Refinement in the formal verification of the seL4 microkernel

Gerwin Klein[1,2], Thomas Sewell[1], and Simon Winwood[1,2]

Abstract We present an overview of the different refinement frameworks used in the L4.verified project to formally prove the functional correctness of the seL4 microkernel. The verification is conducted in the interactive theorem prover Isabelle/HOL and proceeds in two large refinement steps: one proof between two monadic, functional specifications in HOL and one proof between such a monadic specification and a C program. To connect these proofs into one overall theorem, we map both refinement statements into a common overall framework.

1 Introduction

seL4, the subject of this verification, is an operating system (OS) microkernel. The OS kernel by definition is the part of the software that runs in the most privileged mode of the hardware. As such it has full privileges to access and change all parts of the system. Therefore, any defect in the OS kernel is potentially fatal to the operation of the whole system, not just to isolated parts of it. One approach to reduce the risk of such bugs is the microkernel approach: to reduce the privileged kernel code to an absolute minimum. The remaining code base — 8,700 lines of C and 600 lines of assembly in the case of seL4 — is small enough to be amenable to formal verification on the implementation level. The L4.verified project has produced such an implementation proof for the C code of seL4. The overall proof comes to about 200,000 lines of proof script and roughly 10,000 intermediate lemmas.

The proof assumes correctness of compiler, assembly code and hardware. It also assumes correct use of low-level hardware caches (memory caches and translation-look-aside buffer) and correctness of the boot code (about 1,200 lines of the 8,700).

[1]NICTA, Sydney, Australia ·
[2]School of Computer Science and Engineering, UNSW, Sydney, Australia
{gerwin.klein|thomas.sewell|simon.winwood}@nicta.com.au

It formally derives everything else. The verified version of the seL4 kernel runs on the ARMv6 architecture and the Freescale i.MX31 platform.

This article gives an overview of the main proof technique and the proof framework that was used in this verification project: refinement.

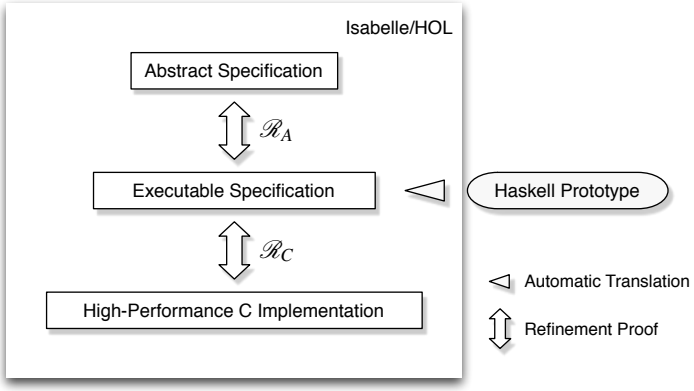

Fig. 1 Refinement steps in L4.verified.

The proof is not done in a refinement calculus that transforms the program in many small steps, but proceeds in two large refinement steps $\mathscr{R}_A$ and $\mathscr{R}_C$ instead. The three main specification artefacts in the proof are shown in Fig. 1. The top level specification of kernel behaviour is an abstract, operational model in higher-order logic. We call it $\mathscr{A}$ in the following. The intermediate specification $\mathscr{E}$ is an executable, detailed model of kernel behaviour that has been translated from a working prototype written in Haskell into Isabelle/HOL. The bottom layer $\mathscr{C}$ is the C program seL4, automatically parsed into Isabelle/HOL.

On the surface, these two large refinement proofs use different formalisms and connect different kinds of specification artefacts. Technical details on these two proofs have appeared elsewhere [4, 23, 17]. This article recalls some of these details and shows how they are put together into a common, general refinement framework that allows us to connect the results and extract the main overall theorem: the C code of seL4 correctly implements its abstract specification.

The next section shows the overall data refinement framework. Sect. 3 gives some example code on the monadic and C level. Sect. 4 summarises the refinement proof $\mathscr{R}_A$ and shows how it is mapped into the framework. Sect. 5 does the same for the C implementation proof $\mathscr{R}_C$.

