Secure Microkernels, State Monads and Scalable Refinement

David Cock[1], Gerwin Klein[1,2], and Thomas Sewell[1]

[1] Sydney Research Lab., NICTA*, Australia
[2] School of Computer Science and Engineering, UNSW, Sydney, Australia
{david.cock|gerwin.klein|thomas.sewell}@nicta.com.au

Abstract. We present a scalable, practical Hoare Logic and refinement calculus for the nondeterministic state monad with exceptions and failure in Isabelle/HOL. The emphasis of this formalisation is on large-scale verification of imperative-style functional programs, rather than expressing monad calculi in full generality. We achieve scalability in two dimensions. The method scales to multiple team members working productively and largely independently on a single proof and also to large programs with large and complex properties.
We report on our experience in applying the techniques in an extensive (100K lines of proof) case study—the formal verification of an executable model of the seL4 operating system microkernel.

1 Introduction

This paper touches on three main topics: the verification of a secure operating system microkernel, the state monad as used in Haskell progams, and formal refinement as the verification technique in the correctness proof.

The main motivation for our work is the first of these three. In the larger context, we are aiming to design and fully formally verify the seL4 microkernel down to the level of its ARM11 C implementation. The seL4 microkernel [4,6] is an evolution of the L4 familiy [15] for secure, embedded devices. As described elsewhere [5], the design of seL4 involved building a binary compatible prototype of the kernel in the programming language Haskell which subsequently was automatically translated into Isabelle/HOL to arrive at a very detailed, executable formal model of the kernel. This operational model is inherently state based, and the corresponding Haskell program makes extensive use of the state monad to express the corresponding state transformations. The model is low level in the sense that it uses data types such as 32 bit wide finite machine words, models the heap memory of the eventual C program explicitly as part of its state, and mutates typical pointer data structures such as doubly linked lists on that heap.

Complementing this executable model is a still operational, but more abstract specification of the functional behaviour of seL4. This more abstract model

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council.

uses nondeterminism to leave details unspecified and uses, for instance, abstract functions instead of explict pointer representations (although it still makes use of *references* on many occasions, e.g. to model the user-visible sharing behaviour of particular data structures).

This paper presents the main techniques we used in verifying that the executable model correctly implements its abstract specification. It should be noted explicitly that we did not aim for maximum generality and theoretical depth in either the formalisations or the techniques. Instead, we focused on simplicity, easy applicability, and most importantly scalability of the methods. As a microkernel, seL4 is neither nicely modular, nor does it implement a nicely self contained abstract algorithm. Compared to other verifications, the main challenge was to deal with a highly complex, intermingled set of low-level data structures with high reliance on global invariants exploited in various optimisations. The size of the specifications, with about 3K lines of Isabelle definitions on the abstract and 7K lines on the concrete side, implies a massive proof effort which we aimed to spread over multiple people working concurrently, with as little need for interaction and coordination as possible.

In summary, this paper can be seen as a study on how far you can get with the simplest possible methods. It is our hypothesis that it was precisely this simplicity that enabled us to achieve this large-scale verification.

The contributions of this paper are as follows.

– We formalise the nondeterministic state monad with exceptions and failure in Isabelle/HOL. This subsumes the state monad with exceptions that is commonly used in Haskell.
– We present a Hoare Logic and refinement calculus on the above, both simple yet scalable and practical.
– We report on our experience in applying the above to a binary compatible, executable model of seL4 microkernel translated from Haskell.

The following sections provide detail on each of these in turn.

2 State Monads

A state monad allows a pure functional model of computation with side effects. For result type *'a* and state type *'s*, the associated monad type (abbreviated (*'s*, *'a*) *state-monad*) is *'s* ⇒ *'a* × *'s*. That is, a function from previous state to next state together with a computation result. A pure state transformer is typically denoted by the one-valued return type *unit* ie. *'s* ⇒ *unit* × *'s*.

All monads define two constructors, here called return and bind. For the state monad they are defined as follows:

$$\begin{array}{l}\mathsf{return} \;::\; 'a \Rightarrow ('s,'a)\ \mathit{state\text{-}monad}\\ \mathsf{return}\ a \equiv \lambda s.\ (a,\ s)\end{array}$$

$$\begin{array}{l}\mathsf{bind} \;::\; ('s,'a)\ \mathit{state\text{-}monad} \Rightarrow ('a \Rightarrow ('s,'b)\ \mathit{state\text{-}monad}) \Rightarrow ('s,'b)\ \mathit{state\text{-}monad}\\ \mathsf{bind}\ f\ g \equiv \lambda s.\ \mathsf{let}\ (v,\ s') = f\ s\ \mathit{in}\ g\ v\ s'\end{array}$$

Note that as Isabelle/HOL is simply typed, it is not possible to straightforwardly define the monad type constructor as in Haskell, defining return, bind and associated syntax once, and thereby proving results generically about the class of monadic types. The solution that we adopt is to instantiate for specific monad constructors, e.g., return :: *'a* $\Rightarrow$ *'a state-monad*.

The constructor return simply injects the value a into the monad type, passing the state unchanged, whilst bind sequentially composes a computation f, and a computation g (a function from the return type of f). The expression BIND f g is abbreviated as a >>= b. To allow concise description of longer computations, we define a do syntax in a similar fashion to Haskell:

$$f \mathbin{>\!>\!=} g \equiv \mathsf{do}\ x \leftarrow f;\ g\ x\ \mathsf{od}$$

A state monad also defines two additional constructors: get and put, the primitive state transformers (here () is the sole element of type *unit*):

$$\begin{array}{ll} \mathsf{get} :: (\text{'}s,\ \text{'}s)\ \textit{state-monad} & \mathsf{put}\ :: \text{'}s \Rightarrow (\textit{unit},\ \text{'}s)\ \textit{state-monad} \\ \mathsf{get} \equiv \lambda s.\ (s,\ s) & \mathsf{put}\ s \equiv \lambda\text{-}.\ ((),\ s) \end{array}$$

