model for every instruction executed; this made our external simulator perform very poorly. The split physical memory model only requires transitions into the Haskell model when events occur that would enter the kernel on a real system, which allows the simulator to be much faster, and more closely resembles the behaviour of a real kernel. The Isabelle translation, for which performance concerns are irrelevant, still uses a unified memory model (Sect. 5.5).

4.3 External Simulator Interface

To provide an event load similar to that of a real implementation, we developed an interface between the Haskell kernel model and an external simulator capable of executing user-level instructions compiled for a real CPU. This interface has been used to integrate the model with a modified version of the M5 Alpha simulator [25], and also with a locally-developed generic CPU simulator instantiated with a model of the ARMv6 user-level instruction set.

The simulator executes instructions until an event occurs that must be handled by the kernel model. When an event is generated, the simulator transfers the current user-level context to the kernel model's state data structure, then calls the kernel to handle the event, and finally restores the current user level context from the kernel state. The kernel may change the contents of the user level context, or select a new current context, while handling the event.

Fig. 5 shows the direction of control flow between the kernel model and the external simulator. Most of the interface consists of routines provided by the kernel and called by the simulator, as follows:

Save/Restore Context: SaveContext supplies the kernel with the current user-level context, which consists of the current register

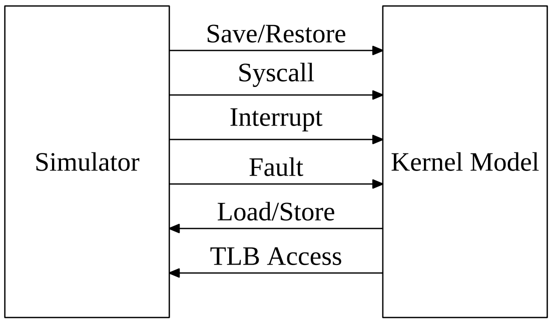

Figure 5. Structure of the simulator interface.

set and the appropriate return address for the kernel call in question; it is called prior to entering the kernel.

At the termination of a call to the kernel model, `RestoreContext` is used to set the simulator execution state. This allows the kernel to restore any execution context, e.g. to achieve a context switch.

Fault: `Fault` is used to signal a memory access fault (TLB miss) to the kernel.

Syscall: `Syscall` is the system call entry point for user-level code (triggered by an explicit user-level instruction, such as the ARMv6 `SWI` instruction).

Interrupt: `Interrupt` is used to notify the kernel of timer interrupts.

There are also a small number of callbacks used by the kernel to modify the state of the simulated machine.

Load/Store Word: These routines allow the kernel to read or write words in frames that are used for either virtual memory pages or hardware-walked page tables — that is, to access data that may also be accessed by the CPU while running user-level code.

TLB Insert/TLB Flush: These routines exist to allow the kernel to manipulate the current TLB state, typically in response to a previously signalled `Fault` or when switching to a different user-level context.

5. Formalisation of the Model

The overall goal of our project includes more than the development of the seL4 API. In future work, we plan to formally verify a high-performance C implementation of this API. The approach is to develop and design the API in Haskell to reach a highly validated and mature specification quickly and at the same time to facilitate easy formalisation of this API in the theorem prover Isabelle/HOL [26]. This formalisation can then be used as a basis for the verification of the C implementation.

Properties we are interested in verifying of the Haskell model fall into three main categories: refinement, low-level properties, and security models. With refinement, we mean that the eventual C implementation is shown to exhibit the same behaviour as the Haskell design. In this case, the Haskell model is by definition correct. Low-level properties we are interested in include termination of all system calls, kernel-object memory never being mapped to user space, and only kernel code every executing in privileged mode. Direct security properties usually are phrased with respect to a security policy. Since it is the point of a microkernel not to provide policies, but mechanisms only, we concentrate on showing that it is possible to implement specific security models instead. We are currently working on showing that seL4 can implement an abstract take/grant capability model [3, 24].

While the actual verification remains future work, we have successfully extracted a formalisation of the seL4 API in Isabelle/HOL and proved termination. This section gives an overview of the formalisation process we used and summarises some of the more interesting problems we encountered.

5.1 Translating Haskell to Isabelle/HOL

In many ways, the prototype implementation of the seL4 API in Haskell can already be seen as an executable specification. The difference to a fully formal specification is that the latter requires a complete, mechanised formal semantics and tool support for reasoning and theorem proving. The work required to gain a fully formal specification for seL4 amounted to translating a specific Haskell program into the theorem prover.