2 Data Refinement

The ultimate objective of our effort is to prove refinement between an abstract and a concrete process. Following de Roever and Engelhardt [6], we define a process as a triple containing an initialisation function, which creates the process state with reference to some external state, a step function which reacts to an event, transforming the state, and a finalisation function which reconstructs the external state.

record *process* = Init :: *'external* ⇒ *'state* set
Step :: *'event* ⇒ (*'state* × *'state*) set
Fin :: *'state* ⇒ *'external*

The idea is that the external state is the one observable on the outside, about which one may formulate Hoare logic properties. A process may also contain hidden state to implement its data structures. In the simple case, the full state space of a component is just a pair of external and hidden states and the projection function Fin is just the canoncial projection from pairs. With more complex processes, the projection function that extracts the observable state may become more complex as well.

The execution of a process may be non-deterministic, starting from a initial external state, resulting via a sequence of inputs in a *set* of external states:

steps δ *s events* $\equiv$ foldl (λ*states event*. (δ *event*) `` *states*) *s events*
execution *A s events* $\equiv$ (Fin *A*) ` (steps (Step *A*) (Init *A s*) *events*)

where *R `` S* and *f ` R* are the images of the set *S* under the relation *R* and the function *f* respectively.

Process *A* is refined by *C*, if with the same initial state and input events, execution of *C* yields a subset of the external states yielded by executing *A*:

$A \sqsubseteq C \equiv \forall s\ events.$ execution $C\ s\ events \subseteq$ execution $A\ s\ events$

This is the classic notion of refinement as reducing non-determinism. Note that it also includes data refinement: *A* and *C* may work on different internal state spaces; they merely both need to project to the same external state space.

A well-known property of refinement is that it is equivalent with the preservation of Hoare logic properties.

Lemma 1. $A \sqsubseteq C$ *iff* $\forall P\ Q.\ A \vdash \{\!|P|\!\}\ events\ \{\!|Q|\!\} \longrightarrow C \vdash \{\!|P|\!\}\ events\ \{\!|Q|\!\}$.

where $A \vdash \{\!|P|\!\}\ events\ \{\!|Q|\!\} \equiv \forall s \in P.$ execution $A\ s\ events \subseteq Q$. The proof is by unfolding of definitions and basic set reasoning.

This means that once refinement is shown, it is enough to prove a Hoare logic property on the abstract level *A* for it to hold on the concrete level *C*. For this to be useful, the external state must be rich enough to represent the properties one is interested in.

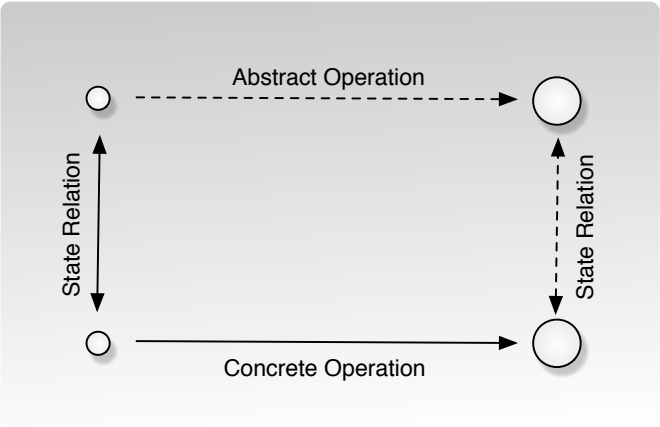

Fig. 2 Forward simulation.

2.1 Forward Simulation

Refinement is commonly proven by establishing forward simulation [6], of which it is a consequence. To demonstrate forward simulation we define a relation, *SR*, between the internal states of the two processes. We must show that the relation is established by Init, is maintained if we advance the systems in parallel, and implies equality of the final external states:

$$\begin{aligned}\textsf{fw-sim}\ SR\ C\ A \equiv\ & (\forall s.\ \textsf{Init}\ C\ s \subseteq SR\ ``\ \textsf{Init}\ A\ s) \\ \wedge\ & (\forall event.\ \textsf{Step}\ C\ event\ O\ SR \subseteq SR\ O\ \textsf{Step}\ A\ event) \\ \wedge\ & (\forall s\ s'.\ (s, s') \in SR \longrightarrow \textsf{Fin}\ C\ s' = \textsf{Fin}\ A\ s)\end{aligned}$$

where $T\ O\ S$ is the composition of relations S and T.

