# Mind the Gap
## A Verification Framework for Low-Level C

Simon Winwood[1,2], Gerwin Klein[1,2], Thomas Sewell[1], June Andronick[1], David Cock[1], and Michael Norrish[1,3]

[1] NICTA*, Australia

[2] School of Computer Science and Engineering, UNSW, Sydney, Australia

[3] Computer Sciences Laboratory, ANU, Canberra, Australia

{first-name.last-name}@nicta.com.au

**Abstract.** This paper presents the formal Isabelle/HOL framework we use to prove refinement between an executable, monadic specification and the C implementation of the seL4 microkernel. We describe the refinement framework itself, the automated tactics it supports, and the connection to our previous C verification framework. We also report on our experience in applying the framework to seL4. The characteristics of this microkernel verification are the size of the target (8,700 lines of C code), the treatment of low-level programming constructs, the focus on high performance, and the large subset of the C programming language addressed, which includes pointer arithmetic and type-unsafe code.

## 1 Introduction

The seL4 kernel [10] is a high-performance microkernel in the L4 family [18], targeted at secure, embedded devices. In verifying such a complex and large – 8,700 lines of C – piece of software, scalability and separation of concerns are of the utmost importance. We show how to achieve both for low-level, manually optimised, real-world C code.

Fig. 1 shows the layers and proofs involved in the verification of seL4. The top layer is an abstract, operational specification of seL4; the middle layer is an executable specification derived automatically [8,11] from a working Haskell prototype of the kernel; the bottom layer is a hand-written and hand-optimised C implementation. The aim is to connect the three layers by formal proof in Isabelle/HOL [21].

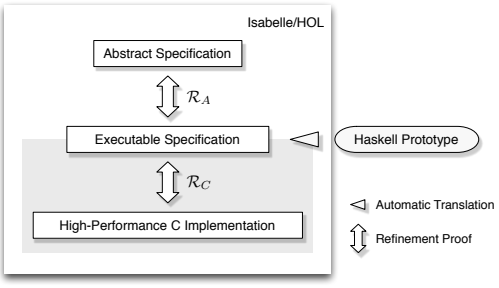


**Fig. 1.** Refinement steps in L4.verified.

* NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program

Previously, we presented a verification framework [5] for proving refinement between the abstract and executable specifications. This paper presents the framework for the second refinement step: the formal, machine-checked proof that the high-performance C implementation of seL4 correctly implements the executable specification.

With these two refinement steps, we manage to isolate two aspects of the verification of seL4. In the first refinement step, which we call $\mathcal{R}_A$, we dealt mostly with semantic concepts such as relationships between data structures and system-global conditions for safe execution. We estimate that 80% of the effort in $\mathcal{R}_A$ was spent on such invariants. In the second refinement step, $\mathcal{R}_C$, the framework we present in this paper allows us to reduce our proof effort and to reuse the properties shown in $\mathcal{R}_A$. The first refinement step established that the kernel design works, the second closes the gap to C.

*Paper Structure* We begin with an example that sketches the details of a typical kernel function. We then explain how the components of the verification framework fit together, summarising relevant details of our earlier work on the monadic, executable specification [5], and on our C semantics and memory model [27,25,26]. In particular, we describe the issues involved in converting the C implementation into Isabelle/HOL. The main part of the paper shows the refinement framework with its fundamental definitions, rules, and automated tactics. We demonstrate the framework's performance by reporting on our experience so far in applying it to the verification of substantial parts of the seL4 C implementation (474 out of 518 functions, 91%).

## 2 Example

The seL4 kernel [10] provides the following operating system kernel services: inter-process communication, threads, virtual memory, access control, and interrupt control. In this section we present a typical function, cteMove, with which we will illustrate the verification framework.

Access control in seL4 is based on *capabilities*. A capability contains an object reference along with access rights. A *capability table entry* (CTE) is a kernel data structure with two fields: a capability and an *mdbNode*. The latter is book-keeping information and contains a pair of pointers which form a doubly linked list.

The cteMove operation, shown in Fig. 2, moves a capability table entry from *src* to *dest*. The left-hand side of the figure shows the executable specification in Isabelle/HOL, while the right-hand side shows the corresponding C code.

The first 6 lines in Fig. 2 initialise the destination entry and clear the source entry; the remainder of the function updates the pointers in the doubly linked list. During the move, the capability in the entry may be diminished in access rights. Thus, the argument *cap* is this possibly diminished capability, previously retrieved from the entry at *src*.

In this example, the C source code is structurally similar to the executable specification. This similarity is not accidental: the executable specification describes the low-level design with a high degree of detail. Most of the kernel

```
cteMove cap src dest ≡
do
  cte ← getCTE src;
  mdb ← return (cteMDBNode cte);
  updateCap dest cap;
  updateCap src NullCap;
  updateMDB dest (const mdb);
  updateMDB src (const nullMDBNode);

  updateMDB
    (mdbPrev mdb)
    (λm. m (| mdbNext := dest |));

  updateMDB
    (mdbNext mdb)
    (λm. m (| mdbPrev := dest |))
od
```

```
void cteMove (cap_t newCap,
              cte_t *srcSlot, cte_t *destSlot){
  mdb_node_t mdb; uint32_t prev_ptr, next_ptr;
  mdb = srcSlot->cteMDBNode;
  destSlot->cap = newCap;
  srcSlot->cap = cap_null_cap_new();
  destSlot->cteMDBNode = mdb;
  srcSlot->cteMDBNode = nullMDBNode;
  prev_ptr = mdb_node_get_mdbPrev(mdb);
  if(prev_ptr) mdb_node_ptr_set_mdbNext(
                 &CTE_PTR(prev_ptr)->cteMDBNode,
                 CTE_REF(destSlot));
  next_ptr = mdb_node_get_mdbNext(mdb);
  if(next_ptr) mdb_node_ptr_set_mdbPrev(
                 &CTE_PTR(next_ptr)->cteMDBNode,
                 CTE_REF(destSlot));
 }
```