Although there are a number of attempts to translate Haskell to theorem provers automatically [1, 11, 13, 16], none of these approaches were mature enough to work for our code base which uses a number of non-trivial Haskell features and GHC extensions. Since the translated Isabelle formalisation is for human consumption, namely for later, interactive verification and refinement, it was important for us to maintain a clear 1:1 correspondence to the original Haskell code. For these reasons we chose to manually, but systematically translate the Haskell program into Isabelle/HOL. We chose the logic HOL over HOLCF which would be closer to Haskell, because one of the properties we are interested in for verification is that the kernel is bottom-free, i.e., that all system calls terminate correctly.

For the most part, this translation was purely syntactical and straightforward with regular expression matching, manual corrections, and Isabelle's interactive type and termination checking as the main tools. The interesting hurdles we encountered are the topic of the next four subsections.

5.2 A Logic of Total Functions

HOL is a logic of total functions and is as such not suitable to express the semantics of Haskell directly. It is however suitable to describe the semantics of Haskell functions that always terminate and that do not make essential use of laziness. The seL4 implementation consists of such functions.

Note that our goal is mainly formalisation, not translation of every language construct. We are free to change for instance `zip xs [1..]` into the equivalent `zip xs [1..length xs]` and thus avoid formalising laziness in all generality as a full translation mechanism would have to.

Danielsson et al [5] show that partiality does not matter if the program is shown to terminate. Since Isabelle/HOL requires a proof of termination for every definition that is entered, the translation process itself already ensures termination and we have thereby already proved a first theorem about the kernel: all API calls terminate.

Almost all of these termination proofs were automatic (the definitions being expressed as either simple abbreviations or with primitive recursion), and the rest had easy measures such as the number of bits still to process. We have to admit to taking the easy way out at one instance, though. The algorithm in question follows pointers in the data structure that models physical machine memory. We have shown a similar mechanism to terminate in the pilot study [18], but, since we still expect changes from the ongoing validation of the seL4 API in real systems, we wanted to avoid deep proofs at this stage. Instead, we observed that the set of machine words is finite and that traversing the tree will visit each pointer at most once. This termination criterion was easily accepted by Isabelle.

5.3 Monads

As shown above, the Haskell implementation of seL4 uses monads heavily. Isabelle does provide single parameter axiomatic type classes, but it does not provide constructor classes, and can hence not express monads in the traditional abstract fashion.

It is, however, possible to define concrete monads in Isabelle. The seL4 implementation uses two main monads: a state transformer (`Kernel`), and a state transformer with an exception monad on top (`KernelF`). They are easily defined in the same way as their Haskell counterparts.

It was easy to prove in Isabelle that the monad laws hold for all of the instantiations, and it was also not hard to provide a slightly modified `do`-notation where `do x ⟵ f; g x od` stands for `bind f (λx. g x)`. In the absence of overloading that a type class would have provided, we provide a different `do`-notation for each of the instantiations (`do` and `doE`). This does introduce a small notational overhead, but we found that this in fact made specification clearer than the original Haskell code because with Haskell's nested `do`-blocks it was often not obvious in which monad the operations are performed.

Fig. 6 shows a typical example of translated monadic code and demonstrates how some of the more complex Haskell case patterns are resolved in Isabelle/HOL.

Haskell:

```
activateThread = do
  thread ⟵ getCurThread
  state ⟵ getWaitState thread
  case state of
     NotWaiting ⟶ return ()
     WaitingToSend { pendingReceiveCap = Nothing } ⟶
        doIPCTransfer thread (waitingIPCPartner state)
     WaitingToReceive {} ⟶
        doIPCTransfer (waitingIPCPartner state) thread
     _ ⟶ error "Current␣thread␣is␣blocked"
```

Isabelle/HOL:

```
activateThread ≡
do thread ⟵ getCurThread;
   state ⟵ getWaitState thread;
   case state of
     NotWaiting ⇒ return ()
   | WaitingToSend eptr badge fault cap ⇒
       if cap = None then
         doIPCTransfer thread (waitingIPCPartner state)
       else
         arbitrary
   | WaitingToReceive eptr ⇒
       doIPCTransfer (waitingIPCPartner state) thread
   | _ ⇒ arbitrary
od
```

