Under consideration for publication in Formal Aspects of Computing

Concerned with the Unprivileged: User Programs in Kernel Refinement

Matthias Daum,[†,‡] Nelson Billing[†,1] and Gerwin Klein[†,‡]

[†]Neville Research Laboratory, NICTA, Kensington, Sydney, Australia.
[‡]The University of New South Wales, Kensington, Sydney, Australia.

Abstract. It is a great verification challenge to prove properties of complete computer systems on the source code level. The L4.verified project achieved a major step towards this goal by mechanising a proof of functional correctness of the seL4 kernel. They expressed correctness in terms of data refinement with a coarse-grained specification of the kernel's execution environment.

In this paper, we strengthen the original correctness theorem in two ways. First, we convert the previous abstraction relations into projection functions from concrete to abstract states. Second, we revisit the specification of the kernel's execution environment: we introduce the notion of virtual memory based on the kernel data structures, we distinguish individual user programs that run on top of the kernel and we restrict the memory access of each of these programs to its virtual memory. Through our work, properties like the separation of user programs gain meaning. This paves the way for proving security properties like non-interference of user programs. Furthermore, our extension of L4.verified's proof facilitates the verification of properties about complete software systems based on the seL4 kernel.

Besides the seL4-specific results, we report on our work from an engineering perspective to exemplify general challenges that similar projects are likely to encounter. Moreover, we point out the advantages of using projection functions in L4.verified's verification approach and for stepwise refinement in general.

Keywords: Trustworthy Systems; Refinement Proof; Microkernel Correctness; Isabelle/HOL

1. Introduction

Computer systems are omnipresent and as they become increasingly complex, traditional means of quality assurance like testing are pushed to their limits. As an alternative, verification provides the means to establish rigorous system guarantees. Yet it remains a grand challenge to formally prove properties about complete computer systems on the source code level. Fortunately, the typical architecture of computer systems facilitates the division of the overall verification task into more managable components. Most systems

Correspondence and offprint requests to: Gerwin Klein, Locked Bag 6016, The University of New South Wales, Sydney NSW 1466, Australia. Phone: +61 (02) 8306 0578. E-mail: Gerwin.Klein@nicta.com.au
[1] This author works for Microsoft, now.

feature an *operating system kernel*, which is responsible for resource management and the separation of different software components. The kernel runs in a *privileged* mode of the hardware, which allows the kernel to configure a hardware mechanism for the separation. All other software components run in an *unprivileged* mode, where separation is enforced but cannot be reconfigured. With this architecture, the correctness of the computer system hinges on kernel correctness because (only) the kernel has complete control over the system and thus a kernel bug may affect the entire system.

To this end, the L4.verified project [KEH+09] produced a formal, machine-checked proof of functional correctness for the seL4 kernel. The kernel comprises about 8,700 lines of C code and its correctness proof required an effort of about 11 person-years. This achievement is a major step towards verified computer systems. Though orders of mangitude smaller than commodity kernels, seL4 still features the essential mechanisms for the implementation of a general-purpose operating system. The straightforward direction for follow-up research is thus constructing provably secure systems based on seL4. Andronick *et al.* [AGE10] propose an overall verification approach for such a system, pointing out the natural fit of a componentised software architecture that separates trusted (and verified) components from untrusted ones. This approach hinges on proven non-interference between these components [MMB+12, MMB+13].

We contribute to this vision of formally proven security by strengthening the correctness statement of seL4. Our contribution is threefold:

1. We exploit the full potential of the auxiliary lemmas that support the original statement of functional correctness. This result alleviates all further proofs building on seL4's correctness.
2. We amend the modelled execution environment of the kernel with the hardware mechanism for the separation of individual programs running in unprivileged mode. We call such an individual program a *process* (we further refine this notion in Section 2.1). Separating processes is an important criterion for kernel correctness, which is related to the formal notion of non-interference. In distinguishing individual processes, we establish an essential prerequisite for the non-interference proof.
3. We introduce a customisable model of processes. Thereby, we facilitate proofs about a componentised system and aim to reduce the effort for proving whole-system guarantees.

In detail, our first result improves the original proof architecture. The L4.verified project phrased functional correctness as a data refinement from an abstract kernel specification down to a representation of the C code. In other words, the correctness statement relates state machines that differ in the representation of the kernel data structures. To compare the state representations at different levels of abstraction, relevant information is projected out into a common representation, the so-called *observable state*. We extend this observable state to comprise the whole abstract kernel state such that the projection from an abstract kernel state into the observable state becomes the identity function. L4.verified prepared the grounds for this result through its notion of *correspondence* and a plethora of auxiliary lemmas founded on this notion. We describe L4.verified's proof techniques in Section 2.2 and return to our extension of the observable state in Section 3.

Our second and main result focusses on distinguishing individual processes. Data refinement is usually concerned with the behaviour of a programming module in the context of a larger program. For a kernel, this context mainly comprises the processes. The original refinement centered around the manipulation of kernel data structures at different layers of abstraction. The kernel's execution environment, in contrast, was initially of less concern and was consequently overapproximated. L4.verified collectively regarded all software running in unprivileged mode as a single entity, referring to it as *user code*, in contrast to the kernel. They overapproximated its behaviour by permitting arbitrary changes of the current register file and of all memory that the kernel has assigned to user code. In practice however, individual processes operate on *virtual memory*, which is typically a small fraction of the collective user-accessible memory.

In this paper, we define a notion of virtual memory for the seL4 kernel and distinguish individual processes. Conceptually, virtual memory is a partial map from so-called virtual memory addresses to pairs of physical memory addresses and corresponding access permissions. It is this additional layer of indirection that allows for the hardware-supported separation of several processes running on top of the same kernel. In Section 4.1, we define an abstract memory-management unit (MMU), i. e. functions that project virtual memory out of the kernel data structures. These data structures closely model the memory that the actual hardware MMU accesses when translating a virtual into a physical address. Based on the abstract MMU, we then specify the transitions of the current process in Section 4.2.

Note that we have not only refined the specification of user transitions but actually strengthened the correctness statement: where previously the set of user-accessible memory locations was determined based

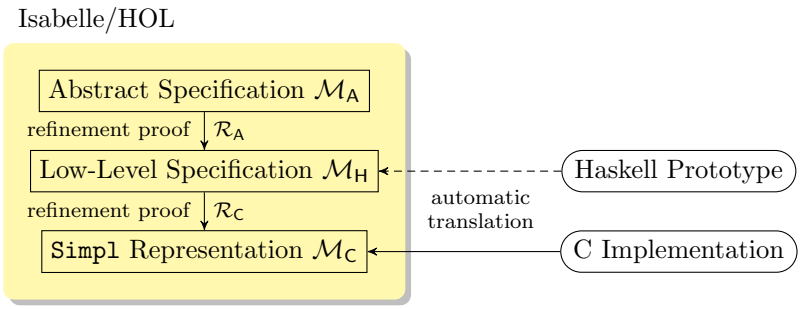

Fig. 1. Refinement steps in the L4.verified project

on kernel-internal accounting, we now base this decision on virtual memory, which is closer to the actual hardware. As a consequence, the original data refinement proof breaks. We have reestablished the proof by showing that any process-accessible virtual memory maps to memory regions that the kernel has assigned to user code. This proof is the subject of Sections 4.3–4.5.