The constructors of all monads must obey the following three laws, which we have instantiated and proved for each monad instance:

$$\begin{array}{ll} \mathsf{return}\ x \mathbin{>\!>\!=} f = f\,x & \text{RETURN\_BIND} \\ m \mathbin{>\!>\!=} \mathsf{return} = m & \text{BIND\_RETURN} \\ (m \mathbin{>\!>\!=} f) \mathbin{>\!>\!=} g = m \mathbin{>\!>\!=} (\lambda x.\ f\,x \mathbin{>\!>\!=} g) & \text{BIND\_ASSOC} \end{array}$$

The simple state monad is able to model sequential computations with side-effects, but does not provide good notation for non-local flow control (e.g. exceptions). A straightforward way to model try-catch-style exceptions is to instantiate the state monad using the sum type *'e* + *'a* (for result type *'a* and exception type *'e*) in place of the simple result type. Every component in the monad now returns either Inr a in case of success, or Inl e in case of failure with exception e. To complete the model we require a new bind constructor, bindE which propagates exceptions, and the catch constructor to embed the error monad into the non-error state monad.

$$\begin{array}{ll} \mathsf{lift}\ f\ g & \equiv \mathsf{case}\ v\ \mathsf{of\ Inl}\ e \Rightarrow \textit{throwError}\ e \mid \mathsf{Inr}\ v' \Rightarrow f\ v' \\ \mathsf{bindE}\ f\ g & \equiv f \mathbin{>\!>\!=} \mathsf{lift}\ g \\ \mathsf{catch}\ f\ \textit{handler} & \equiv \mathsf{do}\ x \leftarrow f; \\ & \qquad \mathsf{case}\ x\ \mathsf{of\ Inl}\ e \Rightarrow \textit{handler}\ e \mid \mathsf{Inr}\ b \Rightarrow \mathsf{return}\ b \\ & \quad \mathsf{od} \end{array}$$

In formulating an abstract behavioural model, it is convenient to express computation nondeterministically. This is readily modelled as an extension of the state monad by allowing each computation to return a (possibly empty) set of value-state pairs: *'s* $\Rightarrow$ (*'a* $\times$ *'s*) set, and redefining bind as $\lambda s.\ \bigcup\{g\ a\ s' \mid (a,s') \in f\ s\}$. This formulation has a drawback however: The obvious way to model failure, fail $\equiv \lambda s.\ \{\}$, admits the existential statement: "For all states s, not all paths fail". What we desire, however, is the universal statement "For all states s, no path fails", which cannot be expressed as a simple predicate on the state set, as the failure case ({}) is dominated in the union by the non-failure case. One solution is to append a failure flag which is propagated separately, and which dominates non-failure in bind. This leads us to the following definitions:

$$\begin{aligned}&\mathsf{return}\ a \equiv \lambda s.\ (\{(a,\ s)\},\ \mathsf{False})\\ &\mathsf{bind}\ f\ g \equiv \lambda s.\ (\bigcup \mathsf{fst}\ `\ (\lambda(x,\ y).\ g\ x\ y)\ `\ \mathsf{fst}\ (f\ s),\\ &\qquad\qquad\quad \mathsf{True} \in \mathsf{snd}\ `\ (\lambda(x,\ y).\ g\ x\ y)\ `\ \mathsf{fst}\ (f\ s) \lor \mathsf{snd}\ (f\ s))\end{aligned}$$

In addition to the state monad constructors get and put, we define select and $\mathsf{alternative}$ to perform nondeterministic actions. Constructor select is a nondetermistic return, and takes a set of values, while $\mathsf{alternative}$ executes one of the two computations passed as arguments.

$$\begin{array}{llll}\mathsf{get} & :: ('s,'s)\ \mathit{nd\text{-}monad} & \mathsf{put} & :: 's \Rightarrow (\mathit{unit},'s)\ \mathit{nd\text{-}monad}\\ \mathsf{get} & \equiv \lambda s.\ (\{(s,\ s)\},\ \mathsf{False}) & \mathsf{put}\ s & \equiv \lambda\text{-}.\ (\{((),\ s)\},\ \mathsf{False})\\ \mathsf{select} & :: 'a\ \mathsf{set} \Rightarrow ('s,'a)\ \mathit{nd\text{-}monad} & \mathsf{fail} & :: ('s,'a)\ \mathit{nd\text{-}monad}\\ \mathsf{select}\ A & \equiv \lambda s.\ (A \times \{s\},\ \mathsf{False}) & \mathsf{fail} & \equiv \lambda s.\ (\{\},\ \mathsf{True})\end{array}$$

The nondeterminism inherent in the model allows us to model input conveniently: $\mathsf{do}\ x \leftarrow \mathsf{select}\ \mathit{InputAction};\ f\ x\ \mathsf{od}$. We use fail to indicate catastrohpic failure in the kernel (e.g. a kernel panic). It is part of the proof to show that these are never triggered.

3 Hoare Logic on State Monads

The Hoare triple $\{\!|P|\!\}\ f\ \{\!|Q|\!\}$ is a predicate on the computation f, stating that if the precondition P holds before executing f, then the precondition Q will hold afterwards. For the nondeterministic state monad, the basic hoare triple also needs to take into account the return value and is defined as follows:

$$\{\!|P|\!\}\ f\ \{\!|Q|\!\} \equiv \forall s.\ P\ s \longrightarrow (\forall (r,\ s')\in\mathsf{fst}\ (f\ s).\ Q\ r\ s')$$

Note that the postcondition Q is a binary predicate while P is unary. For the state monad with exceptions, we define:

$$\{\!|P|\!\}\ f\ \{\!|Q|\!\},\ \{\!|R|\!\} \equiv \{\!|P|\!\}\ f\ \{\!|\lambda r\ s.\ \mathsf{case}\ r\ \mathsf{of}\ \mathsf{Inl}\ a \Rightarrow R\ a\ s\ |\ \mathsf{Inr}\ b \Rightarrow Q\ b\ s|\!\}$$

This specifies a seperate postcondition for the exception and non-exception cases. All of the following rules have a natural expression for the state-exception monad in terms of this augmented Hoare triple.