To prove forward simulation, it is often helpful to use additional facts about the execution of abstract or concrete level. If this information is available in the form of an invariant, it may be established separately and can then be easily integrated into the refinement proof. An invariant is any property which is always established by Init and preserved by Step. In the refinement proof, we may then assume it to be true at the commencement of all steps and before finalisation.

$$\begin{aligned}\textsf{invariant}\ I\ M \equiv\ & (\forall s.\ \textsf{Init}\ M\ s \subseteq I) \wedge (\forall event.\ \textsf{Step}\ C\ event\ ``\ I \subseteq I) \\ \textsf{fw-sim}\ SR\ C\ A\ I_c\ I_a \equiv\ & (\forall s.\ \textsf{Init}\ C\ s \subseteq SR\ ``\ \textsf{Init}\ A\ s) \\ \wedge\ & (\forall event.\ \textsf{Step}\ C\ event\ O\ (SR \cap (I_a \times I_c)) \subseteq SR\ O\ \textsf{Step}\ A\ event) \\ \wedge\ & (\forall s\ s'.\ (s, s') \in SR \wedge s \in I_a \wedge s' \in I_c \longrightarrow \textsf{Fin}\ C\ s' = \textsf{Fin}\ A\ s)\end{aligned}$$

The key theorems are, firstly, that forward simulation implies refinement and, secondly, that forward simulation assuming invariants implies forward simulation in general.

Lemma 2. fw-sim $SR\ C\ A \longrightarrow A \sqsubseteq C$

The proof is by unfolding definitions and induction on the event sequence in the refinement statement, followed by relation reasoning to apply forward simulation in the induction step.

Lemma 3. *Forward simulation assuming invariants implies forward simulation if the invariants are established separately.*

$$\text{fw-simI}\ SR\ C\ A\ I_c\ I_a \wedge \text{invariant}\ I_a\ A \wedge \text{invariant}\ I_c\ C \longrightarrow \text{fw-sim}\ SR\ C\ A$$

This lemma is shown by basic relation and set reasoning after unfolding definitions.

2.2 Structure

The three processes we are interested in have a common structure in their Step operations. We model five kinds of events in our processes. The first two are transitions that do not involve the kernel: user thread execution and idle thread execution. We model the execution of user threads with unrestricted nondeterminism, allowing all possible behaviours. We distinguish the idle thread as it may run in the kernel's context and thus must be better behaved. The next two kinds of events model the transition from user mode to kernel mode when exceptions occur: user mode exceptions and idle mode exceptions. The final event type is the one we are interested in: kernel execution. This is the only part of the Step operation that differs between our processes.

Formally, we model this in a function global-automaton that takes the kernel behaviour as a parameter and implements the above transitions generically. The kernel transition is:

global-automaton *kernel-call* KernelTransition ≡
{ ((*s*, KernelMode, Some *e*), (*s'*, *m*, None)) |*s s' e m*. (*s*,*s'*,*m*) ∈ *kernel-call e*}

The paramenter *kernel-call* is a relation between current and final kernel state, and the next mode the machine is switched into (kernel mode, user mode, and idle mode). The state space of the process is a triple of the kernel-observed machine state, including memory and devices, a current mode and a current kernel entry event. The latter is produced by the other transitions in the model. For instance, in idle mode, only an interrupt event can be generated:

global-automaton *kernel-call* IdleEventTransition
{ ((*s*, IdleMode, None), (*s*, KernelMode, Some *Interrupt*)) |*s*. True }

From user mode, any kernel entry event *e* is possible. The transition from user to kernel mode itself does not change the state, the context switch is modelled inside the kernel transition that comes after, because it is modelled differently at each abstraction level. The transition assumes no further conditions and does not depend on the paramenter *kernel-call*.

global-automaton *kernel-call* UserEventTransition ≡
{ ((*s*, UserMode, None), (*s*, KernelMode, Some *e*)) |*s e*. True}

The other transitions are analogous.

The definition of kernel execution may vary between our three processes, but they share a common aspect. Each is implemented through a call to the top-level kernel handler function from which a call graph proceeds in a structured language. Explointing this structure is the key aspect of our approach.

2.3 Correspondence

Forward simulation, like most properties that can be expressed in a commuting diagram, composes sequentially. This composition over successive Step actions is important in the proof that forward simulation implies refinement. Sequential composition is also useful in proving refinement within a single kernel execution step.

The kernel execution bodies are, as discussed above, each written in a language which affords substantial internal structure. To exploit similarities in this structure, we define a new notion which we call correspondence. Correspondence is essentially forward simulation, but defined not on state transformers but on the terms of the languages in which the kernel execution bodies are defined. This leads us to define two different correspondence predicates for $\mathscr{R}_A$ and $\mathscr{R}_C$, which will be discussed in the following sections. It is crucial that these predicates be defined in a manner that allows the correspondence proofs to be composed across the syntactic composition operators of the relevant languages.