Figure 6. Typical monad code translation

For specification purposes, this concrete treatment of monads proved fully adequate. The main disadvantage is that we cannot reason abstractly about monads just in term of monad laws, which could lead to duplication of theorems. So far this did not turn out to be a problem. We mostly had to reason about the behaviour of the state monad, which involved lemmas specific to state monads, not lemmas about monads in general. Scalability was not a problem. For some programs it might turn out inconvenient to not have monad transformers available as such, but only their results. Applying significantly more than two transformers is unlikely to occur in practice, though.

```
axclass tf_byte < type
to_byte :: 'a::tf_byte ⇒ word8 list
from_byte :: word8 list ⇒ ('a::tf_byte × word8 list) option

axclass storable < tf_byte
from_byte (to_byte x @ xs) = Some (x, xs)
```

Figure 7. The axiomatic type class `storable` in Isabelle/HOL.

```
(f1 -- f2) bs ≡
let r1 = f1 bs;
    (x, r2) = case r1 of None ⇒ None
                       | Some (x, xs) ⇒ Some (x, f2 xs)
in case r2 of
     None ⇒ None
   | Some (y, ys) ⇒ Some ((x, y), ys)

x ▷ f ≡ case x of None ⇒ None
                | Some (y, xs) ⇒ Some (f y, xs)
```

Figure 8. Combinator and extractor for `storable`.

5.4 Dynamic

The Dynamic extension of GHC to Haskell98 allows a limited form of type casting that Isabelle/HOL does not provide: automatic conversion of monomorphic types to the type Dynamic and back. As described above, this extension is used in the kernel implementation to model physical memory.

We do not represent Dynamic in Isabelle/HOL directly, but instead implement the type class Storable (we could shift this to Typeable, but without immediate gain in this specific application). We chose a concrete type that is large enough to support an injection of all storable objects: `word8 list` where `word8` is the type of 8 bit machine words. This choice was arbitrary, we could just as well have chosen natural numbers or anything else large enough. We picked `word8 list`, because we already had some of the infrastructure for encoding/decoding other types into it available from our work on a memory model for C pointers [29]. The difference here is the lifting of these encodings to more complex data structures by using parser combinators.

We avoided encoding objects into byte streams in Haskell, because it is error prone and hard to maintain. Here the situation is different. We do not need to adhere to any specific layout of data and on instantiating a type to class Storable, we need to *prove* the defining axiom of the class. The prover will alert us if something breaks because of subsequent changes.

In Fig. 7 we use the class `tf_byte`, a subclass of Isabelle's default `type`, to restrict the type of the two overloaded constants `to_byte` and `from_byte`. The subclass `storable` introduces the defining axiom. The constant `from_byte` has a slightly more complex type than might be expected, because we are interested in what remains of the stream when we have read an object (`@` is the append operator). Fig. 8 defines a combinator and an extractor.

These can then be used to build up more complex types from existing ones. Fig. 9 for example shows how the datatype used to model capability rights is introduced if we have already proved that `bool::storable`. Type inference and overloading save us from specifying which `to_byte` and `from_byte` are to be used. We only need to give the structure of the encoding.

We have shown machine words, booleans, natural numbers, the option (in Haskell Maybe) type, lists, and functions to be instances of this class. Functions can be encoded as long as their domains can be shown to be finite enumerations. This is done by iterating over the domain and encoding only the range. We found this approach

```
datatype cap_rights = CapRights bool bool bool bool

to_byte (CapRights b1 b2 b3 b4) =
to_byte b1 @ to_byte b2 @ to_byte b3 @ to_byte b4


from_byte bs ≡
(from_byte -- from_byte -- from_byte -- from_byte) bs ▷
(λ(b1, b2, b3, b4). CapRights b1 b2 b3 b4)
```

Figure 9. Example for type class `storable`

to scale well beyond primitive types; once these were defined, the build-up of all other storable data types and records in the kernel was swift, and the instantiation proofs automatic.

5.5 User-level Execution

The outside interface of the kernel in the formalisation is the same as the one described in Fig. 5 in Sect. 4.