Our third result is small but nonetheless valuable. We have added a parameter *user-op* to user transitions (cf. Section 4.1). The parameter is a function taking information about the currently running process—including its virtual memory—and computing the set of possible next process states. The purpose of this parameter is to customise user transitions separately from the actual kernel refinement. For kernel refinement, we all-quantify over this parameter because the kernel should function independently from the nature of the individual processes. Subsequently, this parameter can be instantiated with refined models of user transitions: a coarse-grained overapproximation summarising all possible behaviour or a precise, low-level model capturing the exact hardware instruction set. With this approach, the abstraction layer for each process can be chosen separately. For instance, trusted processes could be modelled precisely while untrusted processes could be specified as behaving largely non-deterministically [Bil12].

Apart from the immediate results for seL4, this paper makes two more general contributions. First, it takes an engineering perspective on operating system verification—thus reporting on technical problems that similar projects in the same application domain will encounter. The most challenging—and most domain-specific—proofs regard global kernel invariants. Specifically, we exemplify some problems we encountered when discussing Lemma 4 as well as in Sections 4.3 and 4.4. Second, our extension of the observable state amends L4.verified's original proof architecture when proving invariants in stepwise refinement. Its advantages are detailed in Section 4.4.

In the next section, we recapitulate the context of our work: Section 2 acquaints us with the seL4 kernel, L4.verified's proof techniques as well as the modelling of the kernel's states and its execution environment.

2. A Primer on L4.verified and seL4

The L4.verified project proved functional correctness for the seL4 kernel by stepwise refinement using the interactive theorem prover Isabelle/HOL [KEH+09]. Fig. 1 depicts this overall approach, which we continue to follow in our work. Most notably, the box labelled "Isabelle/HOL" at the left comprises all formal artefacts: the different layers of abstraction and the refinement steps relating them. The *abstract specification* at the top is an operational model, capturing the system behaviour including the kernel interface without prescribing a particular implementation. The *low-level specification* below is auto-generated from a Haskell prototype. This prototype features the same data structure and implementation details as the high-performance C implementation of seL4. Cock *et al.* [CKS08] describe the refinement proof $\mathcal{R}_\mathsf{A}$ between the upper two layers. Finally, the bottom layer represents the C implementation in the language `Simpl` [Sch05, Sch06]. Note that only the translation from C to `Simpl` is critical for correctness while the translation from Haskell is not. The refinement proof $\mathcal{R}_\mathsf{C}$ down to the `Simpl` representation is described by Winwood *et al.* [WKS+09].

In the remainder of this section, we first sketch the high-level design principles of seL4, then we present the underlying refinement calculus and finally, we discuss the models of the kernel's state space and its execution environment.

2.1. The seL4 Kernel

The seL4 kernel belongs to the *L4 family of microkernels*. This family shares a design principle of minimality [Lie95]: the kernel is minimised to provide only the essential mechanisms for implementing a general-purpose operating system as unprivileged processes. This design allows different operating systems, implementing various policies, to co-exist at runtime on the same microkernel.

The security model of seL4 is based on *capabilities*. A capability is a token of *authority* combining an object reference with associated *access rights*. Capabilities can be communicated but cannot be forged. In seL4, authority over all kernel objects is conferred by capabilities. Processes obtain kernel services by invoking capabilities and the kernel accounts, by capabilities to memory regions, for all memory they directly or indirectly use. A process creates objects by calling the kernel with a capability to a memory region that the objects should live in. If space permits, the kernel creates the objects together with capabilities to them and records the new capabilities as descendants of the memory-region capability. To reclaim a region of memory, a process calls the kernel with the capability to that region, requesting it to destroy all objects in that region.

So far, we have defined a process as "a program running in unprivileged mode". In fact, this notion is a combination of different seL4 concepts. More precisely, a process is characterised by three entities: at least one *thread*, which is an abstraction for (execution time on) a processor, a *virtual address space*, which is a set of virtual addresses mappable to physical addresses, and a *capability space*, which is a set of capability entries mappable to pairs of object references and access rights. Thus, the information about a process is spread over several kernel objects: a *thread object* carries the current content of the processor registers, a *page directory* characterises a virtual address space and a *capability node* describes a capability space. While all information about a thread is contained in a single object, page directories and capability nodes can be the root of an object tree. We expand on virtual address spaces in Section 4.1. For the purpose of this paper, we regard capability nodes as simple tables.

The seL4 kernel supports different hardware architectures. When we describe architecture details in this paper, we generally refer to the ARMv6 architecture.

2.2. Data Refinement in L4.verified

The L4.verified proof used a formalisation of data refinement [CKS08] that largely follows de Roever and Engelhardt [dRE98]. In essence, refinement is a binary relation $\sqsupseteq$ between state machines. The underlying idea is a design process creating correct software by construction. The process starts with a formal specification M_a and gradually transforms it into an executable M_c by adding more and more implementation details. Data refinement, in particular, is concerned with increasing the level of detail in data structures. Intuitively, the refinement relation $M_c \sqsupseteq M_a$ states that M_c is a correct implementation of M_a. In other words, the behaviour of M_a should, in some sense, contain the behaviour of M_c. We require auxiliary notions to turn this intuition into a formal definition.

In a relational semantics, the behaviour of a deterministic program is completely described by pairs of initial and final states. Similarly, the behaviour of a state machine with inputs can be determined by triples comprising an initial state, an input sequence, and a final state. For non-deterministic behaviour, we can simply use a set of final states. These observations lead us to the formal definition of refinement, assuming a function execution that computes the set of final states for a state machine, an initial state and an input sequence:

Definition 1 (Refinement). A state machine M_c refines (or implements) another state machine M_a, if for any initial state σ and finite input sequence *seq*, the set of final states of M_a is a superset of those of M_c:

$$M_c \sqsupseteq M_a \equiv \forall \sigma\ seq.\ \mathsf{execution}\ M_c\ \sigma\ seq \subseteq \mathsf{execution}\ M_a\ \sigma\ seq$$

Note that this definition implicitly assumes equal state spaces, while it is the very purpose of data refinement to relate state machines featuring different state spaces. Data refinement solves this problem by requiring conversions into a common representation, the so-called *observable state*.