**Fig. 2.** cteMove: executable specification and C implementation

functions exhibit this property. Even so, the implementation here makes a small optimisation: in the specification, updateMDB always checks that the given pointer is not NULL. In the implementation this check is done for `prev_ptr` and `next_ptr` – which may be `NULL` – but omitted for `srcSlot` and `destSlot`. In verifying cteMove we will have to prove these checks are not required.

## 3 The Executable Specification Environment

Operations in the executable specification of seL4, such as cteMove, are written in a monadic style inspired by Haskell. The type constructor $'a$ kernel is a monad representing computations returning a value of type $'a$; such values can be injected into the monad using the return :: $'a \Rightarrow 'a$ kernel operation. The composition operator, bind :: $'a$ kernel $\Rightarrow$ ($'a \Rightarrow 'b$ kernel) $\Rightarrow 'b$ kernel, evaluates the first operation and makes the return value available to the second operation. The ubiquitous do ... od syntax seen in Fig. 2 is syntactic sugar for a sequence of operations composed using bind. There are also operations for accessing and mutating k-state, the underlying state.

The type $'a$ kernel is isomorphic to k-state $\Rightarrow$ ($'a \times$ k-state) set $\times$ bool. The motivation for, and formalisation of, this monad are detailed in earlier work [5]. In summary, we take a conventional state monad and add nondeterminism and a failure flag. Nondeterminism, required to model some interactions between kernel and hardware, is modelled by allowing a set of possible outcomes in the return type. The boolean failure flag is used to indicate unrecoverable errors and invalid assertions, and is set only by the fail :: $'a$ kernel operation. The destructors mResults and mFailed access, respectively, the set of outcomes and the failure flag of a monadic operation evaluated at a state.

The specification environment provides a verification condition generator (VCG) for judgements of the form $\{\!|P|\!\}\ a\ \{\!|R|\!\}$, and a refinement calculus for the monadic model. One feature of this calculus is that the refinement property

cannot hold if the failure flag is set by the executable specification, thus $\mathcal{R}_A$ implies non-failure of the executable level. In particular, this allows all assertions in the executable specification to be taken as assumptions in the proof of $\mathcal{R}_C$.

## 4 Embedding C

In this section we describe our infrastructure for parsing C into Isabelle/HOL and for reasoning about the result. The seL4 kernel is implemented almost entirely in C99 [16]. Direct hardware accesses are encapsulated in machine interface functions, some of which are implemented in ARMv6 assembly. In the verification, we axiomatise the assembly functions using Hoare triples.

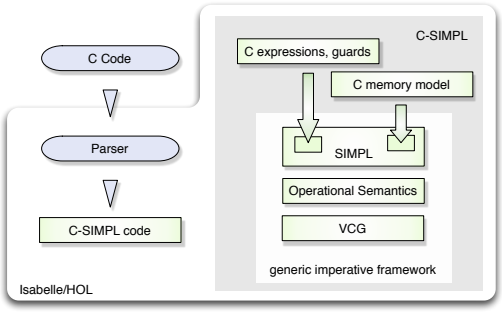


**Fig. 3.** C language framework.

Fig. 3 gives an overview of the components involved. The right-hand side shows our instantiation of SIMPL [23], a generic, imperative language inside Isabelle/HOL. The SIMPL framework provides a program representation, a semantics, and a VCG. This language is generic in its expressions and state space. We instantiate both components to form C-SIMPL, with a precise C memory model and C expressions, generated by a parser. The left-hand side of Fig. 3 shows this process: the parser takes a C program and produces a C-SIMPL program.

### 4.1 The SIMPL Framework

SIMPL provides a data type and semantics for statement forms; expressions are shallowly embedded. The following is a summary of the relevant SIMPL syntactic forms, where $e$ represents an expression

$$\begin{aligned} c \triangleq\ & \mathsf{SKIP} \mid \text{´}v :== e \mid c_1 \ ; \ c_2 \mid \mathsf{IF}\ e\ \mathsf{THEN}\ c_1\ \mathsf{ELSE}\ c_2\ \mathsf{FI} \mid \mathsf{WHILE}\ e\ \mathsf{DO}\ c\ \mathsf{OD} \\ & \mid \mathsf{TRY}\ c_1\ \mathsf{CATCH}\ c_2\ \mathsf{END} \mid \mathsf{THROW} \mid \mathsf{Call}\ f \mid \mathsf{Guard}\ F\ P\ c \end{aligned}$$

The semantics are canonical for an imperative language. The $\mathsf{Guard}\ F\ P\ c$ statement throws the fault $F$ if the condition $P$ is false and executes $c$ otherwise.

Program states in SIMPL are represented by Isabelle records. The record contains a field for each local variable in the program, and a field *globals* containing all global variables and the heap. Variables are then simply functions on the state. SIMPL includes syntactic sugar for dealing with such functions: the term *´srcSlot* refers to the local variable *srcSlot* in the current state. For example, the set of program states where *srcSlot* is NULL is described by $\{\!|\text{´}srcSlot = \mathsf{NULL}|\!\}$.