Since performance of executing user programs is not an issue in the formalisation, we can treat virtual memory accesses as normal system events which access the kernel's state rather than a separate hardware state (Sect. 4.2). Because the rest of the Haskell code uses the external simulator interface for this, we need a mapping between the external simulator functions and system events: *loadWord* and *storeWord* correspond to the read and write events. The functions *tlbInsertEntry* and *tlbFlushAll* can just map to the identity on the system state — they affect the external simulator only, not the kernel. The register part of the user state remains a direct translation of the corresponding Haskell code.

5.6 Next Steps

The next steps in the formalisation branch of this project are in two directions.

On the one side there is the creation of a more abstract, possibly non-executable and non-deterministic specification that is shown (by proof) to be an abstraction of the translation. This specification can then be used for easier proofs of safety properties and security properties that are stable under formal refinement.

On the other side is the high-performance implementation of the seL4 API in C and the formal refinement of the current executable specification towards this implementation.

6. Experience

It is difficult to quantify the advantages of a design methodology, especially in its first application. However, we can make qualitative observations based on our experience with the approach.

6.1 API Design Evolution

When we began work on the executable specification, we had plans for several new features of the seL4 microkernel that would give it the security properties we desired. However, we had no concrete designs for these features. Our approach allowed us to rapidly build prototypes of proposed designs, concurrently testing user-level code in the simulation environment and modifying the kernel model to address issues raised by the tests.

The major changes relative to L4 that we wished to explore included:

- Access control for inter-process communication (IPC). In L4, there is a global namespace for addressing messages, and there have been several unsuccessful attempts to provide secure and efficient mechanisms for restricting its use. The goal for seL4 was to use local namespaces for messaging instead, allowing restrictions to be imposed by simply limiting the set of addressable IPC partners.

```
echoThread :: [State UserContext]
echoThread = [
    do   --- Save the endpoint cap and wait for a message
         Just ep ← getCR
         setCapVar ep "ep"
         trace "Echo thread started" $
             return $ Just $ CapRead ep ,
    do   --- Send a message to the endpoint
         Just ep ← getCapVar "ep"
         n ← getMR 1
         setMR 0 0
         clearCR
         trace ("Echo " ++ (show n)) $
             return $ Just $ CapWrite ep ,
    do   --- Wait for another message and loop
         Just ep ← getCapVar "ep"
         setIP 1
         return $ Just $ CapRead ep
    ]
```

Figure 10. Part of a user level program using the state monad.

- Kernel resource management. L4 kernels have limited pools of kernel memory, and little or no accounting for use of that resource. This leaves them vulnerable to denial-of-service attacks. The goal for seL4 was to develop a mechanism for managing kernel resources from user-level servers, including delegation of resource management to clients.
- Kernel invocation mechanisms. L4 kernels restrict certain system calls to *privileged* threads. We desired a more flexible mechanism which would allow user level servers to implement system call access control policies, rather than having a fixed policy in the kernel.

One advantage of our approach in exploring these areas of the design is that there is no need to completely implement the kernel before beginning to test the new design. We focused first on the new IPC access control mechanisms. We were able to develop user-level code testing those mechanisms before the model implemented any virtual memory or capability address spaces, and before it realistically encoded system call arguments. For example, part of the first IPC test program — a thread which repeatedly receives messages through one IPC endpoint and forwards them to another — is shown in Fig. 10. This program is written in a simple Haskell-based user-level environment in the State UserContext monad, which we used in the early stages of the model's development; it makes use of an arbitrarily large user-level register set, directly possesses opaque capability objects rather than using references to them, and does not perform virtual memory accesses.

Also, once we started adding new kernel services, we were able to develop them gradually, passing through intermediate stages that would be difficult or impossible to implement on bare hardware. For example, the user-level management of capability and virtual memory address space structures, which presently uses a multi-level guarded page table [22], began as a simple set of operations on a large flat array of mappings — an impractical structure in a bare-hardware implementation, but no problem in our abstract model.

The incremental development process is still in use: our simulation environment uses multi-level guarded page tables for virtual memory address spaces, independent of the simulated architecture. This would be possible on, for example, a MIPS or Alpha bare-hardware implementation, but not on the ARMv6, which defines a specific translation table format for which we have not yet specified an interface. We perform most of our testing on an ARMv6 simulator, but the currently specified subset of the seL4 API could not be implemented on real ARMv6 hardware.

6.2 Parallel Development

Our approach of concurrently implementing a kernel model in Haskell, formalising it, and porting applications to the simulation environment has proved productive. It provides feedback during highly interactive and interwoven design iterations that have not yet concluded.