Typically, the overall design process involves multiple transformation steps, where each step focusses on only one programming module in the context of a larger, surrounding computation. The state is partitioned into data belonging to that module and data belonging to the surrounding computation. Only the representation of data belonging to the module changes in this transformation step. Module operations work on

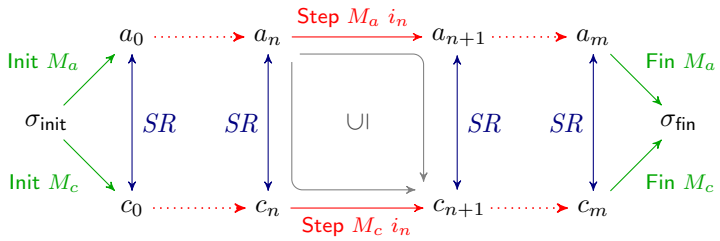

Fig. 2. Forward simulation and refinement

module data, and the surrounding computation works on module-external data or accesses module data via a well-defined interface of module operations. In this setting, the name *observable state* becomes clearer: it should contain all information that the surrounding computation can observe—directly or through the interface operations. Note that, in practice, an implementation might retain some information, for instance left over from a deleted object, simply for performance reasons. Such left-over information is not observable. To accomodate the observable state, we refine the notion *state machine* in the context of data refinement:

Definition 2 (State Machine). Formally, a state machine M is a triple of functions; we use the projections Init, Step and Fin to access its components. Component Init M converts an observable state into a set of internal states, Step M takes an input event as argument and returns the corresponding binary relation of internal states, and Fin M converts an internal state into an observable state.

Now the definition of the execution function for Def. 1 becomes apparent: The execution function converts the initial observable state σ with Init M into a set of internal states. Then, it reduces the input sequence *seq* by recursively applying the transition function Step M with an input from *seq* to the internal states. And finally, it element-wise converts the final states with Fin M into observable states.

Refinement is commonly proven by forward simulation (also known as *L-simulation*). Fig. 2 illustrates the formal definition below and indicates why refinement is a consequence of forward simulation. In essence, the figure shows two different ways of computing the final observable state σ_{fin} at the right hand side from the initial observable state σ_{init} at the left. At the top, the computation involves states and transitions of an abstract state machine M_a and at the bottom those of a concrete machine M_c. Both should result in the same state σ_{fin}.

Definition 3 (Forward simulation). As depicted in Fig. 2, forward simulation assumes a relation SR between the internal states of two state machines M_a, M_c and comprises three proof obligations:

a) For each observable state σ and each concrete internal state s_c in the set of initial states Init M_c σ, there exists an abstract internal state s_a in the set of initial states Init M_a σ such that the pair (s_a, s_c) is contained in the state relation SR.

b) The state relation SR must be maintained under parallel transitions of both state machines with the same input i; in other words, the composition of state relation and concrete transition relation must be a subset of the composition of the abstract transition relation and the state relation.

c) Two correlated internal states $(s_a, s_c) \in SR$ must convert into the same final, observable state.

Formally:

$$\begin{aligned}\text{fw-sim } M_a\ M_c\ SR \equiv\ & (\forall \sigma.\ \forall s_c \in \text{Init } M_c\ \sigma.\ \exists s_a \in \text{Init } M_a\ \sigma.\ (s_a, s_c) \in SR) \ \wedge \\ & (\forall i.\ SR \circ \text{Step } M_c\ i \subseteq \text{Step } M_a\ i \circ SR) \ \wedge \\ & (\forall s_a\ s_c.\ (s_a, s_c) \in SR \ \longrightarrow\ \text{Fin } M_c\ s_c = \text{Fin } M_a\ s_a)\end{aligned}$$

where $R \circ S$ is the composition of the relations R and S.

Proof obligation b) usually requires the most proof effort. In L4.verified, Step M_a and Step M_c consist of smaller steps expressed in non-deterministic state monads with exceptions and failure. State monads introduce side effects to a purely functional model of computation. The type of state monads is typically parametric over a state space S and a result type R. The simplest case is a deterministic state monad m_{det}, which we can view as a function from a state to a computation result and a new state: $m_{\text{det}} \in S \to (R \times S)$. Non-determinism introduces more complexity: there might be several pairs of a result and a new state or none

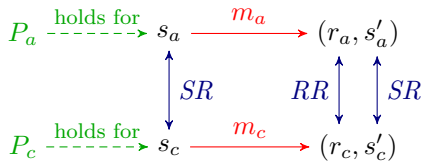

Fig. 3. Correspondence between two state monads

(divergence, i. e. failure). To aid compositionality, Cock at al. [CKS08] represent failure with an additional flag. Their formalisation of monads takes a state $s \in S$ and returns either a) the pair of the empty set and the raised failure flag: $(\emptyset, \mathsf{True})$, or b) a pair of a non-empty set (with pairs of return values and consecutive states) as well as False for non-failure. In other words, they represent non-deterministic state monads as functions of type $S \to (R \times S)\ \mathsf{set} \times \mathsf{bool}$.

The bind operator allows for sequential composition of state monads. More specifically, $\mathsf{bind}\ m\ f$ composes a new state monad from the state monad m and the function f that takes the result from m and returns another monad. To aid readability, we write **do** $r \leftarrow m;\ f\ r$ **od**. By repeated composition of state monads, a monad m^{big} can be constructed that essentially represents a big step $\mathsf{Step}\ M$. Disregarding the monad's results, m^{big} can easily be converted into the relational form required for $\mathsf{Step}\ M$, i. e. $\mathsf{Step}\ M$ essentially becomes $\{(s, s').\ \exists r.\ (r, s') \in \mathsf{fst}\,(m^{\mathsf{big}}\ s)\}$ (where fst denotes the first component of a pair).

The L4.verified project translated the forward simulation problem to state monads and then broke it down into smaller steps to address scalability concerns (see also Fig. 3 for an illustration):

Definition 4 (Correspondence). Assuming a state relation SR and a return relation RR, we say that the abstract state monad m_a and the concrete state monad m_c correspond under the preconditions P_a and P_c, respectively, if for all states $(s_a, s_c) \in SR$, the preconditions $P_a\ s_a$ and $P_c\ s_c$ imply the following two conditions:

a) if the monad m_c fails with s_c, so does m_a with s_a.
b) for each pair (r_c, s'_c) in the set of return values and consecutive states of $m_c\ s_c$, there is a pair (r_a, s'_a) in the set of return values and consecutive states of $m_a\ s_a$, such that the states and return values are related, i. e. $(s'_a, s'_c) \in SR$ and $(r_a, r_c) \in RR$.

The predicate $\mathsf{corres}\ SR\ RR\ P_a\ P_c\ m_a\ m_c$ captures this notion formally.

In practice, statements of correspondence involve two different kinds of preconditions: *global invariants*, which are maintained by the respective big steps m_a^{big} and m_c^{big}, and *variants*, which reflect assumptions on previous computation along the control flow path. The smaller, monadic steps that constitute a big step, frequently break and later reestablish global invariants, possibly relying on previously established variants.