The semantics are represented by judgements of the form $\Gamma \vdash \langle c,x\rangle \Rightarrow x'$ which means that executing statement $c$ in state $x$ terminates and results in state $x'$; the parameter $\Gamma$ maps function names to function bodies. Both $x$ and $x'$ are *extended states*: for normal program states, Normal $s$, the semantics are as expected; abrupt termination states (Abrupt $s$) are propagated until a surrounding TRY ... CATCH ... END statement is reached; and Stuck and Fault $u$ states, generated by calls to non-existent procedures and failed Guard statements respectively, are passed through unchanged. Abrupt states are generated by THROW statements and are used to implement the C statements `return`, `break`, and `continue`.

The SIMPL environment also provides a VCG for partial correctness triples; Hoare-triples are represented by judgements of the form $\Gamma \vdash_{/F} P\ c\ C,A$, where $P$ is the precondition, $C$ is the postcondition for normal termination, $A$ is the postcondition for abrupt termination, and $F$ is the set of ignored faults; if $F$ is $\mathcal{U}$, the universal set, then all Guard statements are effectively ignored. Both $A$ and $F$ may be omitted if empty.

### 4.2 The memory model

Our C subset allows type-unsafe operations including casts. To achieve this soundly, the underlying heap model is a function from addresses to bytes. This allows, for example, the C function `memset`, which sets each byte in a region of the heap to a given value. We use the abbreviation $\mathcal{H}$ for the heap in the current state; the expression $\mathcal{H}\ p$ reads the object at pointer $p$, while $\mathcal{H}(p \mapsto v)$ updates the heap at pointer $p$ with value $v$.

While this model is required for such low-level memory accesses, it is too cumbersome for routine verification. By extending the heap model with typing information and using tagged pointers we can lift bytes in the heap into Isabelle terms. Pointers, terms of type $'a$ ptr, are raw addresses wrapped by the polymorphic constructor Ptr; the phantom type $'a$ carries the type information. Pointers may be unwrapped via the ptr-val function, which simply extracts the enclosed address. Struct field addressing is also supported: the pointer $\&(p\rightarrow[f])$ refers to the field $f$ at the address associated with pointer $p$. The details of this memory model are described by Tuch *et al* [27,26].

### 4.3 From C to C-SIMPL

The parser translates the C kernel into a C-SIMPL program. This process generally results in a C-SIMPL program that resembles the input. Here we describe the C subset we translate, and discuss those cases where translation produces a result that is not so close to the input.

**Our C Subset** As mentioned above, local variables in SIMPL are represented by record fields. It is therefore not meaningful to take their address in the framework, and so the first restriction of our C subset is that local variables may not have their addresses taken. Global variables may, however, have their addresses taken.

As we translate all of the C source at once, the parser can determine exactly which globals do have their addresses taken, and these variables are then given addresses in the heap. Global variables that do not have their addresses taken are, like locals, simply fields in the program state. The restriction on local variables could be relaxed at the cost of higher reasoning overhead.

The other significant syntactic omissions in our C subset are `union` types, bitfields, `goto` statements, and `switch` statements that allow cases to fall-through. We handle `union` types and bitfields with an automatic code generator [4], described in Sect. 6, that implements these types with structs and casts. Furthermore, we do not allow function calls through function pointers and take care not to introduce a more deterministic evaluation order than C prescribes. For instance, we translate the side-effecting C expressions `++` and `--` as statements.

**Internal Function Calls and Automatic Modifies Proofs** SIMPL does not permit function calls within expressions. If a function call appears within an expression in the input C, we lift it out and transform it into a function call that will occur before the expression is evaluated. For example, given a global variable `x`, the statement `z = x + f(y)` becomes `tmp = f(y); z = x + tmp`, where `tmp` is a new temporary variable.

This translation is only sound when the lifted functions are side-effect free: evaluation of the functions within the original expression is linearised, making the translated code more deterministic than warranted by the C semantics. The parser thus generates a VCG "modifies" proof for each function, stating which global variables are modified by the function. Any function required to be side-effect free, but not proved as such, is flagged for the verification team's attention.

**Guards and Short-Circuit Expressions** Our parser uses Guard statements to force verifiers to show that potentially illegal conditions are avoided. For example, expressions involving pointer dereferences are enclosed by guards which require the pointer to be aligned and non-zero.

Guards are statement-level constructors, so whole expressions accumulate guards for their sub-expressions. However, C's short-circuiting expression forms (`&&`, `||` and `?:`) mean that sub-expressions are not always evaluated. We translate such expressions into a sequence of `if`-statements, linearising the evaluation of the expression. When no guards are involved, the expression in C can become a C-SIMPL expression, using normal, non-short-circuiting, boolean operators.

*Example* While we have shown the C implementation in the example Fig. 2, refinement is proven between the executable specification and the imported C-SIMPL code. For instance, the assignment `mdb = srcSlot->cteMDBNode` in Fig. 2 is translated into the following statement in C-SIMPL

$$\begin{array}{l}\mathsf{MemGuard}\ \&(\acute{}\mathit{srcSlot}{\rightarrow}[\mathsf{cteMDBNode\text{-}C}]) \\ \quad (\acute{}\mathit{mdb} :== \mathcal{H}\ \&(\acute{}\mathit{srcSlot}{\rightarrow}[\mathsf{cteMDBNode\text{-}C}]))\end{array}$$

The MemGuard constructor abbreviates the alignment and non-`NULL` conditions for pointers.