The translation to Isabelle/HOL started relatively early, when the seL4 API was nearing a first stable point and first user-level binaries could be run through the machine simulator. During the translation process, we found and fixed a number of problems, for example an unintentionally unbounded runtime of the IPC send operation. It was discovered because Isabelle demanded termination proofs for operations that were supposed to execute in constant time.

This shows that formalisation and the use of theorem proving tools is beneficial even if full verification is not yet performed. In our setting the formalisation cost so far has been significantly lower than the implementation and testing cost, while the design team did not have to switch to completely new methods or notations. The application of formalisation early in the design phase also avoids potentially costly corrections later.

The porting of existing software to the simulation environment has also led to the identification of issues requiring attention. When attempting to implement a higher-level system upon the microkernel, a required operation on a particular type of capability was found to be missing. The missing operation was added in hours, and formalised soon afterwards.

Summarising, we have found our methodology has enabled the kernel designers and implementors, the formal modellers, and the higher-level system programmers to work more closely together, leading to faster and better results than we would expect if the phases had been sequential.

6.3 Progress

It is difficult to quantify the productivity gain we believe we have by using our approach. We know of two data points with which we can roughly compare: the VFiasco project [15], and the Coyotos project [27]. Both projects aim to produce a formally verified microkernel via differing approaches.

The VFiasco project aims to verify the existing Fiasco microkernel directly by developing a formal semantics for a subset of C++ (its implementation language). The project began in Nov 2001 and has produced formal semantics for some of C++. It is not clear how much progress has been made on formalisation of the microkernel itself, nor how near they are to a subset of C++ sufficient to cover the subset used to implement Fiasco.

The Coyotos project takes the approach of developing a new low-level language (BitC) with precise formal semantics that can serve as the implementation language. They have released a specification and an alpha-release compiler for BitC, but are yet to publish a formal semantics for the language. The project has also published an informal reference manual for the Coyotos kernel itself. It is unclear how the actual implementation of the reference manual is progressing beyond what is publicly available in their source repository, which contains mostly kernel support libraries and utilities required to bootstrap a kernel on raw hardware.

In contrast, our approach has produced a precisely specified kernel API, together with a usable reference implementation. We also have a formal model in Isabelle for the implementation.

6.4 Precise Specification

Our choice of Literate Haskell as our modelling language has enabled us to produce a reference manual and implementation that is one and the same thing, ensuring that our reference manual and reference implementation are consistent. Our catch phrase is “we run the manual”. While our hope is to produce a readily understandable reference manual describing each operation with the reference Haskell implementation as the definitive definition of each operation, structuring our code to avoid too much implementation detail has proved challenging. However, the document is improving with each iteration.

7. Related Work

Operating systems in functional languages. Early examples of the use of functional languages for systems programming are the work on Nebula [17] and KAOS [28]. These early works used CPS and stream processing to model the state and event-based interfaces of the underlying hardware. Improving on some of the early approaches, Wallace & Runciman investigated the use of functional programming for embedded systems [33].

Recently, the House and Osker kernels [12] (in Haskell) and the Hello kernel [9] (in Standard ML) demonstrated that modern functional languages can be used to develop bare metal implementations of operating systems. A central aspect of this work is the adaptation of the runtime system (RTS) of a high-performance implementation, such as that of GHC, to run without any operating-system support on bare metal. Building on such a modified RTS, the work on House & Osker has contributed abstractions of the underlying hardware (such as memory management) that simplify the reasoning about low-level code. This work also used a monadic interface whose properties were formalised in P-Logic.

Verification of operating systems. Earlier work on OS verification includes PSOS [7] and UCLA Secure Unix [32]. Later, KIT [2] describes verification of process isolation properties down to object code level, but for an idealised kernel with far simpler and less general abstractions than modern microkernels. A number of case studies [4, 6, 31] describe the IPC and scheduling subsystems of microkernels in PROMELA and verify them with the SPIN model checker. Manually constructed, these abstractions are not necessarily sound, and so while useful for discovering concurrency bugs, they cannot provide guarantees of correctness. The VeriSoft project [10] is attempting to verify a whole system stack, including hardware, compiler, applications, and a simplified microkernel called VAMOS. We discussed VFiasco [15] and Coyotos [27] in the previous section.