Conceptually, the overall approach consists of establishing individual corres statements and composing them into larger ones until eventually a statement of correspondence is derived that bears the global invariants I_a and I_c as preconditions and relates two state monads m_a^{big} and m_c^{big} representing the big steps:

$$\mathsf{corres}\ SR\ \{(r_a, r_c).\ \mathsf{True}\}\ I_a\ I_c\ m_a^{\mathsf{big}}\ m_c^{\mathsf{big}}$$

The remaining link to refinement is using this corres statement to establish the second proof obligation of forward simulation. Recall, however, that the above definition of forward simulation does not feature a notion of global invariants as they appear in the above corres statement. In fact, the assumed global invariants I_a and I_c and the state relation SR established in correspondence proofs can be combined to form the stronger state relation $SR' = SR \cap \{(s_a, s_c).\ I_a\ s_a \wedge I_c\ s_c\}$. This leads to the proof obligation that the global invariants are preserved by m_a^{big} and m_c^{big}, respectively.

In L4.verified, Hoare triples were used to express facts about computations of state monads. The Hoare triple $\{\!|\lambda s.\ P\ s|\!\}\ m\ \{\!|\lambda r\ s'.\ Q\ r\ s'|\!\}$ states that if the precondition P holds for an initial state s, for each pair (r, s') in the first component of $m\ s$, the postcondition $Q\ r\ s'$ holds. We use λ-notation to emphasize that the precondition P is a unary predicate about the initial state s, and the postcondition Q is a binary predicate about m's return value r and the final state s'.

Below, we illustrate the general approach using the example of *user transitions*. These transitions capture the behaviour of software running in unprivileged mode. Each layer of abstraction features its own version:

$\mathsf{user\text{-}trans_A}$ and $\mathsf{user\text{-}trans_H}$ describe user behaviour on the two specification layers and $\mathsf{user\text{-}trans_C}$ on the `Simpl` layer. The vast majority of the verification effort in the L4.verified project was split between two kinds of lemmas: correspondence, on the one hand, and invariant preservation, on the other hand. The correspondence lemmas relate monads of adjacent layers; we show the according statement between the specification layers as an example.

Lemma 1. The user transitions at the two specification layers, $\mathsf{user\text{-}trans_A}$ and $\mathsf{user\text{-}trans_H}$, respectively, correspond w. r. t. the state relation $\mathsf{SR_A}$ and equality of return values:

$$\begin{array}{l}\mathsf{corres}\ \mathsf{SR_A}\ \{(r_a, r_c).\ r_a = r_c\} \\ \quad (\lambda s.\ \mathsf{I_A}\ s\ \wedge\ \mathsf{ut\text{-}pre_A}\ s)\ (\lambda s.\ \mathsf{I_H}\ s\ \wedge\ \mathsf{ut\text{-}pre_H}\ s) \\ \quad (\mathsf{user\text{-}trans_A}\ tc)\ (\mathsf{user\text{-}trans_H}\ tc)\end{array}$$

where $\mathsf{I_A}$ and $\mathsf{I_H}$ are the global invariants at the respective layers and $\mathsf{ut\text{-}pre_A}$ and $\mathsf{ut\text{-}pre_H}$ are variants about the scheduler and the current thread, which hold whenever the system performs user transitions.

Moreover, L4.verified showed invariant preservation for user transitions at each abstraction layer. As an example, we present the corresponding lemma for the abstract specification.

Lemma 2. User transitions at the abstract specification layer maintain the invariants:

$$\{\!|\lambda s.\ \mathsf{I_A}\ s\ \wedge\ \mathsf{ut\text{-}pre_A}\ s|\!\}\ \mathsf{user\text{-}trans_A}\ tc\ \{\!|\lambda r\, s.\ \mathsf{I_A}\ s|\!\}$$

Given the highly non-deterministic nature of the original user transitions, the proofs for the above lemmas are simple. We return to user transitions in Section 4. In the two remaining subsections, we introduce the kernel's state space at different levels of abstraction and the model of the kernel's execution environment.

2.3. Modelling Kernel State

Essentially, seL4 is a C program. The code is represented in the language `Simpl`, where a program state is a tuple[2] combining the values of all program variables; C types are mapped to Isabelle/HOL types. To this end, Tuch [TK05, Tuc08] developed a heap model that accurately captures the low-level semantics of C pointer types, including pointer arithmetic, pointers into substructures and unions. In particular, the heap is a state component consisting of two functions, which assign an 8-bit value and a C type, respectively, to each 32-bit address. Furthermore, the partial function $\mathsf{clift}_t\ h\ ptr$ dereferences pointer ptr on heap h for C type t. If dereferencing is not type safe, the function returns the value None. For example, the kernel uses a specific type umt to access user memory. Hence, we can use $\mathsf{clift_{umt}}$ on `Simpl` states to extract the contents of all memory that the kernel has assigned to user code. Notably, this approach allows us to interpret the same heap at the same address with many different types.

In general, the two specification layers $\mathcal{M}_\mathsf{A}$ and $\mathcal{M}_\mathsf{H}$ follow the same state layout and represent the state as a tuple of variable values. Only the representation of the kernel heap changes. In particular, the heap is split into two separate components: on the one hand, there is an abstract kernel heap for kernel objects like page tables or thread control structures; on the other hand, there is the physical memory of the underlying machine. The abstract heap h is a partial function from 32-bit addresses to values of a more abstract, fixed type of kernel objects. A page table on the abstract specification, for instance, is a function from an 8-bit index to page table entries (PTEs). The kernel object representing an empty page table is $\mathsf{PageTable}\ (\lambda i.\ \mathsf{InvalidPTE})$. Although kernel objects generally occupy more than 8 bits in C, the whole object value is associated with the starting address. The contents of the physical memory only matter if assigned to user code. This assignment is marked by special kernel objects ($\mathsf{DataPage}\ size$) in the abstract heap.

2.4. Modelling the Execution Environment of seL4

The seL4 kernel is responsible for managing resources on behalf of user code, which naturally means that user code and kernel share state. Thus, the state machines for data refinement must include state changes

[2] Strictly speaking, program states are Isabelle/HOL *records*—special types that are internally represented by tuples and feature read and update functions for each of their components.

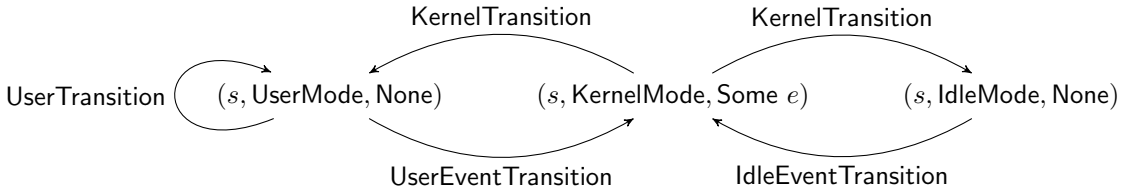

Fig. 4. The system automaton's transitions between states of different machine control modes

from the execution environment of the kernel. Moreover, hardware events like device interrupts can disrupt the user code's normal control flow. To capture the generic data structures and control flow of the kernel's environment, the L4.verified project defined a single *system automaton*, serving as a framework for state machines at all abstraction layers with parameters accounting for differences in abstraction.