## 5 Refinement

Our verification goal is to prove refinement between the executable specification and the C implementation. Specifically, this means showing that the C kernel entry points for interrupts, page faults, exceptions, and system calls refine the executable specification's top-level function callKernel. We show refinement using a variation of forward simulation [7] we call *correspondence*: evaluation of corresponding functions takes related states to related states.

In previous work [5], while proving $\mathcal{R}_A$, we found it useful to divide the proof along the syntactic structure of both programs as far as possible, and then prove the resulting subgoals semantically. Splitting the proof has two main benefits: firstly, it is a convenient unit of proof reuse, as the same pairing of abstract and concrete functions recurs frequently for low-level functions; and secondly, it facilitates proof development by multiple people. One important feature of this approach is that preconditions are discovered lazily *à la* Dijkstra [9]. Rules for showing correspondence typically build preconditions from those of the premises.

In this section we describe the set of tools and techniques we developed to ease the task of proving correspondence in $\mathcal{R}_C$. First, we give our definition of correspondence, followed by a discussion of the use of the VCG. We then describe techniques for reusing proofs from $\mathcal{R}_A$ to solve proof obligations from the implementation. Next, we present our approach for handling operations with no corresponding analogue. Finally, we describe our splitting approach and sketch the proof of the example.

### 5.1 The correspondence statement

In practice, the definition of correspondence is more complex than simply linking related states, as: (1) verification typically requires preconditions to hold of the initial states; (2) we allow early returns from functions and breaks from loops; and (3) function return values must be related.

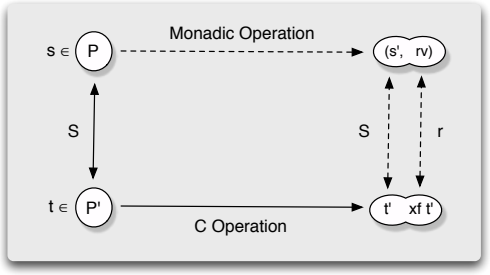


**Fig. 4.** Correspondence.

To deal with early return, we extend the semantics to lists of statements, using the judgement $\Gamma \Vdash \langle c{\cdot}hs,\ s\rangle \Rightarrow x'$. The statement sequence $hs$ is a *handler stack*; it collects the CATCH handlers which surround usages of the statements `return`, `continue`, and `break`. If $c$ terminates abruptly, each statement in $hs$ is executed in sequence until one terminates normally.

Relating the return values of functions is dealt with by annotating the correspondence statement with a *return value relation* $r$. Although evaluating a monadic operation results in both a new state and a return value, functions in C-SIMPL return values by updating a function-specific local variable; because

local variables are fields in the state record, this is a function from the state. We thus annotate the correspondence statement with an *extraction function xf*, a function which extracts the return value from a program state.

The correspondence statement is illustrated in Fig. 4 and defined below

$$\begin{aligned}&\mathsf{ccorres}\ r\ xf\ P\ P'\ hs\ a\ c \equiv\\ &\quad \forall (s,\ t)\in\mathcal{S}.\ \forall\, t'.\ s \in P \wedge t \in P' \wedge \neg\ \mathsf{mFailed}\ (a\ s) \wedge \Gamma \Vdash \langle c\cdot hs,\ t\rangle \Rightarrow t'\\ &\quad \longrightarrow \exists\, (s',rv)\in\mathsf{mResults}\ (a\ s).\\ &\qquad\qquad \exists\, t'_N.\ t' = \mathsf{Normal}\ t'_N \wedge (s',\ t'_N) \in \mathcal{S} \wedge r\ rv\ (xf\ t'_N)\end{aligned}$$

The definition can be read as follows: given related states $s$ and $t$ with the preconditions $P$ and $P'$ respectively, if the abstract specification $a$ does not fail when evaluated at state $s$, and the concrete statement $c$ evaluates under handler stack $hs$ in extended state $t$ to extended state $t'$, then the following must hold:

1. evaluating $a$ at state $s$ returns some value $rv$ and new abstract state $s'$;
2. the result of the evaluation of $c$ is some extended state $\mathsf{Normal}\ t'_N$, that is, not Abrupt, Fault, or Stuck;
3. states $s'$ and $t'_N$ are related by the state relation $\mathcal{S}$; and
4. values $rv$ and $xf\ t'_N$ – the extraction function applied to the final state of $c$ – are related by $r$, the given return value relation.

Note that $a$ is non-deterministic: we may pick any suitable $rv$ and $s'$. As mentioned in Sect. 3, the proof of $\mathcal{R}_A$ entails that the executable specification does not fail. Thus, in the definition of ccorres, we may assume $\neg$ mFailed $(a\ s)$. In practice, this means assertions and other conditions for (non-)failure in the executable specification become known facts in the proof. For example, the operation getCTE *srcSlot* in the example in Fig. 2 will fail if there is no CTE at *srcSlot*. We can therefore assume in the refinement proof that such an object exists. Of course, these facts are only free because we have already proven them in $\mathcal{R}_A$.

*Example* To prove correspondence for cteMove, we must, after unfolding the function bodies, show the statement in Fig. 5. The cteMove operation has no return value, so our extraction function (*xf* in the definition of ccorres above) and return relation ($r$ above) are trivial. The specification precondition ($P$ above) is the system invariant *invs*, while the implementation precondition ($P'$ above) relates the formal parameters *destSlot*, *srcSlot*, and *newCap* to the specification arguments *dest*, *src*, and *cap* respectively. As all functions are wrapped in a TRY ... CATCH SKIP block to handle `return` statements, the handler stack is the singleton list containing SKIP.