To build a calculus for reasoning about monadic computations, we first state and prove axiomatic rules for the basic constructors:

$$\begin{array}{ll}\{\!|\lambda s.\ P\ ()\ x|\!\}\ \mathsf{put}\ x\ \{\!|P|\!\}\ \textsc{put-wp} & \{\!|\lambda s.\ P\ s\ s|\!\}\ \mathsf{get}\ \{\!|P|\!\}\ \textsc{get-wp}\\ \{\!|P\ x|\!\}\ \mathsf{return}\ x\ \{\!|P|\!\}\ \textsc{return-wp} & \end{array}$$

Constructor bind requires a more complicated family of rules, to capture the interaction of the pre- and post-conditions of composed computations:

$$\frac{\forall x.\ \{\!|B\ x|\!\}\ g\ x\ \{\!|C|\!\} \qquad \{\!|A|\!\}\ f\ \{\!|B|\!\}}{\{\!|A|\!\}\ f\ >\!\!>\!= g\ \{\!|C|\!\}}\ \textsc{seq}$$

Note that the SEQ rule does not lose information in the universal quantifier. The intermediate predicat B can always be made strong enough to describe result values of f and preconditions on the parameter of g precisely.

Finally, to complete the basic calculus we introduce a weakening rule, to substitute arbitrary preconditions. The analogous rule holds or postconditions.

$$\frac{\{\!|Q|\!\}\ f\ \{\!|R|\!\} \qquad \forall s.\ P\ s \longrightarrow Q\ s}{\{\!|P|\!\}\ f\ \{\!|R|\!\}}\ \textsc{weaken}$$

Similar rules can be created for the absence of failure in the computation.

4 Verification Condition Generator

As usual in Hoare Logic, reasoning within this calculus can be substantially automated by the use of a verification condition generator (VCG) if we phrase our structural Hoare rules in weakest-precondition (WP) form. The rules given for put, get and return in Sect. 3 are weakest-precondition rules. As an example, consider the following definition of the modify constructor, and the proof of its associated weakest precondition rule:

$$\mathsf{modify}\ f \equiv \mathsf{do}\ s \leftarrow \mathsf{get};\ \mathsf{put}\ (f\ s)\ \mathsf{od}$$

We wish to show $\{\!|\lambda s.\ P\ ()\ (f\ s)|\!\}\ \mathsf{modify}\ f\ \{\!|P|\!\}$. Before invoking the VCG, we unfold definitions until the goal is phrased in terms of known operations. The VCG then produces the following proof steps automatically.

It starts by applying the WEAKEN rule to replace the concrete precondition with a schematic[3] precondition, $?Q$, and an implication. We get two new goals:

1. $\{\!|\lambda s.\ ?Q|\!\}\ \mathsf{do}\ s \leftarrow \mathsf{get};\ \mathsf{put}\ (f\ s)\ \{\!|P|\!\}$
2. $\forall s.\ P\ ()\ (f\ s) \longrightarrow\ ?Q$

The VCG now repeatedly tries to apply one from its set of WP rules. The rule for the bind operator will match the current goal, because its postcondition is fully general (matching the concrete P) and the precondition, that might be concrete in the rule, matches the schematic precondition that we have just created in the goal. If the WP set is constructed correctly, the goal will alway remain in this form and for every operator there will be one rule that matches. In our example, the VCG would apply SEQ, PUT-WP, and GET-WP in turn, which leaves the user with only the implication introduced at the first step. This is a HOL formula, free of both monad and Hoare syntax:

1. $\forall s.\ P\ ()\ (f\ s) \longrightarrow P\ ()\ (f\ s)$

Here, the goal is trivial, because the precondition we set out to prove was the weakest precondition. We could now add this new WP rule for *modifiy* to the set available to the VCG, to avoid having to unfold the definition of modify in the future. In this manner we progressively build the calculus towards a higher and higher level of abstraction. So far, our use of the VCG is fairly standard.

[3] Schematic variables in Isabelle stand for terms that can be instantiated (as opposed to free variables that need to remain fixed in proofs).

Note that, if we add rules that are not strictly weakest-precondition, we do not affect the soundness of the VCG, we simply take the risk that the implication goal produced may be false, indicating that our rules need to be strengthened.

The weakest-precondition rules mentioned so far all apply to an arbitrary postcondition. For elementary functions like put, get and modify, rules of this form are easily stated. In principle such a rule can be stated for any of the monadic functions we use. In practice, however, the preconditions in these rules will be of exponential term size with respect to the complexity of the operator. The tractable solution we have found is to supply the VCG instead with Hoare triples that have specific postconditions and manually simplified preconditions. In principle these can still be weakest precondition rules, however this is not normally the case. An example is SET-EP-VALID-OBJS:

$$\{\!|\lambda s.\ \mathsf{valid\text{-}objs}\ s \wedge \mathsf{valid\text{-}ep}\ v\ s|\!\}\ \mathsf{set\text{-}endpoint}\ ep\ v\ \{\!|\lambda rv\ s.\ \mathsf{valid\text{-}objs}\ s|\!\}$$

The $\mathsf{set\text{-}endpoint}$ function replaces the state of an endpoint found at a given pointer location. The $\mathsf{valid\text{-}objs}$ predicate in the postcondition is one of our global invariants, and establishes that all objects satisfy certain validity criteria. Clearly for this to be maintained we need the additional information that the endpoint value being inserted satisfies the validity criteria for endpoints, $\mathsf{valid\text{-}ep}$. This is not the weakest possible precondition, as it globally asserts in $\mathsf{valid\text{-}objs}$ that the endpoint about to be replaced is valid, which is unnecessary. The precise weakest precondition would be tedious to define, and the too-strong precondition will always be true in practice.