Our approach occupies the middle ground between two extremes: the a priori approach where the kernel is designed formally from the start, and the a posteriori approach where a traditional (C/C++) implementation is created first and formalised later. Both can be found in the literature, e.g. the formal design process of PSOS [7] and implementation verifications such as [4, 6, 31].

In our setting, the a priori approach would design the kernel directly in the theorem prover and extract a program to be used for validation. This requires that the OS designers are intimately familiar with the formal specification language, which they are usually not. Haskell on the other hand is commonly taught to undergraduate students. They also would be restricted in their use of the language by the executable fragment of HOL, since validation of low-level design decisions is necessary to distinguish between those designs that can possibly be implemented efficiently and those that cannot. This restriction is significant, because, as opposed to Haskell, even full Isabelle/HOL, while perfectly suited for specification, is not a comfortable programming language yet, certainly not one for rapid development, testing and prototyping of sizeable, low-level, and largely imperative systems.

The a posteriori approach would create a traditional C implementation first. Folklore says and our own experience [30] shows that the effort for formalisation here is significantly higher and correspondence to the prototype much less obvious. Additionally, the effort for implementation is significantly higher as well — we estimate the effort for creating a micro-kernel prototype the traditional way in our OS group to be about 1 person year. This does not include the numerous iterative changes to the API that we went through in our process.

Our approach lies in between. Compared to the a priori method, we enjoy the richness and expressiveness of a full functional programming language and keep the intricacies of formalisation from the OS designers. Compared to the a posteriori method, we arrive at a precise formalisation very quickly and easily. We also significantly speed up development and make an iterative prototyping process possible that in a few months has gone through more API changes than what would otherwise have taken years to implement.

8. Conclusion

We have described and applied a method for high turnaround, high assurance development of microkernels. At the heart of the method is the use of the functional programming language Haskell, which is used to specify and implement an abstract model of the microkernel. The use of a high-level language to specify the kernel avoids the common pitfalls of high assurance tools being inaccessible to typical kernel developers, inadvertent ambiguity of informal specification, and the complexity of managing low-level hardware just to prototype ideas. The method produces a specification that is readily amenable to formalisation, a requirement for high assurance. When combined with a machine simulator, the specification also serves as a reference platform for the construction of higher-level systems upon the prototype kernel.

Our experience with the methodology has been that it enables the prototyping of ideas without requiring a semi-complete prototype to simply boot and test the kernel, and it provides both formal modellers and application developers with prototype implementations earlier, leading to faster design iterations. Formalisation has proved to be much easier using our methodology compared to extracting a formal model from a traditional reference manual together with a low-level language implementation. This is due to the nature of the specification language, and the fact it is a precise specification. We believe the methodology provides us with productivity gains compared to approaches taken by projects with similar goals.

We expect to continue stepping through iterations of our design while continuing porting our higher-level application environment to the prototype kernel. We expect the design to mature in the coming months, at which point we will embark on a bare-metal implementation using a traditional systems language, which the verification project expects to show is a refinement of our original kernel specification.

Acknowledgements We thank Simon Winwood, Gernot Heiser and the anonymous reviewers for feedback on earlier drafts of this paper. National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through *Backing Australia's Ability* and the ICT Research Centre of Excellence programs.

References

[1] A. Abel, M. Benke, A. Bove, J. Hughes, and U. Norell. Verifying Haskell programs using constructive type theory. In *Haskell'05, Tallinn, Estonia*, 2005.

[2] W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.

[3] M. Bishop and L. Snyder. The transfer of information and authority in a protection system. In *SOSP '79: Proceedings of the seventh ACM symposium on Operating systems principles*, pages 45–54, New York, NY, USA, 1979. ACM Press.

[4] T. Cattel. Modelization and verification of a multiprocessor realtime OS kernel. In *Proceedings of FORTE '94, Bern, Switzerland*, October 1994.

[5] N. A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In J. G. Morrisett and S. L. P. Jones, editors, *POPL*, pages 206–217. ACM, 2006.

[6] G. Duval and J. Julliand. Modelling and verification of the RUBIS μ-kernel with SPIN. In *SPIN95 Workshop Proceedings*, 1995.

[7] R. J. Feiertag and P. G. Neumann. The foundations of a provably secure operating system (PSOS). In *AFIPS Conference Proceedings (NCC 79)*, pages 329–334, New York, NY, USA, June 1979.