### 5.2 Proving correspondence via the VCG

Data refinement predicates can, in general [7], be rephrased and solved as Hoare triples. We do this in our framework by using the VCG after applying the following rule

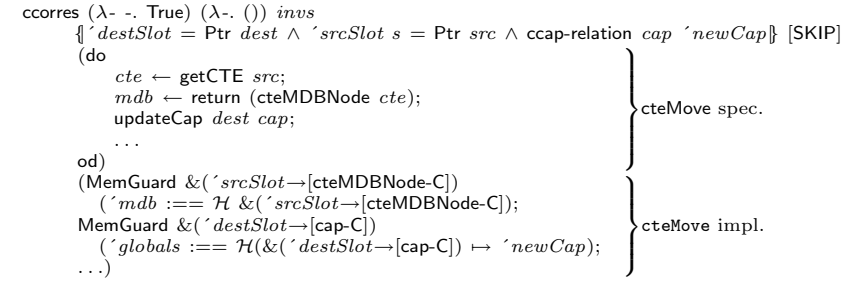


**Fig. 5.** The cteMove correspondence statement.

$$\frac{\forall\, s.\ \Gamma \vdash_{c} \{t \mid s \in P \wedge t \in P' \wedge (s,\, t) \in \mathcal{S}\} \quad \{t' \mid \exists\, (rv,\, s') \in \mathsf{mResults}\ (a\ s).\ (s',\, t') \in \mathcal{S} \wedge r\ rv\ (xf\ t')\}}{\mathsf{ccorres}\ r\ xf\ P\ P'\ hs\ a\ c}$$

In essence, this rule states that to show correspondence between $a$ and $c$, for a given initial specification state $s$, it is sufficient to show that executing $c$ results in normal termination where the final state is related to the result of evaluating $a$ at $s$. The VCG precondition can assume that the initial states are related and satisfy the correspondence preconditions.

Use of this rule in verifying correspondence is limited by two factors. Firstly, the verification conditions produced by the VCG may be excessively large or complex. Our experience is that the output of a VCG step usually contains a separate term for every possible path through the target code, and that the complexity of these terms tends to increase with the path length. Secondly, the specification return value and result state are existential, and thus outside the range of our extensive automatic support for showing universal properties of specification fragments. Fully expanding the specification is always possible, and in the case of deterministic operations will yield a single state/return value pair, but the resulting term structure may also be large.

In the case of our example, the goal produced by the VCG has 377 lines before unfolding the specification and 800 lines afterward. Verifying such non-trivial functions is made practical by the approach described in the remainder of this section.

### 5.3 Local variable lifting

The feasibility of proving $\mathcal{R}_C$ depends heavily on proof reuse from $\mathcal{R}_A$. Consider the following rule for dealing with a guard introduced by the parser (see Sect. 4.3)

$$\frac{\mathsf{ccorres}\ r\ xf\ G\ G'\ hs\ a\ c}{\mathsf{ccorres}\ r\ xf\ (G \cap \mathsf{cte\text{-}at}'\ (\mathsf{ptr\text{-}val}\ p))\ G'\ hs\ a\ (\mathsf{MemGuard}\ (\lambda s.\ p)\ c)}$$

This states that the proof obligation introduced by MemGuard at the CTE pointer $p$ can be discharged, assuming that there exists a CTE object on the

specification side (denoted cte-at′ (ptr-val $p$)); this rule turns a proof obligation from the implementation into an assumption of the specification. There is, however, one major problem: the pointer $p$ cannot depend on the C state, because it is also used on the specification side.

To see why this is such a problem, recall that local variables in C-SIMPL are fields in the state record; any pointer, apart from constants, in the program will *always* refer to the state, making the above rule inapplicable; in the example, the first guard refers to the local variable *´srcSlot*.

All is not lost, however: the values in local variables generally correspond to some value available in the specification. We have developed an approach that automatically replaces such local variables with new HOL variables representing their value. Proof obligations which refer to the local variable can then be solved by facts about the related value from the specification precondition. We call this process *lifting*.

*Example* If we examine the preconditions to the example proof statement in Fig. 5, we note the assumption *´srcSlot* = Ptr *src* and observe that *srcSlot* depends on the C state. By lifting this local variable and substituting the assumption, we get the following implementation fragment

```
MemGuard &(Ptr src→[cteMDBNode-C])
  (´mdb :== H &(Ptr src→[cteMDBNode-C]);
...
```

The pointer Ptr *src* no longer depends on the C state and is a value from the specification side, so the MemGuard can be removed with the above rule.

Lifting is only sound if the behaviour of the lifted code fragment is indistinguishable from that of the original code; the judgement $d' \sim d[v/f]$ states that replacing applications of the function $f$ in statement $d$ with value $v$ results in the equivalent statement $d'$. This condition is defined as follows

$$d' \sim d[v/f] \equiv \forall\, t\ t'.\ f\ t = v \longrightarrow \Gamma\vdash \langle d,\mathsf{Normal}\ t\rangle \Rightarrow t' = \Gamma\vdash \langle d',\mathsf{Normal}\ t\rangle \Rightarrow t'$$

This states that $d$ and $d'$ must be semantically equivalent, assuming $f$ has the value $v$ in the initial state. In practice, $d'$ depends on a locally bound HOL variable; in such cases, it will appear as $d'\ v$.