Definition 5 (System Automaton). The system automaton is defined by its state space and its transition relation.

The *state space* is a set of triples, parametrized over the underlying kernel state K. The first component s of each triple is a pair (r, k) with the register context r of the currently running thread and the kernel state $k \in K$; The second component represents the machine control mode, one of UserMode, KernelMode or IdleMode. The third component is an optional hardware event to be processed, which can be a device interrupt (Interrupt), system call, or fault. We write None if there is no event and Some e for an event e to be processed.

The *transition relation* takes two parameters: *user-trans* describes a local computation of user code and *kernel-call* describes a computation from entering kernel mode until leaving it again. More specifically, *user-trans* is a function that takes a register context and yields a kernel monad returning a register context. The kernel transition *kernel-call* is a function from an event to a set of triples, each consisting of a starting state, a machine mode, and a final state.

The transition relation distinguishes four kinds of transitions t between the machine control modes (cf. Fig. 4): a) a KernelTransition performs a computation of the kernel and afterwards returns control to a user or the idle thread, b) a UserTransition remains in user mode, c) a UserEventTransition traps from user mode into the kernel, and d) an IdleEventTransition traps from the idle mode into the kernel because of a device interrupt.

sys-trans-rel *user-trans kernel-call* $t \equiv$
  case t **of** KernelTransition $\Rightarrow \{((s, \text{KernelMode}, \text{Some } e), (s', m, \text{None}))\ .\ (s, m, s') \in \textit{kernel-call}\ e\}$
  | UserTransition $\Rightarrow \{((s, \text{UserMode}, \text{None}), (s', \text{UserMode}, \text{None}))\ .\ s' \in \text{fst}\ (\text{split}\ \textit{user-trans}\ s)\}$
  | UserEventTransition $\Rightarrow \{((s, \text{UserMode}, \text{None}), (s, \text{KernelMode}, \text{Some } e))\ .\ \text{True}\}$
  | IdleEventTransition $\Rightarrow \{((s, \text{IdleMode}, \text{None}), (s, \text{KernelMode}, \text{Some Interrupt}))\ .\ \text{True}\}$

where split takes a function f and a pair (r, k) and returns the function application $f\ r\ k$.

Instantiated with the respective kernel state space and with the corresponding user and kernel transitions, this automaton defines the internal states as well as the transition relations Step M of the state machines $\mathcal{M}_\mathsf{A}$, $\mathcal{M}_\mathsf{H}$, and $\mathcal{M}_\mathsf{C}$.

The original work in the L4.verified project was mostly concerned with showing forward simulation for the different representations of *kernel-call*, while computation of user code (*user-trans*) was originally of less concern. Our focus is the opposite: in Section 4, we revisit the different instantiations of the *user-trans* parameter to introduce virtual memory and distinguish individually running processes. In contrast, we do not alter the modelling of kernel transitions *kernel-call*—although we have proven new properties about these transitions.

The next section is concerned with the projections Fin M from the internal to the observable state. This effort simplifies some of our proofs about the changed user transitions, as we discuss in Sections 4.4 and 4.5.

3. Extending the Observable State

In this section, we improve L4.verified's general proof architecture (cf. Section 2.2) such that our data refinement statement reflects the full strength of the auxiliary correspondence lemmas (cf. Def. 4) that L4.verified proved to support their original refinement statement.

Recall that the strength of a data refinement statement hinges on the size of the observable state. When data refinement correlates two state machines, the Fin-projections can filter out parts of the states *at both abstraction layers*. Although the name *observable state* indicates that it should contain all information that the surrounding computation can directly access or acquire by interacting with the module under refinement, the refinement calculus does not require this to be the case—taken to the extreme, the space of observable states could just contain a single element, which would render the according refinement statement meaningless. It is the very nature of refinement that properties can be shown at the abstract layer (where the proof is typically simpler) and then transferred down to the concrete layer. This transfer only works if the property of interest can be expressed in terms of the observable state.

The original state machines $\mathcal{M}_\mathsf{A}$, $\mathcal{M}_\mathsf{H}$ and $\mathcal{M}_\mathsf{C}$ in the initial L4.verified project featured Fin-projections such that the common, observable state comprised only those parts of the kernel state that were encoded almost identically at all three layers of abstraction. In short, these parts comprised all directly user-visible information—like processor registers and the memory content that the kernel assigned to user code—but almost no information about kernel-managed data structures. This is not enough to determine the future behaviour of the system.

Among many other things, the behaviour of kernel and processes depends on kernel objects like page directories (mapping virtual to physical memory), page objects (accounting for physical memory assigned to user code) and capabilities (unforgeable references to kernel objects). Moreover, many invariants shown at the abstract layer $\mathcal{M}_\mathsf{A}$ are concerned with kernel objects. We would like to rely on refinement to infer that equivalent invariants hold at the lower layers $\mathcal{M}_\mathsf{H}$ and $\mathcal{M}_\mathsf{C}$. Section 4 presents an example for this approach: after introducing virtual memory mappings to user transitions, we require a complex state invariant about the respective kernel objects on the low-level specification $\mathcal{M}_\mathsf{H}$. Once proven for the abstract specification, we would like to transfer it down to $\mathcal{M}_\mathsf{H}$ using refinement. This is only possible when the observable state comprises kernel objects.

At the same time, it is the very purpose of refinement to discard irrelevant information. In the implementation, such information might be kept for performance reasons. A typical example is object deletion in the kernel: when the C implementation deletes an object, it simply marks the memory as free but does not reset the memory content. Changing the contents is deferred to when a new object is created. In the meantime, the content of this memory region is irrelevant. The Fin-projections are intended to discard precisely this kind of information.

Concluding, performance is a good reason to keep information at the lower abstraction layers $\mathcal{M}_\mathsf{C}$ and $\mathcal{M}_\mathsf{H}$ although it has no influence on the future behaviour of the system. However, it should disappear in the abstract specification $\mathcal{M}_\mathsf{A}$ to improve clarity. Following that argument, states of $\mathcal{M}_\mathsf{A}$ should comprise precisely the information relevant for system behaviour. This means, $\mathsf{Fin}\ \mathcal{M}_\mathsf{A}$ should be the identity function.