Hoare triples with specific postconditions complicate the VCG as additional efforts must be made to connect the postconditions available to the one that is needed. To illustrate this problem, consider the scenario in which we wish to establish $\mathsf{valid\text{-}objs}$ after a pair of endpoint updates.

$$\begin{array}{l}\{\!|\lambda s.\ \mathsf{valid\text{-}objs}\ s \wedge \mathsf{valid\text{-}ep}\ v\ s \wedge \mathsf{valid\text{-}ep}\ v'\ s|\!\}\\ \quad \mathsf{do}\ \mathsf{set\text{-}endpoint}\ p\ v;\ \mathsf{set\text{-}endpoint}\ p'\ v'\ \mathsf{od}\\ \{\!|\lambda rv.\ \mathsf{valid\text{-}objs}|\!\}\end{array}$$

The VCG can divide the problem using SEQ and apply SET-EP-VALID-OBJS to the Hoare triple for the second update. The postcondition for the first update will then be $\lambda rv\ s.\ \mathsf{valid\text{-}objs}\ s \wedge \mathsf{valid\text{-}ep}\ v'\ s$. To apply SET-EP-VALID-OBJS again, the VCG must use the conjunction lifting rule.

$$\frac{\{\!|P|\!\}\ f\ \{\!|Q|\!\} \qquad \{\!|P'|\!\}\ f\ \{\!|Q'|\!\}}{\{\!|\lambda s.\ P\ s \wedge P'\ s|\!\}\ f\ \{\!|\lambda rv\ s.\ Q\ rv\ s \wedge Q'\ rv\ s|\!\}}\ \text{CONJ-LIFT}$$

The conjunction operator is one of a family of first order logic operators that have a VCG lifting rule. Conjunction, disjunction, universal and existential operators have lifting rules, but the negation operator does not. Implication is dealt with by reducing to a disjunction and negation, after which the negation must be dealt with explicitly.

By default the VCG will use only the CONJ-LIFT lifting rule and will use it only conservatively, that is, only when one of its assumptions can immediately be resolved using an available rule. If asked, the VCG can use any of these rules aggressively, that is, whenever possible. Conjunction occurs in our postconditions regularly and other operators rarely, thus this facility has been found sufficient.

The motivation for this restricted behaviour is pragmatism with respect to the interactive proof process. Our hypothesis is that it is more helpful for the VCG to be conservative, manipulating postconditions only when it knows it is making progress and stopping otherwise, than to be aggressive and occasionally return to the user an unexpected or counterintuitive interactive state. The above behaviour loosely conforms to this policy.

The VCG is not limited to Hoare triples. Rules for absence of failure as mentioned in Sect. 3 can be similarly automated within the same tool.

5 Refinement Calculus

The ultimate objective of our proof effort is to prove the refinement property [3] between abstract and concrete processes. For our purposes, a process is defined by an initialisation function which sets up the total state with reference to some external state, a step function which reacts to an event, and a finalisation function which retreives the external state.

$$\begin{aligned}\textbf{record}\ \textit{process} = {} & \mathsf{Init} :: \textit{'external} \Rightarrow \textit{'state}\ \mathsf{set} \\ & \mathsf{Step} :: \textit{'event} \Rightarrow (\textit{'state} \times \textit{'state})\ \mathsf{set} \\ & \mathsf{Fin} :: \textit{'state} \Rightarrow \textit{'external}\end{aligned}$$

The execution of a process from a starting external state through a series of input events results in a set of external states at that point.

$$\begin{aligned}&\mathsf{steps}\ \delta \equiv \mathsf{foldl}\ (\lambda S\ j.\ \delta\ j\ ``\ S) \\ &\mathsf{execution}\ A\ s\ js \equiv \mathsf{Fin}\ A\ `\ \mathsf{steps}\ (\mathsf{Step}\ A)\ (\mathsf{Init}\ A\ s)\ js\end{aligned}$$

One process is refined by another if its execution over the same input events always results in a set of states that is a subset of those produced by the other.

$$A \sqsubseteq C \equiv \forall js\ s.\ \mathsf{execution}\ C\ s\ js \subseteq \mathsf{execution}\ A\ s\ js$$

Refinement is commonly proven by establishing forward simulation, a property which implies it. To prove forward simulation we introduce a relation state-relation which connects the states of the two processes. We must show that the relation is established by Init, that it is maintained if we advance the systems in parallel, and that it is sufficient to equate the external state. The ;; operator in this definition is relation composition.

$$\begin{aligned}\mathsf{fw\text{-}sim}\ \mathsf{state\text{-}relation}\ C\ A \equiv {} & (\forall s.\ \mathsf{Init}\ C\ s \subseteq \mathsf{state\text{-}relation}\ ``\ \mathsf{Init}\ A\ s) \\ & \wedge (\forall j.\ \mathsf{state\text{-}relation}\ ;;\ \mathsf{Step}\ C\ j \subseteq \mathsf{Step}\ A\ j\ ;;\ \mathsf{state\text{-}relation}) \\ & \wedge (\forall s\ s'.\ (s,\ s') \in \mathsf{state\text{-}relation} \longrightarrow \mathsf{Fin}\ C\ s' = \mathsf{Fin}\ A\ s)\end{aligned}$$

To address our scalability concerns, we wish to decompose the refinement problem into smaller subproblems and transfer the refinement statement to the state monad. The simplest way to do this is to scale the forward simulation down to component functions. The corres predicate captures forward simulation over a single concrete monadic function and its abstract counterpart. It takes three additional parameters. The relation R must apply to any possible return values.