Lifting is accomplished through the following rule

$$\frac{\forall\, v.\ d'\ v \sim d[v/f] \qquad \forall\, v.\ P\ v \longrightarrow \mathsf{ccorres}\ r\ xf\ G\ G'\ hs\ a\ (d'\ v)}{\mathsf{ccorres}\ r\ xf\ G\ (G' \cap \{s \mid P\ (f\ s)\})\ hs\ a\ d}$$

Note that $d'$, the lifted fragment, appears only in the assumptions; proving the first premise involves inventing a suitable candidate. We have developed tactic support for automatically calculating the lifted fragment and discharging such proof obligations, based on a set of syntax-directed proof rules.

### 5.4 Symbolic execution

The specification and implementation do not always match: there may be fragments on either side that are artefacts of the particular model. In our example, it is clear that the complex function getCTE has no direct analogue; the implementation accesses the heap directly.

In both cases we have rules to symbolically execute the code using the appropriate VCG, although we must also show that the fragment preserves the state relation. On the implementation side this case occurs frequently; in the example we have the `cap_null_cap_new`, `mdb_node_get_mdbNext`, and `mdb_node_get_mdbPrev` functions. We have developed a tactic which can symbolically execute any side-effect free function which has a VCG specification. This tactic also takes advantage of variable lifting: the destination local variable is replaced by a new HOL variable and we gain the assumption that the variable satisfies the function's postcondition.

### 5.5 Splitting

If we examine our example, there is a clear match between most lines. Splitting allows us to take advantage of this structural similarity by considering each match in isolation; formally, given the specification fragment $\mathsf{do}\ rv \leftarrow a;\ b\ rv\ \mathsf{od}$ and the implementation fragment $c$; $d$, splitting entails proving a first correspondence between $a$ and $c$ and a second between $b$ and $d$.

In the case where we can prove that $c$ terminates abruptly, we discard $d$. Otherwise, the following rule is used

$$\frac{\begin{array}{c}\mathsf{ccorres}\ r'\ xf'\ P\ P'\ hs\ a\ c \qquad \forall v.\ d'\ v \sim d[v/xf'] \\ \forall rv\ rv'.\ r'\ rv\ rv' \longrightarrow \mathsf{ccorres}\ r\ xf\ (Q\ rv)\ (Q'\ rv\ rv')\ hs\ (b\ rv)\ (d'\ rv') \\ \{\!|R|\!\}\ a\ \{\!|Q|\!\} \qquad \Gamma \vdash_{/\mathcal{U}} R'\ c\ \{s \mid \forall rv.\ r'\ rv\ (xf'\ s) \longrightarrow s \in Q'\ rv\ (xf'\ s)\}\end{array}}{\mathsf{ccorres}\ r\ xf\ (P \cap R)\ (P' \cap R')\ hs\ (\mathsf{do}\ rv \leftarrow a;\ \ b\ rv\ \mathsf{od})\ (c;\ d)}$$

In the second correspondence premise, $d'$ is the result of lifting $xf'$ in $d$; this enables the proof of the second correspondence to use the result relation from the first correspondence. To calculate the final preconditions, the rule includes VCG premises to move the preconditions from the second correspondence across $a$ and $c$. In the C-SIMPL VCG obligation, we may ignore any guard faults as their absence is implied by the first premise. In fact, in most cases the C-SIMPL VCG step can be omitted altogether, because the post condition collapses to true after simplifications.

We have developed a tactic which assists in splitting: C-SIMPL's encoding of function calls and struct member updates requires multiple specialised rules. The tactic symbolically executes and moves any guards if required, determines the correct splitting rule to use, instantiates the extraction function, and lifts the second correspondence premise.

*Example* After lifting, moving the guard, and symbolically executing the getCTE function, applying the above rule to the example proof statement in Fig. 5 gives the following as the first proof obligation

$$\begin{array}{l}\mathsf{ccorres\ cmdb\text{-}relation}\ mdb\ (\ldots)\ \{\ldots\}\ [\mathsf{SKIP}]\\ \quad (\mathsf{return}\ (\mathsf{cteMDBNode}\ cte))\\ \quad (\acute{}mdb :== \mathcal{H}\ \&(\mathsf{Ptr}\ src{\rightarrow}[\mathsf{cteMDBNode\text{-}C}])\end{array}$$

This goal, proved using the VCG approach from Sect. 5.2, states that, apart from the state correspondence, the return value from the specification side (cteMDBNode $cte$) and implementation side ($\mathcal{H}$ &(Ptr $src$→[cteMDBNode-C]) stored in $mdb$) are related through cmdb-relation, that is, the linked list pointers in the returned specification node are equal to those in the implementation.

### 5.6 Completing the example

The proof of the example, cteMove, is 25 lines of Isabelle/HOL tactic style proof. The proof starts by weakening the preconditions (here abbreviated $P$ and $P'$) with new Isabelle schematic variables; this allows preconditions to be calculated on demand in the correspondence proofs.

We then lift the function arguments and proceed to prove by splitting; the leaf goals are proved as separate lemmas using the C-SIMPL VCG. Next, the correspondence preconditions are moved back through the statements using the two VCGs on specification and implementation. The final step is to solve the proof obligation generated by the initial precondition weakening: the stated preconditions (our $P$ and $P'$) must imply the calculated preconditions.

The lifting and splitting phase takes 9 lines, the VCG stage takes 1 line, using tactic repetition, while the final step takes 15 lines and is typically the trickiest part of any correspondence proof.