3 Example

The seL4 kernel [8] 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 two proof frameworks for refinemnt. Fig. 3 shows the same function in the monadic executable specification and in the C implementation. The first refinement proof relates two monadic specifications; the second refinement proof relates the two layers shown in the figure.

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. 3, moves a capability table entry from *src* to *dest*.

The first 6 lines in Fig. 3 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

```
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. 3 cteMove: executable specification and C implementation

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 functions exhibit this property. It is also true, to a lesser degree, for the refinement between two monadic specifications. 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 that these checks are not required.

4 Monadic Refinement

4.1 Nondeterministic State Monads

The abstract and executable specifications over which $\mathscr{R}_A$ is proved are written in a monadic style inspired by Haskell. The type constructor $('a, 's)$ nd-monad is a nondeterministic state monad representing computations with a state type *'s* and a return value type *'a*. Return values can be injected into the monad using the return :: $'a \Rightarrow ('a, 's)$ nd-monad operation. The composition operator bind :: $('a, 's)$ nd-monad $\Rightarrow ('a \Rightarrow ('b, 's)$ nd-monad$) \Rightarrow ('b, 's)$ nd-monad performs the first operation and makes the return value available to the second operation. These canonical operators form a monad over $('a, 's)$ nd-monad and satisfy the usual monadic laws. More details are given elsewhere [4]. The ubiquitous do ... od syntax seen in Sect. 3 is syntactic sugar for a sequence of operations composed using bind.

The type $('a, 's)$ nd-monad is isomorphic to $'s \Rightarrow ('a \times 's)$ set $\times$ bool. This can be thought of as a nondeterministic state transformer (mapping from states to sets of states) extended with a return value (required to form a monad) and a boolean failure flag. The flag is set by the fail :: $('a, 's)$ nd-monad operation to indicate unrecoverable errors in a manner that is always propagated and not confused by nondeterminism. The destructors mResults and mFailed access, respectively, the set of outcomes and the failure flag of a monadic operation evaluated at a state.

Exception handling is introduced by using a return value in the sum type. An alternative composition operator op $>>=E$ :: $('e + 'a, 's)$ nd-monad $\Rightarrow$ $('a \Rightarrow ('e + 'b, 's)$ nd-monad$) \Rightarrow ('e + 'b, 's)$ nd-monad inspects the return value, executing the subsequent operation for normal (right) return values and skipping it for exceptional (left) ones. There is an alternative return operator *returnOk* and these form an alternative monad. Exceptions are thrown with *throwError* and caught with catch.

We define a Hoare triple denoted $\{\!|P|\!\}\ a\ \{\!|R|\!\}$ on a monadic operator a, precondition P and postcondition Q. We have a verification condition generator (VCG) for such hoare triples, which are used extensively both to establish invariants and to make use of them in correspondence proofs.

4.2 Correspondence

The components of our monadic specifications are similar to the nondeterministic state transformers on which forward simulation is defined. To extend to a correspondence framework we must determine how to handle the return values and failure flags. This is accomplished by the corres predicate. It captures forward simulation between a component monadic computation C, and its abstract counterpart A, with SR instantiated to our standard state relation state-relation. It takes three additional parameters: R is a predicate which will relate abstract and concrete return values, and the preconditions P and P' restrict the input states, allowing use of information such as global invariants:

$$\begin{aligned}&\text{corres } R\ P\ P'\ A\ C \equiv \forall (s, s') \in \text{state-relation}.\ P\ s \wedge P'\ s' \longrightarrow \\ &\quad (\forall (r', t') \in \text{mResults } (C\ s').\ \exists (r, t) \in \text{mResults } (A\ s).\ (t, t') \in \text{state-relation} \wedge R\ r\ r') \\ &\quad \wedge (\neg \text{mFailed } (C\ s'))\end{aligned}$$

Note that the outcome of the monadic computation is a pair of result and failure flag. The last conjunct of the corres statement mandates non-failure for C.

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