The remainder of this section deals with finding two projections, Π_C from `Simpl` states to low-level specification states and Π_H from low-level specification states to abstract states, such that we can use them to export the full abstract state as the observable state:

Definition 6. The state machine's projections from internal states to the observable state are:

$$\mathsf{Fin}\ \mathcal{M}_\mathsf{A} \equiv (\lambda s.\ s)$$
$$\mathsf{Fin}\ \mathcal{M}_\mathsf{H} \equiv \Pi_\mathsf{H}$$
$$\mathsf{Fin}\ \mathcal{M}_\mathsf{C} \equiv \Pi_\mathsf{H} \circ \Pi_\mathsf{C}$$

where $\circ$ denotes function composition.

Our main concern is fitting Π_H and Π_C retroactively into the original proof architecture. As Section 2.2 describes, L4.verified originally expressed forward simulation between $\mathcal{M}_\mathsf{A}$ and $\mathcal{M}_\mathsf{H}$ with a state relation

$$\mathsf{SR}'_\mathsf{A} \equiv \mathsf{SR}_\mathsf{A} \cap \{(s_\mathsf{A}, s_\mathsf{H}).\ \mathsf{I}_\mathsf{A}\ s_\mathsf{A} \wedge \mathsf{I}_\mathsf{H}\ s_\mathsf{H}\}$$

composing a deliberately weak state relation SR_A with state invariants I_A and I_H about the internal states of the respective layers. Extending the observable state mainly affects the third proof obligation of forward

simulation: for any abstract specification state s_A and any low-level specification state s_H correlated by SR'_A, the observable states projected out by $\mathsf{Fin}\ \mathcal{M}_\mathsf{H}$ and $\mathsf{Fin}\ \mathcal{M}_\mathsf{A}$ should be equal:

$$\forall (s_\mathsf{A}, s_\mathsf{H}) \in \mathsf{SR}'_\mathsf{A}.\quad \mathsf{Fin}\ \mathcal{M}_\mathsf{H}\ s_\mathsf{H} = \mathsf{Fin}\ \mathcal{M}_\mathsf{A}\ s_\mathsf{A}$$

Or using Def. 6:

$$\forall (s_\mathsf{A}, s_\mathsf{H}) \in \mathsf{SR}'_\mathsf{A}. \qquad \Pi_\mathsf{H}\ s_\mathsf{H} = s_\mathsf{A}$$

Thus, the function Π_H should return the abstract state s_A that is correlated to the low-level specification state s_H given as function argument. This implicitly assumes that SR'_A is injective (also called left-unique), i. e. that there is at most one s_A correlated with s_H:

$$(s_\mathsf{A}, s_\mathsf{H}) \in \mathsf{SR}'_\mathsf{A}\ \wedge\ (s'_\mathsf{A}, s_\mathsf{H}) \in \mathsf{SR}'_\mathsf{A}\ \longrightarrow\ s'_\mathsf{A} = s_\mathsf{A}$$

Unfortunately, this assumption does not hold for the original SR'_A. To make SR'_A injective, we have to strengthen the state invariants I_A and I_H. Once injective, we could theoretically define Π_H directly as the function that takes a state s_H and returns the unique abstract state s_A such that the pair $(s_\mathsf{A}, s_\mathsf{H})$ is in SR'_A. Isabelle/HOL provides the definite choice operator[3] to state this definition formally but requires us to show the wellformedness of this definition (i. e. the uniqueness of s_A for any given s_H) each time we use the operator. Practically, spelling out an explicit definition for Π_H is easier to use and has the additional benefit of documenting the correlation between low-level specification states s_H and abstract states s_A. Hence, we manually defined the state projection function and then proved the following lemma:

Lemma 3 (Correctness of Projection Π_H). If an abstract specification state s_A and a low-level specification state s_H are correlated and the respective invariants hold, the projection Π_H of s_H yields s_A:

$$\mathsf{I}_\mathsf{A}\ s_\mathsf{A}\ \wedge\ \mathsf{I}_\mathsf{H}\ s_\mathsf{H}\ \wedge\ (s_\mathsf{A}, s_\mathsf{H}) \in \mathsf{SR}_\mathsf{A}\ \longrightarrow\ \Pi_\mathsf{H}\ s_\mathsf{H} = s_\mathsf{A}$$

The same holds respectively for the projection Π_C and the state machines $\mathcal{M}_\mathsf{H}$ and $\mathcal{M}_\mathsf{C}$: L4.verified originally expressed forward simulation between the two state machines with a composed state relation $\mathsf{SR}_\mathsf{C} \cap \{(s_\mathsf{H}, s_\mathsf{C}).\ \mathsf{I}_\mathsf{H}\ s_\mathsf{H}\ \wedge\ \mathsf{I}_\mathsf{C}\ s_\mathsf{C}\}$. Again, this relation was not injective and we have strengthened the original state invariant I_H to be able to define Π_C such that the following lemma becomes true:

Lemma 4 (Correctness of Projection Π_C). If a low-level specification state s_H and an implementation state s_C are correlated and invariant I_H holds for s_H, the projection Π_C of s_C yields s_H:

$$\mathsf{I}_\mathsf{H}\ s_\mathsf{H}\ \wedge\ (s_\mathsf{H}, s_\mathsf{C}) \in \mathsf{SR}_\mathsf{C}\ \longrightarrow\ \Pi_\mathsf{C}\ s_\mathsf{C} = s_\mathsf{H}$$

Although conceptually simple, the formal definitions of Π_H and Π_C as well as the proofs of correctness w. r. t. the state relations are technically quite involved—and certainly specific to the seL4 kernel. Hence, we do not present details but summarise typical problems at a more general level that are likely to appear in similar contexts. In particular, we describe the following three problems in more detail:

- missing information at lower abstraction layers,
- multiple representations of the same semantic value, and
- differences in the granularity of heap entries.

Finally, we summarise our insights at the end of this section.

Missing information at lower abstraction layers. Originally, the states of the abstract specification $\mathcal{M}_\mathsf{A}$ contained additional information that we could not infer from the states at the lower abstraction layers $\mathcal{M}_\mathsf{H}$ and $\mathcal{M}_\mathsf{C}$. This additional information stemmed from two sources: a) irrelevant information, which has been left over after the removal of a kernel object, and b) static configuration data, which is only used to state some invariants at the abstract layer.

The irrelevant information is certainly dispensable, so we have changed the abstract specification to erase the information together with the kernel object. The configuration data mentioned above partitions the virtual address space into regions reserved for special purposes, on the one hand, and memory regions available to the user, on the other hand. The abstract invariant I_A refers to this data and, among other

[3] The definite choice operator The is axiomatised as $\mathsf{The}\,(\lambda x.\ x = a) \equiv a$.