The preconditions P and P' restrict the input states, allowing use of information such as global invariants.

$$\begin{array}{l}\textsf{corres}\ R\ P\ P'\ m\ m' \equiv \forall (s,\ s')\in\textsf{state-relation}.\ P\ s \wedge P'\ s' \longrightarrow \\ \quad (\forall (r',\ t')\in\textsf{fst}\ (m'\ s').\ \exists (r,\ t)\in\textsf{fst}\ (m\ s).\ (t,\ t') \in \textsf{state-relation} \wedge R\ r\ r') \\ \quad \wedge\ (\textsf{snd}\ (m'\ s') \longrightarrow \textsf{snd}\ (m\ s))\end{array}$$

Note that the outcome of the monadic computation is a pair of result and failure flag. The last part of the corres statement above is stronger than strictly needed for refinement. It states that failure on the concrete m' implies failure on the abstract m. This means we only have to show absence of failure on the most abstract level and get absence of failure on all concrete levels by refinement.

The key property of corres is that it decomposes over the bind constructor through the CORRES-SPLIT rule.

CORRES-SPLIT:

$$\frac{\textsf{corres}\ r'\ P\ P'\ a\ c \qquad \forall rv\ rv'.\ r'\ rv\ rv' \longrightarrow \textsf{corres}\ r\ (R\ rv)\ (R'\ rv')\ (b\ rv)\ (d\ rv') \qquad \{\!|Q|\!\}\ a\ \{\!|R|\!\} \qquad \{\!|Q'|\!\}\ c\ \{\!|R'|\!\}}{\textsf{corres}\ r\ (P\ \textsf{and}\ Q)\ (P'\ \textsf{and}\ Q')\ (a >\!>= b)\ (c >\!>= d)}$$

Similar splitting rules exist for other common monadic constructs such as bindE, catch and conditional expressions. There are standard results for the elementary monadic functions. An example is:

CORRES-RETURN:

$$\frac{r\ x\ y}{\textsf{corres}\ r\ (\lambda s.\ \textsf{True})\ (\lambda s.\ \textsf{True})\ (\textsf{return}\ x)\ (\textsf{return}\ y)}$$

The corres predicate also has a weakening rule, similar to the Hoare Logic.

CORRES-PRECOND-WEAKEN:

$$\frac{\textsf{corres}\ r\ Q\ Q'\ f\ g \qquad \forall s.\ P\ s \longrightarrow Q\ s \qquad \forall s.\ P'\ s \longrightarrow Q'\ s}{\textsf{corres}\ r\ P\ P'\ f\ g}$$

Proofs of the corres property take a common form. Firstly the definitions of the terms under analysis are unfolded and the CORRES-PRECOND-WEAKEN rule is used. Like in the VCG, this allows the syntactic construction of a precondition which suits the proof. The various splitting rules are used to decompose the problem, in some cases with carefully chosen return value relations. Preexisting results are used to solve the component corres problems. Some of these preexisting results, such as CORRES-RETURN, require compatibility properties on their parameters. These are typically established using information from previous return value relations. The VCG eliminates the Hoare triples, bringing preconditions assumed in corres properties at later points back to preconditions on the starting states. Finally, the precondition that was actually used must be proved a consequence of the one that was originally assumed.

6 Case Study – The seL4 Microkernel

In this section, we give an overview of the seL4 microkernel, its two formalisations in Isabelle/HOL, some of the properties we have proved on them, and our experience in this verification. With about 10,000 lines of C code, 7,500 lines of executable model and 3,000 lines of abstract Isabelle/HOL specification, the kernel is too large for us to provide any kind of useful detail in a conference paper, or even just a comprehensive overview of its formalisation. We do not attempt to do so; instead we provide a very high level view of its functionality, and show bits and pieces of the formalisation to give an impression of the general flavour.

6.1 Overview

As mentioned in the introduction, seL4 is an evolution of the L4 microkernel family. The main difference to L4 is that it is entirely capability based, unifying all resource accounting and access control into a single mechanism.

All kernel abstractions and system calls are provided via named, first-class kernel objects. Authorised users can obtain kernel services by invoking operations on kernel objects. Authority over these objects is conferred via capabilities only. System call arguments can either be data or other capabilities. Similar to L4, seL4 provides three basic abstractions: threads, address spaces and inter-process communication (IPC). In addition, seL4 introduces an abstraction, *untyped memory* (UM), which represents a region of currently unused physical memory.

An important part of the seL4 design is that all memory—be it the memory directly used by an application (e.g. memory frames) or indirectly in the kernel (e.g. page tables), is fully accounted for by capabilities. A parent capability to untyped memory can be refined into child capabilities to smaller sized untyped memory blocks or into other kernel objects via the *retype* operation on UM objects. The creator can then delegate all or part of the authority it possesses over the object to one or more of its clients. Untyped capabilities can be *revoked*. This removes all corresponding child capabilities from clients and prepares the memory spanned by that capability for retyping.

These mechanisms make seL4 a highly flexible microkernel supporting a number of practical application scenarios. A simple one is running a full legacy guest OS (e.g. Linux) next to a critical, trusted communications stack. Another one is to provide full separation between components at multiple different security levels with strict controls on explicit information channels between them.

6.2 Formalisation

We now give a very brief introduction to the formalisation of seL4. We begin with the state space of the abstract model.

This state is embedded into a process modelling machine execution of which we only make the kernel execution precise. User mode is free to mutate any user-accessible part of the state. The transitions for kernel execution are defined by nondeterministic monadic functions as presented in the previous sections. The triggers for these transitions are timer interrupts, kernel trap instructions (user level kernel calls), page faults, and user-level faults. We collect all of these in the data structure *event* that is shared with the executable level:

datatype *syscall* = *Send* | *Wait* | *SendWait* | *Identify* | *Yield*

datatype *event* = *SyscallEvent syscall* | *UnknownSyscall nat* |
UserLevelFault nat | *TimerInterrupt* | *VMFaultEvent vptr bool*

The type *syscall* models user level calls (sending/replying to IPC, identifying capabilities, yielding the current time slice). The other events are machine generated. Arguments to system calls are read from machine registers in binary form and decoded for further processing. This decoding phase is fully precise in the abstract specification, and therefore very similar on the executable and the abstract level. It is a major part of the programmer visible API specification. In fact, typical kernel reference manuals describe almost exclusively this syntactic part, and only sketch the semantics of the system. The latter is the bulk of the specification in our case.