CORRES-SPLIT:

$$\frac{\forall r\ r'.\ R'\ r\ r' \longrightarrow \text{corres } R\ (S\ r)\ (S'\ r')\ (B\ r)\ (D\ r') \qquad \{\!|Q|\!\}\ A\ \{\!|S|\!\} \qquad \{\!|Q'|\!\}\ C\ \{\!|S'|\!\}}{\text{corres } R\ (P \text{ and } Q)\ (P' \text{ and } Q')\ (A >>= B)\ (C >>= D)}\ \text{corres } R'\ P\ P'\ A\ C$$

This splitting rule decomposes the problem into four subproblems. The first two are corres predicates relating the subcomputations. Two Hoare triples are also

required. This is because the input states of the subcomputations appearing in the second subproblem are intermediate states, not input states, of the original problem. Any preconditions assumed in solving the second subproblem must be shown to hold at the intermediate states by proving a Hoare triple over the partial computation. Use of Hoare triples to demonstrate intermediate conditions is both a strength and a weakness of this approach. In some cases the result is repetition of existing invariant proofs. However, in the majority of cases this approach makes the flexibility and automation of the VCG available in demonstrating preconditions that are useful as assumptions in proofs of the corres predicate.

The decision to mandate non-failure for concrete elements and not abstract ones is pragmatic. Proving non-failure on either system could be done independently; however, the preconditions needed are usually the same as in corres proofs and it is convenient to solve two problems simultaneously. Unfortunately we cannot so easily prove abstract non-failure. Because the concrete specification may be more deterministic than the abstract one, there is no guarantee we will examine all possible failure paths. In particular, if a conjunct mandating abstract non-failure were added to the definition of corres the splitting rule above would not be provable.

Similar splitting rules exist for other common monadic constructs including bindE, catch and conditional expressions. There are terminating rules for the elementary monadic functions, for example:

CORRES-RETURN:

$$\frac{R\ a\ b}{\mathsf{corres}\ R\ \top\ \top\ (\mathsf{return}\ a)\ (\mathsf{return}\ b)}$$

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

CORRES-PRECOND-WEAKEN:

$$\frac{\mathsf{corres}\ R\ Q\ Q'\ A\ C \qquad \forall s.\ P\ s \longrightarrow Q\ s \qquad \forall s.\ P'\ s \longrightarrow Q'\ s}{\mathsf{corres}\ R\ P\ P'\ A\ C}$$

Proofs of the corres property take a common form: first the definitions of the terms under analysis are unfolded and the CORRES-PRECOND-WEAKEN rule is applied. As with the VCG, this allows the syntactic construction of a precondition to suit the proof. The various splitting rules are used to decompose the problem; in some cases with carefully chosen return value relations. Existing results are then used to solve the component corres problems. Some of these existing 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, as in Dijkstra's postcondition propagation [7], the precondition used must be proved to be a consequence of the one that was originally assumed.

4.3 Mapping to processes

To prove $\mathscr{R}_A$ we must connect the corres framework described above to the forward simulation property we wish to establish. The Step actions of the processes we are interested in are equal for all events other than kernel executions, and simulation is trivial to prove for equal operations. In the abstract process $\mathscr{A}$, kernel execution is defined in the monadic function call-kernel. The semantics of the whole abstract process $\mathscr{A}$ are then derived by using call-kernel in the call to global-automaton. The context switch is modelled by explicitly changing all user accessible parts, for instance the registers of the current thread, fully nondeterministically. The semantics of the intermediate process for the executable specification $\mathscr{E}$ are derived similarly from a monadic operation callKernel. These two top-level operators satisfy a correspondence theorem KERNEL-CORRES:

$\forall$ *event*. corres (λ *rv rv'*. True) *invs invs'* (call-kernel *event*) (callKernel *event*)

The required forward simulation property for kernel execution (assuming the system invariants) is implied by this correspondence rule. Invariant preservation for the system invariants follows similarly from Hoare triples proved over the top-level monadic operations:

$\forall$ *event*. $\{\!|$*invs*$|\!\}$ call-kernel *event* $\{\!|\lambda$*-. invs*$|\!\}$
$\forall$ *event*. $\{\!|$*invs'*$|\!\}$ callKernel *event* $\{\!|\lambda$*-. invs'*$|\!\}$