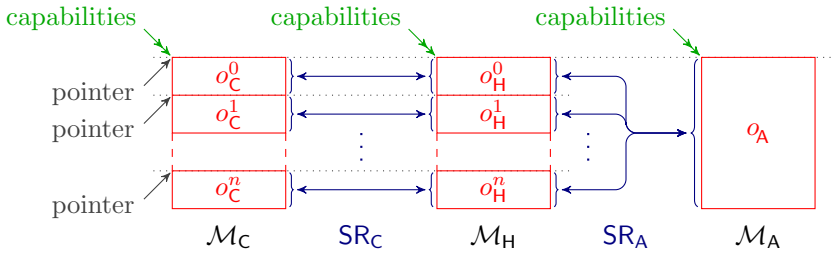

Fig. 5. An example kernel object with its heap entries at different abstraction layers and their correlation

things, states that the kernel indeed complies with this partitioning. Thus, the data is necessary to transfer the invariant I_{A} down to the lower layers. Hence, we have amended the lower layers with it and strengthened the state relations SR_{A} and SR_{C} accordingly.

Multiple representations of the same semantic value. A problem similar to the above is that the data representation on the abstract layer occasionally uses wider types than on the more concrete layers, resulting in multiple representations of the same semantic value. We illustrate this general problem by a concrete example: the mappings from virtual to physical addresses are stored in page tables together with access permissions for the mapped region of memory. Page tables are data structures shared between processor and hardware MMU; hence, their data format is prescribed by the hardware architecture.

Consequently, the `Simpl` layer $\mathcal{M}_{\mathsf{C}}$ uses this data format, where access permissions are determined by a combination of three bits. Not all combinations are meaningful and not all hardware-supported permissions are used by the seL4 kernel. The low-level specification $\mathcal{M}_{\mathsf{H}}$ introduces its own data type and limits permissions to only four values: VMNoAccess, VMKernelOnly, VMReadOnly and VMReadWrite. The abstract specification $\mathcal{M}_{\mathsf{A}}$, in contrast, reuses the same type for memory access permissions as for capability rights, which contains more values than necessary for permissions. Namely, capability rights are any subset of $\{\mathsf{Read}, \mathsf{Write}, \mathsf{Grant}\}$. However, the Grant right is meaningless in the context of memory; moreover, the hardware does not allow for mapping a region of memory as write-only (corresponding to $\{\mathsf{Write}\}$).

Originally, the abstract specification had multiple representations for memory access permissions, where meaningless rights in the set were disregarded (e. g. $\{\mathsf{Read}, \mathsf{Grant}\} \mathrel{\widehat{=}} \{\mathsf{Read}\}$). Naturally, the state relation $\mathsf{SR}'_{\mathsf{A}}$ cannot be injective in this case. Thus, we have changed the abstract specification to use one representation per permission consistently. We have strengthened the abstract invariant I_{A} to allow only these representations and proved that the invariant holds.

Note that our problem mainly arises from a redundancy at the abstract specification layer: capability rights and memory access permissions coincide. In other words, we could theoretically reconstruct the memory access permissions from capability rights. In practice, this reconstruction is a complex operation and removing the redundancy would invalidate a substantial fraction of the original refinement proof.

Differences in the granularity of heap entries. This problem is more specific to seL4, although other capability-based systems are likely to have similar issues. Recall that as kernel, seL4 is responsible for resource management: it creates, manipulates and eventually deletes kernel objects like page tables or communication channels on behalf of user threads. Since kernel objects can be created and deleted during runtime, the kernel stores the respective data structures at the kernel heap. User threads refer with capabilities to existing objects (cf. Section 2.1). A layer of indirection protects capabilities against forgery: a capability just refers to a row within a kernel-managed capability table, which itself contains a pointer to the kernel object.

With these prerequisites in place, we can now focus on the kernel models. At the implementation, many kernel objects comprise a collection of equally typed data structures: a page table, for instance, comprises 256 page table entries. Thus the `Simpl` layer $\mathcal{M}_{\mathsf{C}}$ treats each of these small data structures as a separate entry in the heap (cf. Section 2.3). For simplicity, the low-level specification $\mathcal{M}_{\mathsf{H}}$ preserves this granularity of heap entries. The abstract specification $\mathcal{M}_{\mathsf{A}}$, in contrast, combines all entries into one large entry in the heap that corresponds with the user's notion of a kernel object.

Fig. 5 shows heap entries of different granularity representing a single kernel object: at the lower layers $\mathcal{M}_{\mathsf{C}}$ (left) and $\mathcal{M}_{\mathsf{H}}$ (center), the object consists of several heap entries $o_{\mathsf{C}}^0, \ldots, o_{\mathsf{C}}^n$ and $o_{\mathsf{H}}^0, \ldots, o_{\mathsf{H}}^n$ while at the abstract specification layer $\mathcal{M}_{\mathsf{A}}$ (right), there is only a single heap entry o_{A} for the whole kernel object.

We see that capabilities only point to the first heap entry of each layer (o^0_C, o^0_H, o_A) because from the user's perspective, there is only one kernel object. From the kernel's perspective, however, pointers (illustrated to the left) refer to the different heap entries belonging to this object. Braces and arrows between the abstraction layers illustrate how the state relations SR_C and SR_A relate the heap entries at adjacent layers: while the relation is one to one between o^i_C and o^i_H for all i, the collection of all heap entries $o^0_\mathsf{H}, \ldots, o^n_\mathsf{H}$ is related to the one abstract heap entry o_A. Dotted lines illustrate that heap addresses are equal at all three abstraction layers; for instance, the pointer value for o^0_C is the same as for o^0_A.

Kernel objects are aligned to their size, i. e. the size of the kernel object divides the pointer value for o^0_C. Thus, we can compute the pointer value for o^0_A from the one for any o^i_H by dividing it by the kernel object's size—assuming the latter is known. Unfortunately, some kernel objects can have different sizes; namely, capability tables and memory pages. For kernel objects of variable size, the object size is stored in the capability table entry. Recall that users can only refer to a kernel object by capabilities; consequently, an object is useless unless a capability points to it. Thus, we could in theory reconstruct the object size from the capabilities pointing to it. In practice, this approach poses two challenges: a) the relationship is rather indirect and b) we would rely on complex invariants to correlate states.

The relationship is indirect because we need two stages to correlate kernel heaps at the two abstraction layers: first, scanning through the kernel heap of $\mathcal{M}_\mathsf{H}$ to fetch all capability table entries and to reconstruct the size for variable kernel objects, and second, comparing the objects of corresponding types and sizes in the kernel heaps of $\mathcal{M}_\mathsf{A}$ and $\mathcal{M}_\mathsf{H}$ pairwise with each other.

More importantly, reconstructing the object sizes from capabilities would intertwine the state relation SR_A with two complex invariants: first, there is always at least one capability to each kernel object, and second, capabilities always point to a valid kernel object. Although these properties are part of the overall, abstract invariant I_A, this invariant is only preserved by big steps $m^{\mathsf{big}}_\mathsf{A}$. For a short fraction of such a big step, objects may outlive their capabilities. Thus, correlating intermediate states would become a challenge if we were relying on capabilities to reconstruct object sizes.