The abstract state space of seL4 is a record with the following components.

record *state* = *pspace* :: *obj-ref* ⇀ *kernel-object*
cdt :: *cte-ptr* ⇀ *cte-ptr*
cdt-revokable :: *cte-ptr* ⇒ *bool*
cur-thread :: *obj-ref*
machine-state :: *machine-state*

datatype *kernel-object* = *CapTable cap-ref* ⇀ *cap* | *TCB tcb* |
Endpoint endpoint | *AsyncEndpoint async-ep* | *Frame*

The whole state space declaration is about 200 lines of Isabelle definitions, we mention only the salient points. The *pspace* component models the kernel-accessible part of memory. In this abstract view, it is a partial function from object references (machine words) to kernel objects. Separately from this, we model the capability derivation tree (CDT) in two components. The mapping database is the data structure that keeps track of the parent/child relationship between capabilities. It is realised as a partial function from child *capability table entry* (CTE) locations to parent CTE locations, i.e., a tree of CTE locations. In this model[4], there are two types of of kernel objects that can store capabilities: cap tables and thread control blocks (TCBs). A CTE location is fully determined by the location of the kernel object (an *obj-ref*) and a position within that kernel object (a *cap-ref*). As mentioned, cap tables store capabilities. TCBs back the kernel accounting for threads, synchronous and asyncronous endpoints are the kernel objects that back IPC and *Frame* objects stand for user data frames. The remaining two components of the global state are a pointer to the TCB of the current thread and the machine context (register state, for instance). We currently do not model the machine context in detail, but instead have a set of axiomatised functions like loadRegister/storeRegister on type *machine-state*.

Since the machine context is the main part of the shared outside-observable part of the two models, we have proved during refinement that the observable effect of reads, writes, cache flushes, TLB flushes, etc. is the same on both

[4] We present a slightly simpler, earlier version of the model here. The current version also contains interrupt tables and page table data structures.

levels. In the next step of refinement, to C, we plan to eliminate these remaining straightforward axioms and provide a direct model for the machine context.

In the concrete, executable model our abstract view of the CDT, as well as the abstract state of cap tables and other kernel objects vanish. Instead, they are encoded in much more detail. The CDT, for instance, becomes a doubly linked list together with a number of flags for level information, stored in machine words within CTEs. On the other hand, we gain a number of additional state components backing data structures that were not necessary on the abstract level. These are: a table of scheduling ready queues (indexed by a priority byte), and a scheduler action which effectively points to the next thread's TCB.

$$
\begin{aligned}
\textbf{record}\ \textit{kernel-state} = {} & \textit{ksPSpace} :: \textit{pspace} \\
& \textit{ksReadyQueues} :: \textit{8 word} \Rightarrow \textit{ready-queue} \\
& \textit{ksCurThread} :: \textit{32 word} \\
& \textit{ksSchedulerAction} :: \textit{scheduler-action} \\
& \textit{ksMachineState} :: \textit{machine-state}
\end{aligned}
$$

On the C level, *ksPSpace* corresponds to the heap and *ksMachineState* to the machine context as it does on the abstract side. The rest are global pointer variables. This means, the executable model is close to the final implementation.

For refinement, we need to define the process datatypes of the models. The executable model has a single entry point callKernel which handles the *event* type defined above. It is fairly natural then to define the Step component of the process datatype as the outcome of this nondeterministic monadic operator. Likewise, the Init component is based on resetting the state to the default newKernelState and then running the initKernel function. The Fin component is simply the *ksMachineState* projection. The abstract process is defined similarly.

The refinement property can then be proven from corres properties and Hoare triples. Firstly, we establish that our global invariant collections *invs* and *invs'* are invariants of the respective processes.

$$
\begin{aligned}
&\{\!|\lambda s.\ s = \textsf{new-kernel-state}|\!\}\ \textsf{init-kernel}\ \textit{entry frames offset kFrames}\ \{\!|\lambda rv.\ \textit{invs}|\!\} \\
&\{\!|\textit{invs}|\!\}\ \textsf{call-kernel}\ e\ \{\!|\lambda rv.\ \textit{invs}|\!\},\ \{\!|\lambda rv.\ \textit{invs}|\!\}
\end{aligned}
$$

$$
\begin{aligned}
&\{\!|\lambda s.\ s = \textsf{newKernelState}|\!\}\ \textsf{initKernel}\ \textit{entry frames offset kFrames}\ \{\!|\lambda rv.\ \textit{invs'}|\!\} \\
&\{\!|\textit{invs'}|\!\}\ \textsf{callKernel}\ e\ \{\!|\lambda rv.\ \textit{invs'}|\!\},\ \{\!|\lambda rv.\ \textit{invs'}|\!\}
\end{aligned}
$$

Secondly, we establish that all elements of the Init sets are related.

$$(\textsf{new-kernel-state},\ \textsf{newKernelState}) \in \textsf{state-relation}$$

$$
\begin{aligned}
&\textsf{corres}\ \textit{dc}\ (\lambda s.\ s = \textsf{new-kernel-state})\ (\lambda s.\ s = \textsf{newKernelState}) \\
&\qquad (\textsf{init-kernel}\ \textit{entry frames offset kFrames}) \\
&\qquad (\textsf{initKernel}\ \textit{entry frames offset kFrames})
\end{aligned}
$$

Finally we establish that the main execution steps correspond.

$$\textsf{corres}\ (\textit{intr} \oplus \textit{dc})\ \textit{invs}\ \textit{invs'}\ (\textsf{call-kernel}\ \textit{event})\ (\textsf{callKernel}\ \textit{event})$$

From these we can establish forward simulation, which implies refinement. The statements above are slightly simplified versions of our theorems which involve more preconditions on machine behaviour.

6.3 Properties

We now give a description of some of the properties and invariants we proved on these two formalisations in addition to the main refinement theorem that states the concrete level is a correct implementation of the abstract specification.