From these facts we may thus conclude that $\mathscr{R}_A$ holds:

Theorem 1. *The executable specification refines the abstract one.*

$$\mathscr{A} \sqsubseteq \mathscr{E}$$

5 C Refinement

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 [15]. 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. 4 gives an overview of the components involved in importing the kernel into Isabelle/HOL. The right-hand side shows our instantiation of SIMPL [18], a generic, imperative language inside Isabelle. 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. 4 shows this process: the parser takes a C program and produces a C-SIMPL program.

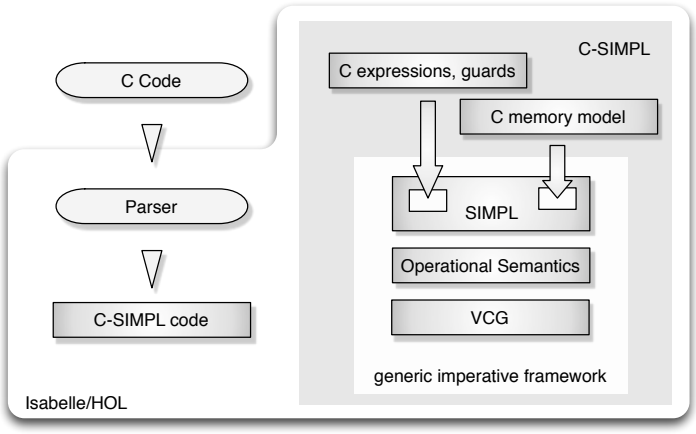

Fig. 4 C language framework.

SIMPL provides a data type and semantics for statement forms; expressions are shallowly embedded. Along with the usual constructors for conditional statements and iteration, SIMPL includes statements of the form Guard *F P c* which raises the fault *F* if the condition *P* is false and executes *c* otherwise.

Program states in SIMPL are represented by Isabelle records containing 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 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 Γ maps function names to function bodies. These states include both the program state and control flow information, including that for abruptly terminating THROW statements 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 *UNIV*, the universal set, then all Guard statements are effectively ignored. Both A and F may be omitted if empty.

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 generally use a more abstract interface to this heap: we use additional typing information to lift the heap into functions from typed pointers to Isabelle terms; see Tuch *et al* [21, 20] for more detail.

The C parser takes C source files and generates the corresponding C-SIMPL terms, along with Hoare-triples describing the set of variables mutated by the functions. Although our C subset does not include union types, we have a tool which generates

data types and manipulation functions which implement tagged unions via C structures and casts [3]. The tool also generates proofs of Hoare-triples describing the operations.

5.1 Refinement Calculus for C

Refinement phase $\mathscr{R}_C$ involves proving 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.

As with $\mathscr{R}_A$, we introduce a new correspondence notion that implies forward simulation. We again aim to divide the proof along the syntactic structure of both programs as far as possible, and then prove the resulting subgoals semantically.

In the following, we first give our definition of correspondence, followed by a discussion of the use of the VCG. We then describe techniques for reusing proofs from $\mathscr{R}_A$ 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.

5.2 The correspondence statement

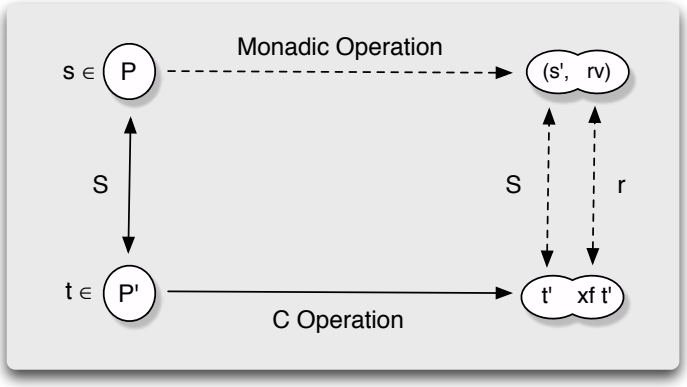

Fig. 5 Correspondence.

As with the correspondence statement for $\mathscr{R}_A$, we deal with state preconditions and return values by including guards on the states and a return value relation in the $\mathscr{R}_C$ correspondence statement. In addition, we include an extra parameter used for

dealing with early returns and breaks from loops, namely a list of statements called a *handler stack*.

We thus extend the semantics to lists of statements, writing $\Gamma \Vdash \langle c{\cdot}hs, s\rangle \Rightarrow x'$. 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. 5 and defined below

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

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 normal (non-abrupt) state $\mathsf{Normal}\ t'_N$
3. states *s'* and t'_N are related by the state relation $\mathscr{S}$; 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. 4.2, the proof of $\mathscr{R}_A$ entails that the executable specification does not fail. Thus, in the definition of ccorres, we may assume $\neg\, \mathsf{mFailed}\ (a\ s)$. In practice, this means assertions and other conditions for (non-)failure in the executable specification become known facts in the proof. Of course, these facts are only free because we have already proven them in $\mathscr{R}_A$.

5.3 Proving correspondence via the VCG

Data refinement predicates can, in general [6], be rephrased and solved as Hoare triples. We do this in our framework by using the VCG after applying the following rule:

$$\frac{\begin{array}{c}\forall s.\ \Gamma \vdash \{t \mid s \in P \wedge t \in P' \wedge (s,t) \in \mathscr{S}\} \\ c \\ \{t' \mid \exists (rv, s') \in \mathsf{mResults}\ (a\ s).\ (s', t') \in \mathscr{S} \wedge r\ rv\ (xf\ t')\}\end{array}}{\mathsf{ccorres}\ r\ xf\ P\ P'\ hs\ a\ c}$$