## 6 Experience

In this section we explore how our C subset influenced the kernel implementation and performance. We then discuss our experience in applying the framework.

We chose to implement the C kernel manually, rather than synthesising it from the executable specification. Initial investigations had shown that generated C code would not meet the performance requirements of a real-world microkernel. Message-passing (IPC) performance, even in the first hand-written version, completed after two person months, was slow, on the order of the Mach microkernel. After optimisation, this operation is now comparable to that of the modern, commercially deployed, OKL4 2.1 [22] microkernel: we measured 206 cycles for OKL4's hand-crafted assembly IPC path, and 756 cycles for its non-optimised C version on the ARMv6 Freescale i.MX31 platform. On the same hardware, our C kernel initially took over 3000 cycles, after optimisations 299. The fastest other IPC implementation for ARMv6 in C we know of is 300 cycles.

The C subset and the implementation developed in parallel, influencing each other. We extended the subset with new features such as multiple side-effect free function calls in expressions, but we also needed to make trade-offs such as for references to local variables. We avoided passing large structures on the C

**Table 1.** Code and proof statistics. Changes for step $\mathcal{R}_C$.

| | Lines | | | Changes | |
|---|---|---|---|---|---|
| | Haskell/C | Isabelle | Proof | Bugs | Convenience |
| Executable specification | 5,700 | 13,000 | 117,000 | 8 | 10 |
| Implementation | 8,700 | 15,000 | 50,000[a] | 97 | 34 |

[a] With 474 of 518 (91%) of the functions verified.

stack across function boundaries. Instead, we stored these in global variables and accessed them through pointers. Whilst the typical pattern was of conflicting pressures from implementation and verification, in a few cases both sides could be neatly satisfied by a single solution. We developed a code-generation tool [4] for efficient, packed bitfields in tagged unions with a clean, uniform interface. This tool not only generates the desired code, but also the associated Isabelle/HOL proofs and specifications that integrate directly into our refinement framework. The resulting compiled code is faster, more predictable, and more compact than the bitfield code emitted by GCC on ARM.

Code and proof statistics are shown in Table 1. Of the 50,000 lines of proof in $\mathcal{R}_C$, approximately 5,000 lines are framework related, 7,700 lines are automatically generated by our bitfield tool, and the remaining 37,300 lines are hand-written. We also have about 1,000 lines of tactic code. We spent just over 2 person years in 6 months of main activity on this proof and estimate another two months until completion. We prove an average of 3–4 functions per person per week.

One important aspect of the verification effort was our ability to change both the specification and the implementation. These changes, included in Table 1, fell into two categories: true bug fixes and proof convenience changes. In the specification, bug fixes were not related to safety — the proof of $\mathcal{R}_A$ guaranteed this. Rather, they export implementation restrictions such as the number of bits used for a specific argument encoding. Although both versions were safe, refinement was only possible with the changed version. The implementation had not been intensively tested, because it was scheduled for formal verification. It had, however, been used in a number of student projects and was being ported to the x86 architecture when verification started. The former activities found 16 bugs in the ARMv6 code; the verification has so far found 97. Once the verification is complete, the only reason to change the code will be for performance and new features: C implementation defects will no longer exist.

A major aim in developing the framework presented in this paper was the avoidance of invariant proofs on the implementation. We achieved this primarily through proof reuse from $\mathcal{R}_A$: the detailed nature of the executable specification's treatment of kernel objects meant that the state relation fragment for kernel objects was quite simple; this simplicity allowed proof obligations from the implementation to be easily solved with facts from the specification. Furthermore, when new invariants were required we could prove them on the executable

specification. For example, the encoding of Isabelle's option type using a default value in C (such as `NULL`) required us to show that these default values never occurred as valid values.

We discovered that the difficulty of verifying any given function in $\mathcal{R}_C$ was determined by the degree of difference between the function in C and its executable specification, arising either from the control structures of C or its impure memory model. Unlike the proof of $\mathcal{R}_A$, the semantic complexity of the function seems mostly irrelevant. For instance, the operation which deletes a capability — by far the most semantically complex operation in seL4 — was straightforward to verify in $\mathcal{R}_C$. On the other hand, a simpler operation which employs an indiscriminate `memset` over a number of objects was comparatively difficult to verify. It is interesting to note that, even here, proofs from $\mathcal{R}_A$ were useful in proving facts about the implementation.

An important consequence of the way we split up proofs is that local reasoning becomes possible. No single person needed a full, global understanding of the whole kernel implementation.

## 7 Related work

Earlier work on OS verification includes PSOS [12] and UCLA Secure Unix [28]. Later, Bevier [3] describes verification of process isolation properties down to object code level, but for an idealised kernel (KIT) far simpler than modern microkernels. We use the same general approach — refinement — as KIT and UCLA Secure Unix, however the scale, techniques for each refinement step, and level of detail we treat are significantly different.

The Verisoft project [24] is working towards verifying a whole system stack, including hardware, compiler, applications, and a simplified microkernel VAMOS. The VFiasco [15] project is attempting to verify the Fiasco kernel, another variant of L4 directly on the C++ level. For a comprehensive overview of operating system verification efforts, we refer to Klein [17].

Deductive techniques to prove annotated C programs at the source code level include Key-C [20], VCC [6], and Caduceus [13], recently integrated into the Frama-C framework [14]. Key-C only focuses on a type-safe subset of C. VCC, which also supports concurrency, appears to be heavily dependent on large axiomatisations; even the memory model [6] axiomatises a weaker version of what Tuch proves [26]. Caduceus supports a large subset of C, with extensions to handle certain kinds of unions and casts [1,19]. These techniques are not directly applicable to refinement, although Caduceus has at least been used [2] to extract a formal Coq specification for verifying security and safety properties.