One of the first properties proved on both levels was that all system calls terminate. Since HOL is a logic of total functions, this is a necessary condition to write down the kernel behaviour. The proof for most of the kernel was straightforward, we had initially only one complex, mutually recursive case: the *delete* operation that removes capabilities. We have since transformed it into a statically bounded recursion such that we can guarantee the absence of stack overflows will at runtime.

The main invariant of the kernel is simple: all references in the kernel, be it in capabilities, kernel objects or other data structures always point to an object of the expected type. This is a dynamic property as memory can be re-typed during runtime. Despite its simplicity, it is the major driver for almost all other kernel invariants. Exceptions are low-level invariants like *address 0 is never inhabited by any object*, or *objects are always aligned to their size.*

The main validity predicates (such as valid-objs and valid-ep mentioned previously) are liftings of the well-typedness criterion above to the entire heap, thread states, scheduler queues and other state components. An example of a more complex invariant, that we required to prove that well-typedness is preserved, is: *A kernel object k_1 contains a reference to kernel object k_2 if and only if there exists a (possibly transitive) reference back from k_2 to k_1*. This symmetry condition can be used to conclude that if an object contains no references itself, there will be no dangling references to it in the rest of the kernel. It would therefore be safe to remove such an object once capability references are checked. To avoid inefficient object state checks, we additionally observe: *If an object is live (contains references to other objects), there exists a capability to it.*

Testing for capabilities is much easier, because they are tracked explicitly in the CDT. CDT-related properties are:

Linked List. The doubly linked list structure is consistent (back/forward pointers are implemented correctly), the lists always terminate in NULL, and the list together with the additonal tags correctly implements a tree. This is a basic shape property.

Chunks. If two CTEs point to the same memory location, they have a common ancestor and all entries between them in the CDT point to this same memory location. This ensures various tests in the kernel can be implemented locally.

Cap Parency. If an untyped capability c_1 covers a sub-region of another capability c_2, then c_1 must be a descendant of c_2 according to the CDT.

Object Parency. If a capability c_1 points to a kernel object whose memory is covered by an untyped capability c_2, then c_1 must be a descendant of c_2.

All of these together ensure that memory can be retyped safely and with minimal local checks: if an untyped capability has no children, then all kernel objects in its region must be non-live (otherwise there would be capabilities to them, which in turn would have to be children). If the objects are not live and no capabilities to them exist, there is no further reference in the whole system that could be made unsafe by the type change.

This example is the most complex chain of invariants we had to create for a single operation. Other operations, such as IPC and scheduling have their own, but simpler requirements.

6.4 Experience and Lessons Learned

The total effort for the refinement proof described here was ca. 100K lines of Isabelle/HOL and 5 person years. The proof lead to over 100 changes in each of the two specifications. The majority of the changes were for ease of proof: slight re-arrangement of code, avoidance of unnecessary optimisations, local tests instead of global assumptions. The majority of actual bugs were typographical and copy & paste errors that slipped through prior testing. Unsurprisingly, there were far more of these simple mistakes on the abstract level than on the executable one. The abstract level was only type checked, never run, since it is not executable. We found on the order of 10 conceptual problems and oversights which would have lead to crashes or security exploits—as would have most of the typos. These were mainly missing checks on user input, subtle side effects in the middle of operations, or (rarely) too-strong assumptions on what is invariant during execution. Security attacks became apparent via invariant violations.

We found that the kernel programming team usually knew the invariants precisely and used them for optimisations. In fact, the developers were often able to articulate clearly why a certain property should hold or why a certain test was unnecessary. A number of the security breaches mentioned above were discovered during these discussions with the developers. On the other hand, it was the formal proof that forced us to have the discussion in the first place.

In terms of lessons learned, we confirm the usual observation that the more abstract the easier, and the less assumptions on global state the easier. In this light, it was expected that the low-level CDT and the large, concrete initialisation phase of the kernel belonged to the more unpleasant parts of the proof.

After an initial full proof of refinement was achieved, we found that new features could be added with reasonable effort. This depends on how independent the new feature is from the rest of the system. If it uses its own data structure that is not accessed anywhere else, the effort is largely proportional to the size of the feature. If, on the other hand, it is highly intermingled with the rest of the system, a factor of the size of the kernel times the size of the feature is to be expected. For instance, adding multiple capability arguments to system calls (as opposed to one only) was easy, with about 2 person weeks of effort, although it concerned changes fairly deep in IPC message decoding and transfer. On the other hand, we hypothesise that the effort for the whole verification so far was quadratic in the size of the kernel. Since in a microkernel almost every basic feature relies on properties of almost all other features (IPC, TCBs, CTEs, CDT are all highly connected), proving preservation of a new invariant on one feature will involve significant work not only on this, but on all other features in the kernel as well. The refinement proof itself remains linear in the size of the kernel. With more modular code, one would expect independent data structures and therefore invariant proofs of a size proportional to the code.

Another observation concerns invariant discovery. We began with a simple invariant that was needed for the refinement proof (well-typedness) and let that drive the invariant discovery process. In hindsight it would, after a short initial phase, have probably been more effective to just write down all the invariants that we suspected we would need in one go. At multiple points we hoped to get away with a simpler formulation, but then were caught out halfway through the kernel by a particular operation after several thousand lines of proof. We then ended up with the complex, precise form anyway. The lesson is: in such a complex system, the simple formulation is unlikely to succeed. Take the precise formulation instead, even if it looks like more work initially. As mentioned above, a good source of invariants is the development team.

In terms of proof engineering and the methods presented in this paper, we believe we have achieved our goal of scalability. Up to four people worked on this proof concurrently and independently without much conflict. We estimate that for code of this size (10K lines of C code) a team with more than five or six persons would need a more serious effort in planning and synchronisation.