[8] B. Ford, M. Hibler, J. Lepreau, R. McGrath, and P. Tullmann. Interface and execution models in the Fluke kernel. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 101–115, New Orleans, LA, USA, Feb. 1999. USENIX.

[9] G. Fu. Design and implementation of an operating system in Standard ML. Master's thesis, Dept. of Information and Computer Sciences, University of Hawaii at Manoa, 1999. Available: `http://www2.ics.hawaii.edu/~esb/prof/proj/hello/index.html`.

[10] M. Gargano, M. Hillebrand, D. Leinenbach, and W. Paul. On the correctness of operating system kernels. In *Proc. 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs'05)*, pages 1–16, Oxford, UK, 2005.

[11] T. Hallgren, J. Hook, M. P. Jones, and R. B. Kieburtz. An overview of the Programatica ToolSet. High Confidence Software and Systems Conference, HCSS04, 2004.

[12] T. Hallgren, M. P. Jones, R. Leslie, and A. Tolmach. A principled approach to operating system construction in Haskell. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 116–128, New York, NY, USA, 2005. ACM Press.

[13] W. L. Harrison and R. B. Kieburtz. The logic of demand in Haskell. *Journal of Functional Programming*, 15(6):837–891, 2005.

[14] Haskell hierarchical libraries. `http://www.haskell.org/ghc/docs/latest/html` 2006.

[15] M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *Proc. 2nd ECOOP Workshop on Programm Languages and Operating Systems*, Glasgow, UK, Oct. 2005.

[16] B. Huffman, J. Matthews, and P. White. Axiomatic constructor classes in Isabelle/HOLCF. In J. Hurd and T. F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 147–162. Springer Verlag, 2005.

[17] K. Karlsson. Nebula: a functional operating system. Technical Report LPM11, Laboratory for Programming Methodology, Chalmers University of Technology and University of Goteburg, 1981.

[18] G. Klein and H. Tuch. Towards verified virtual memory in L4. In K. Slind, editor, *TPHOLs Emerging Trends '04*, Park City, Utah, USA, 2004.

[19] R. Kolanski and G. Klein. Formalising the L4 microkernel API. In B. Jay and J. Gudmundsson, editors, *Computing: The Australasian Theory Symposium (CATS 06)*, volume 51 of *Conferences in Research and Practice in Information Technology*, pages 53–68, Hobart, Australia, Jan. 2006.

[20] L4Ka Team. L4Ka::Pistachio kernel. `http://l4ka.org/projects/pistachio/`.

[21] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *SOSP '75: Proc. Fifth Symposium on Operating Systems Principles*, pages 132–140, New York, NY, USA, 1975. ACM Press.

[22] J. Liedtke. Address space sparsity and fine granularity. *SIGOPS Oper. Syst. Rev.*, 29(1):87–90, 1995.

[23] J. Liedtke. Towards real microkernels. *Communications of the ACM*, 39(9):70–77, Sept. 1996.

[24] R. J. Lipton and L. Snyder. A linear time algorithm for deciding subject security. *J. ACM*, 24(3):455–464, 1977.

[25] The M5 simulator system. `http://m5.eecs.umich.edu/`, 2006.

[26] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer Verlag, 2002.

[27] J. Shapiro. Coyotos. `www.coyotos.org`, 2006.

[28] W. Stoye. Message-based functional operating systems. *Science of Computer Programming*, 6(3):291–311, 1986.

[29] H. Tuch and G. Klein. A unified memory model for pointers. In *Proceedings of the 12th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, pages 474–488, Montego Bay, Jamaica, Dec. 2005.

[30] H. Tuch, G. Klein, and G. Heiser. OS verification — now! In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, Santa Fe, NM, USA, June 2005.

[31] P. Tullmann, J. Turner, J. McCorquodale, J. Lepreau, A. Chitturi, and G. Back. Formal methods: a practical tool for OS implementors. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems*, pages 20–25, 1997.

[32] B. Walker, R. Kemmerer, and G. Popek. Specification and verification of the UCLA Unix security kernel. *CACM*, 23(2):118–131, 1980.

[33] M. Wallace and C. Runciman. Lambdas in the liftshaft—functional programming and an embedded architecture. In *FPCA '95: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, pages 249–258, New York, NY, USA, 1995. ACM Press.