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.

5.4 Splitting

As with $\mathscr{R}_A$, we prove correspondence by splitting the proof into corresponding program lines. Splitting allows us to take advantage of 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:

$$\frac{\begin{array}{cc}\multicolumn{2}{c}{\mathsf{ccorres}\ r'\ xf'\ P\ P'\ hs\ a\ c} \\ \forall v.\ d'\ v \sim d[v/xf'] & \forall rv\ rv'.\ r'\ rv\ rv' \longrightarrow \mathsf{ccorres}\ r\ xf\ (Q\ rv)\ (Q'\ rv\ rv')\ hs\ (b\ rv)\ (d'\ rv') \\ \{\!|R|\!\}\ a\ \{\!|Q|\!\} & \Gamma \vdash_{/\mathscr{U}} R'\ c\ \{s \mid \forall rv.\ r'\ rv\ (xf'\ s) \longrightarrow s \in Q'\ rv\ (xf'\ s)\}\end{array}}{\mathsf{ccorres}\ r\ xf\ (P \cap R)\ (P' \cap R')\ hs\ (a >>= b)\ (c; d)}$$

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 the universal set 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.

5.5 Mapping to processes

We map the C kernel into a process by lifting the operational semantics of the kernel C code into a non-deterministic monad:

$$\text{exec-C}\ \Gamma\ c \equiv \lambda s.\ (\{()\} \times \{s' \mid \Gamma \vdash \langle c, \text{Normal}\ s\rangle \Rightarrow \text{Normal}\ s'\},\ \text{False})$$

that is, for a given statement c we construct a function from an initial state s into the set of states resulting from evaluating c at s. We define the return value of this execution as the unit. We set the failure flag to False and require a successful Normal result from C.

We then construct a function callKernel-C, parametrised by the input event, which simulates the hardware exception dispatch mechanism. The function examines the argument and dispatches the event to the corresponding kernel entry point. Finally, we form the process *ADT-C* by instantiating the global automaton with this step function.

We again establish a correspondence result between the kernel entry points, this time between callKernel in $\mathscr{E}$ and callKernel-C in C. This time, we did not need to prove additional invariants about the concrete level (the C program). The framework presented above enabled us to shift all such reasoning the level of the executable specification $\mathscr{E}$.

Theorem 2. *The translated C code refines its executable specification.*

$$\mathscr{E} \sqsubseteq \mathscr{C}$$

6 Main theorem

Putting the two theorems from the previous sections together, we arrive via transitivity of refinement at the main functional correctness theorem.

Theorem 3. $\mathscr{A} \sqsubseteq \mathscr{C}$

7 Related Work

We briefly summarise related work on OS verification; a comprehensive overview is provided by Klein [16].

Early work on OS verification includes PSOS [9] and UCLA Secure Unix [22]. Later, KIT [2] describes verification of process isolation properties down to object code level, but for an idealised kernel much simpler than modern microkernels.

The VFiasco project [13] and later the Robin project [19] attempted to verify C++ kernel implementations. They created a precise model of a large, relevant part of C++, but did not verify substantial parts of the kernel.

Heitmeyer et al. [12] report on the verification and Common Criteria certification of a "software-based embedded device" featuring a small (3,000 LOC) separation kernel. Similarly, Green Hills' Integrity kernel [11] recently underwent formal verification during a Common Criteria EAL6+ certification [10]. The Separation Kernel Protection Profile [14] of Common Criteria demands data separation only rather than functional correctness.

A closely related project is Verisoft [1], which is attempting to verify not only the OS, but a whole software stack from verified hardware up to verified application programs. This includes a formally verified, non-optimising compiler for a Pascal-like implementation language. While Verisoft accepts a simplified (but verified) hardware platform and two orders of magnitude slow-down for the simplified VAMOS kernel, we deal with real C and standard tool chains on ARMv6, and have aimed for a commercially deployable, realistic microkernel. A successor project, Verisoft XT, is aiming to verify the functional correctness of the Microsoft Hypervisor, which contains concurrency and is substantially larger than seL4. While initial progress has been made on this verification [5], it is unclear at this stage if the goal will be reached.

8 Conclusion