Once the framework and the invariants are set up, and more importantly once the kernel is well understood, the proof is not too hard and should be readily repeatable. We see potential for more automation in the refinement proof and in exploratory automatic invariant proofs. Simple invariants can often be stated and proved automatically for many functions at a time. This could automatically be tried for a set of basic properties before manual proof starts. We have developed first steps in this direction, but have not made it the focus so far.

In conclusion, code verification on this level of detail and size of code is entirely feasible with current theorem proving technology.

7 Related Work

Earlier work on OS verification includes PSOS [7] and UCLA Secure Unix [19]. Later, KIT [1] describes verification of process isolation properties down to object code level, but for an idealised kernel far simpler than modern microkernels. The VeriSoft project [9] is attempting to verify a whole system stack, including hardware, compiler, applications, and a simplified microkernel VAMOS. The VFiasco [13] project is attempting to verify the Fiasco kernel, another variant of L4 directly on the C++ level. The Coyotos [17] kernel is being designed for verification, but it is unclear how much progress has been made. Craig [2] provides a number of high-level OS kernel formalisations in Z, but does not attempt any machine-checked proofs which at this size of specification we consider a necessity.

The House and Osker kernels [11] (in Haskell) and the Hello kernel [8] (in Standard ML) demonstrated that modern functional languages can be used to develop bare metal implementations of operating systems. In contrast to that, we see our Haskell implementation of the kernel as a prototype only.

Other approaches to translating Haskell into Isabelle include [10,12,14]. Since none of the approaches were able to parse our code base, we use an own translator. For the work presented here, we need to assume correctness of this translator. In the longer term, this is not needed, because the final theorem will be a refinement theorem between the abstract Isabelle model and the C program. We have already invested significant effort into modelling C precisely [18].

Our treatment of Hoare Logic on monads is much less general than the one of Mossakowski et al. [16]. We do not make assertions part of the program which in our setting would provide barriers for splitting up the same proof among multiple persons. As mentioned before, we trade generality for simplicity, and for lighweight infrastructure with an emphasis on scalability.

8 Conclusion

We have presented simple, but effective techniques for reasoning about state-based functional programs and for proving formal refinement on them. Although we have not aimed at full generality, we are convinced that the combination of basic monads we used covers a wide range of practical programs in languages such as Haskell or ML. Our case study has shown that it is practical to fully formally verify programs of thousands of lines of code at this level.

The salient points of our Hoare Logic are that it is simple enough to be automated effectively; but despite its simplicity, expressive enough to be easily applicable. Our extension of classic refinement to the nondeterministic state monad is formally largely straightforward, and the calculus presented is not complete. Again, the main point is that it is engineered such that a large-scale proof can be effectively divided up into mostly independent parts. Classical step-wise refinement calculi do not necessarily work well within this paradigm and often require window reasoning and other complex context tracking.

The case study we report on constitutes the formal, fully machine-checked verification of a binary-compatible executable model of seL4. Binary compatible means that the corresponding Haskell program, together with a hardware simulator, can execute normal, compiled user-level ARM11 binaries that would run unchanged on bare hardware. This includes low-level hardware feedback: cache flushes, TLB loads, etc. To our knowledge, this is the first such verification of an OS microkernel of this size and complexity.

Although the verification reported on here reaches a level of detail far greater than that usually present when a software system is claimed to be verified, we refrain from calling seL4 itself "fully formally verified" yet. Our goal is to take the verification from the executable model down to the level of C code, compiled and running on hardware.

Acknowledgements We thank the other current and former members of the L4.verified and seL4 teams, Michael Norrish, Jia Meng, Catherine Menon, Jeremy Dawson, Simon Winwood, Harvey Tuch, Rafal Kolanski, David Tsai, Andrew Boyton, Kai Engelhardt, Kevin Elphinstone, Philip Derrin and Dhammika Elkaduwe for their help and support.

References

1. W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
2. I. D. Craig. *Formal Models of Operating System Kernels.* Springer, 2007.
3. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.
4. P. Derrin, K. Elphinstone, G. Klein, D. Cock, and M. M. T. Chakravarty. Running the manual: An approach to high-assurance microkernel development. In *Proc. ACM SIGPLAN Haskell WS*, Portland, OR, USA, Sept. 2006.
5. D. Elkaduwe, P. Derrin, and K. Elphinstone. A memory allocation model for an embedded microkernel. In *Proc. 1st MIKES*, pages 28–34, Sydney, Australia, 2007.
6. K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, and G. Heiser. Towards a practical, verified kernel. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, page 6, San Diego, CA, USA, May 2007.
7. R. J. Feiertag and P. G. Neumann. The foundations of a provably secure operating system (PSOS). In *AFIPS Conf. Proc., 1979National Comp. Conf.*, pages 329–334, New York, NY, USA, June 1979.
8. G. Fu. Design and implementation of an operating system in Standard ML. Master's thesis, Dept. of Information and Computer Sciences, Univ. Hawaii at Manoa, 1999.
9. 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.
10. T. Hallgren, J. Hook, M. P. Jones, and R. B. Kieburtz. An overview of the Programatica Tool Set. High Confidence Software and Systems Conference, 2004.
11. 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.
12. W. L. Harrison and R. B. Kieburtz. The logic of demand in Haskell. *Journal of Functional Programming*, 15(6):837–891, 2005.
13. 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.
14. 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 *LNCS*, pages 147–162. Springer, 2005.
15. J. Liedtke. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, December 1995.
16. T. Mossakowski, L. Schröder, and S. Goncharov. A generic complete dynamic logic for reasoning about purity and effects. In J. Fiadeiro and P. Inverardi, editors, *Fundamental Approaches to Softeware Engineering (FASE 2008)*, volume 4961 of *LNCS*, pages 199–214. Springer, 2008.
17. J. Shapiro. Coyotos. `www.coyotos.org`, 2006.
18. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, page 12, Nice, France, Jan. 2007.
19. B. Walker, R. Kemmerer, and G. Popek. Specification and verification of the UCLA Unix security kernel. *CACM*, 23(2):118–131, 1980.