We directly use the SIMPL verification framework [23] from the Verisoft project, but we instantiate it differently. While Verisoft's main implementation language is fully formally defined from the ground up, with well-defined Pascal-like semantics and C-style syntax, we treat a true, large subset of C99 [16] on ARMv6 with all the realism and ugliness this implies. Our motivation for this is our desire to use standard tool-chains and compilers for real-world deployment

of the kernel. Verisoft instead uses its own non-optimising compiler, which in exchange is formally verified. Another difference is the way we exploit structural similarities between our executable specification and C implementation. Verisoft uses the standard VCG-based methodology for implementation verification. Our framework allows us to transport invariant properties and Hoare-triples from our existing proof on the executable specification [5] down to the C level. This allowed us to avoid invariants on the C level, speeding up the overall proof effort significantly.

## 8 Conclusion

We have presented a formal framework for verifying the refinement of a large, monadic, executable specification into a low-level, manually performance-optimised C implementation. We have demonstrated that the framework performs well by applying it to the verification of the seL4 microkernel in Isabelle/HOL, and by completing a large part of this verification in a short time. The framework allows us to take advantage of the large number of invariants proved on the specification level, thus saving significant amounts of work. We were able to conduct the semantic reasoning on the more pleasant monadic, shallowly embedded specification level, and leave essentially syntactic decomposition to the C level.

We conclude that our C verification framework achieves both the scalability in terms of size, as well as the separation of concerns that is important for distributing such a large proof over multiple people.

*Acknowledgements* We thank Timothy Bourke and Philip Derrin for reading and commenting on drafts of this paper.

## References

1. J. Andronick. *Modélisation et Vérification Formelles de Systèmes Embarqués dans les Cartes à Microprocesseur—Plate-Forme Java Card et Système d'Exploitation.* PhD thesis, Université Paris-Sud, Mar. 2006.
2. J. Andronick, B. Chetali, and C. Paulin-Mohring. Formal verification of security properties of smart card embedded source code. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM*, volume 3582 of *LNCS*, pages 302–317. Springer, 2005.
3. W. R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
4. D. Cock. Bitfields and tagged unions in C: Verification through automatic generation. In B. Beckert and G. Klein, editors, *Proc, 5th VERIFY*, volume 372 of *CEUR Workshop Proceedings*, pages 44–55, Sydney, Australia, Aug. 2008.
5. D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Proc. 21st TPHOLs*, volume 5170 of *LNCS*, pages 167–182. Springer, Aug. 2008.
6. E. Cohen, M. Moskał, W. Schulte, and S. Tobies. A precise yet efficient memory model for C. `http://research.microsoft.com/apps/pubs/default.aspx?id=77174`, 2008.

7. 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.
8. 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.
9. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8):453–457, 1975.
10. 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*, 2007.
11. K. Elphinstone, G. Klein, and R. Kolanski. Formalising a high-performance microkernel. In R. Leino, editor, *VSTTE*, Microsoft Research Technical Report MSR-TR-2006-117, pages 1–7, Seattle, USA, Aug. 2006.
12. R. J. Feiertag and P. G. Neumann. The foundations of a provably secure operating system (PSOS). In *AFIPS Conf. Proc., 1979 National Comp. Conf.*, pages 329–334, New York, NY, USA, June 1979.
13. J.-C. Filliâtre and C. Marché. Multi-prover verification of C programs. In *Proc. 6th ICFEM*, volume 3308 of *LNCS*, pages 15–29. Springer, 2004.
14. Frama-C. `http://frama-c.cea.fr/`, 2008.
15. M. Hohmuth and H. Tews. The VFiasco approach for a verified operating system. In *Proc. 2nd ECOOP-PLOS Workshop*, Glasgow, UK, Oct. 2005.
16. *Programming languages—C*, 1999. ISO/IEC 9899:1999.
17. G. Klein. Operating system verification—An overview. *Sādhanā*, 34(1):27–69, 2009.
18. J. Liedtke. On $\mu$-kernel construction. In *Proc. 15th SOSP*, December 1995.
19. Y. Moy. Union and cast in deductive verification. In *Proc. C/C++ Verification Workshop*, Technical Report ICIS-R07015. Radboud University Nijmegen, 2007.
20. O. Mürk, D. Larsson, and R. Hähnle. KeY-C: A tool for verification of C programs. In F. Pfenning, editor, *Proc. 21st CADE*, volume 4603 of *LNCS*, pages 385–390. Springer, 2007.
21. T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
22. Open Kernel Labs. OKL4 v2.1. `http://www.ok-labs.com`, 2008.
23. N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.
24. N. Schirmer, M. Hillebrand, D. Leinenbach, E. Alkassar, A. Starostin, and A. Tsyban. Balancing the load — leveraging a semantics stack for systems verification. *JAR, special issue on Operating System Verification*, 42(2–4):389–454, 2009.
25. H. Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Aug. 2008.
26. H. Tuch. Formal verification of C systems code: Structured types, separation logic and theorem proving. *JAR, special issue on Operating System Verification*, 42(2–4):125–187, 2009.
27. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *Proc. 34th POPL*, pages 97–108. ACM, 2007.
28. B. Walker, R. Kemmerer, and G. Popek. Specification and verification of the UCLA Unix security kernel. *CACM*, 23(2):118–131, 1980.