We have presented the different refinement techniques used in the verification of the seL4 microkernel. We have given an overview of the overall unifying framework, of the refinement calculus used for stateful, monadic specification, of the refinement calculus for imperative programs, and we have shown how these are put together into the final theorem.

The two frameworks presented here have withstood the test of large-scale application to high-performance C code in the Isabelle/HOL verification of the seL4 microkernel. Proving functional correctness for real-world application down to the implementation level is possible and feasible.

Acknowledgements

We thank the other current and former members of the L4.verified and seL4 teams: David Cock, Tim Bourke, June Andronick, Michael Norrish, Jia Meng, Catherine Menon, Jeremy Dawson, Harvey Tuch, Rafal Kolanski, David Tsai, Andrew Boyton,

Kai Engelhardt, Kevin Elphinstone, Philip Derrin and Dhammika Elkaduwe for their contributions to this verification.

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.

References

1. Eyad Alkassar, Mark Hillebrand, Dirk Leinenbach, Norbert Schirmer, Artem Starostin, and Alexandra Tsyban. Balancing the load — leveraging a semantics stack for systems verification. *JAR*, 42(2–4), 2009.
2. William R. Bevier. Kit: A study in operating system verification. *IEEE Transactions on Software Engineering*, 15(11):1382–1396, 1989.
3. David Cock. Bitfields and tagged unions in C: Verification through automatic generation. In Bernhard Beckert and Gerwin Klein, editors, *VERIFY'08*, volume 372 of *CEUR Workshop Proceedings*, pages 44–55, Aug 2008.
4. David Cock, Gerwin Klein, and Thomas Sewell. Secure microkernels, state monads and scalable refinement. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *21st TPHOLs*, volume 5170 of *LNCS*, pages 167–182. Springer, Aug 2008.
5. Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42, Munich, Germany, 2009. Springer.
6. Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Number 47 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998.
7. Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.
8. Kevin Elphinstone, Gerwin Klein, Philip Derrin, Timothy Roscoe, and Gernot Heiser. Towards a practical, verified kernel. In *Proc. 11th Workshop on Hot Topics in Operating Systems*, pages 117–122, San Diego, CA, USA, May 2007.
9. Richard J. Feiertag and Peter 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.
10. Green Hills Software, Inc. INTEGRITY-178B separation kernel security target version 1.0. `http://www.niap-ccevs.org/cc-scheme/st/st_vid10119-st.pdf`, 2008.
11. Greenhills Software, Inc. Integrity real-time operating system. `http://www.ghs.com/products/rtos/integrity.html`, 2008.
12. Constance L. Heitmeyer, Myla Archer, Elizabeth I. Leonard, and John McLean. Formal specification and verification of data separation in a separation kernel for an embedded system. In *CCS '06: Proc. 13th Conf. on Computer and Communications Security*, pages 346–355. ACM, 2006.
13. Michael Hohmuth and Hendrik Tews. The VFiasco approach for a verified operating system. In *2nd PLOS*, Jul 2005.
14. Information Assurance Directorate. *U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness*, Jun 2007. Version 1.03. `http://www.niap-ccevs.org/cc-scheme/pp/pp.cfm/id/pp_skpp_hr_v1.03/`.
15. ISO/IEC. Programming languages — C. Technical Report 9899:TC2, ISO/IEC JTC1/SC22/WG14, May 2005.
16. Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, Feb 2009.

17. Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proc. 22th SOSP*, pages 207–220, Big Sky, MT, USA, October 2009. ACM.
18. Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL.* PhD thesis, Technische Universität München, 2006.
19. Hendrik Tews, Tjark Weber, and Marcus Völp. A formal model of memory peculiarities for the verification of low-level operating-system code. In Ralf Huuck, Gerwin Klein, and Bastian Schlich, editors, *Proc. 3rd Int. WS on Systems Software Verification (SSV'08)*, volume 217 of *ENTCS*, pages 79–96. Elsevier, Feb 2008.
20. Harvey 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.
21. Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 97–108, Nice, France, 2007. ACM.
22. Bruce Walker, Richard Kemmerer, and Gerald Popek. Specification and verification of the UCLA Unix security kernel. *Commun. ACM*, 23(2):118–131, 1980.
23. Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock, and Michael Norrish. Mind the gap: A verification framework for low-level C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proc. TPHOls'09*, volume 5674 of *LNCS*, pages 500–515, Munich, Germany, August 2009. Springer.