We have circumvented the problems above by amending the lower layers with ghost state and code that explicitly records the object sizes.

Conclusion. In this section, we have strengthened the top-level refinement statement by extending the observable state to the entire abstract state. This is the strongest, achievable data refinement between an abstract specification $\mathcal{M}_\mathsf{A}$ and a `Simpl` representation $\mathcal{M}_\mathsf{C}$. In fact, Lemma 4 facilitates a data refinement between a low-level specification $\mathcal{M}_\mathsf{H}$ and a `Simpl` representation $\mathcal{M}_\mathsf{C}$, where the entire low-level specification state is observable: it discharges the third proof obligation of forward simulation, while the second proof obligation is independent of the observable state (and already proven). The only remaining effort lies in the definition of suitable $\mathsf{Init}\ M$ functions and discharging the first proof obligation of forward simulation.

Our effort has immensely profited from the earlier L4.verified project. The overall proof architecture and an uncounted number of auxiliary lemmas have from the project's start aimed at the strong top-level statement that we have finally proven. Thus, we initially expected that reworking the abstract specification and strengthening the state invariants would require a proof effort of only a few months. Ultimately, we estimate our overall effort for the extension of the observable state with 7–8 person-months. Although considerably higher than expected, this effort is still small compared to about 11 person-years that the original refinement proof did cost. The prediction of the proof effort has been so difficult because of many subtle interdependencies between the different predicates that contribute to the overall invariant. These subtleties often emerge at only one place in an otherwise very large proof such that the process of discovering a suitable invariant involves many iterations.

In principle, we could have reduced the required proof effort by defining $\mathsf{Fin}\ \mathcal{M}_\mathsf{A}$ as a lightweight projection from abstract states to abstract states instead of the identity. This projection could eliminate information from the abstract state that is missing in the lower layers and consistently choose one of multiple encodings per semantic value. Thus, we could have circumvented the first two problems mentioned above (information missing from lower layers and multiple representations of one semantic value).

We did not take this approach for several reasons. First, we estimate that these problems required at most a forth of our total effort. Second, even if $\mathsf{Fin}\ \mathcal{M}_\mathsf{A}$ is a lightweight projection, any projection comes at the cost of an additional layer of indirection, which requires additional effort to formally establish that properties shown at the abstract layer can be transferred down to the implementation layer. Third, the stronger invariants and an unambiguous state encoding improve readability and clarity of the specification

and simplify further proofs at the specification layer. Hence, we are confident that our additional effort for identifying the abstract state with the observable state will eventually pay off.

With the extended observable state in place, we can now focus on our main contribution: basing user transitions on virtual memory. The projection functions Π_{H} and Π_{C} will not only help us in redefining user transitions in Section 4.2, but our revised proof architecture also alleviates necessary invariant proofs in Section 4.4.

4. User Transitions with Virtual Memory

This section introduces a notion of virtual memory to seL4, thereby distinguishing different processes, and thus remodelling L4.verified's original user transitions. In Section 4.1, we elaborate on the definition of an abstract MMU based on the kernel data structures. With the MMU in place, we can model user transitions more precisely (Section 4.2) and thus strengthen the top-level refinement statement. We reestablish this refinement statement by strengthening the invariants at both specification layers (Sections 4.3 and 4.4) and adjusting the correspondence (Section 4.5) proofs accordingly.

4.1. Abstract Memory-Management Unit

Most processors facilitate the separation of individual processes with virtual memory: when executing assembly instructions, the hardware MMU transparently translates virtual memory addresses into physical addresses. The translation is directed by data structures stored in physical memory; the MMU finds a pointer to them in a special register. This special register is protected, i. e. only the kernel can access it. Hence, each process can have a different *address space*—a particular mapping from virtual to physical addresses. The seL4 kernel runs on the ARMv6 architecture, which features a MMU that supports two stages of translation: the special register points into a *page directory*, a table that might itself contain pointers to another table, the *page table*, which finally maps virtual to physical addresses.[4]

Like other resources, seL4 manages the mapping data structures of the hardware; i. e. it creates, manipulates and deletes page directories and page tables (cf. Section 2.3). Thus, there is no need for adding state to our kernel models; we can simply project the virtual memory mappings out of the existing state, which already models these hardware data structures. Given that we need a model of user transitions for each abstraction layer, we require a notion of virtual memory mappings at each layer. We achieve this by defining an MMU on an abstract specification state and use the projections Π_{C} and Π_{H} from Section 3 to inherit equivalent definitions at the lower abstraction layers.

More specifically, we aim at functions virt-to-phys and vm-perms. Both take an abstract kernel state and a virtual address. Provided there is a translation of this virtual address for the current thread, virt-to-phys returns the corresponding physical address, and vm-perms returns the access permissions associated with the virtual address. These functions provide a convenient interface for user transitions. Internally, both functions perform the same lookup (in fact, their formal definitions utilise the same auxiliary functions) to provide their respective functionality: at first, they find the respective page directory, and then, they look up the mapping data structures just like the hardware does.

In the following, we first describe how both lookup stages work and then formally define the functions virt-to-phys and vm-perms as the interface for user transitions.

Page directory search. Using Fig. 6, we explain how the page directory for the current thread is found in the abstract specification. The figure depicts the relevant parts of an abstract kernel state; the names of the kernel state variables are on the left and their structured values on the right. Three variables of the kernel state are involved in a successful search: the address-space table asid-table, the current thread pointer cur-thread and the abstract kernel heap kheap. At first, we dereference cur-thread using kheap to find the thread object for the current thread. If there is a valid thread object in the kernel heap, we check the respective entry for the page directory (called vtable) for a capability (cap) with some page directory pointer pd and mapping data (p, i). The mapping data provides a pointer p into the asid-table, which itself

[4] The seL4 kernel uses the ARMv6 coarse page table format with subpages disabled.

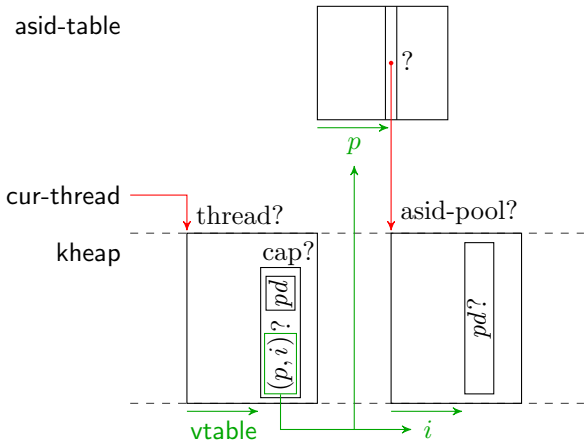

Fig. 6. Page directory search

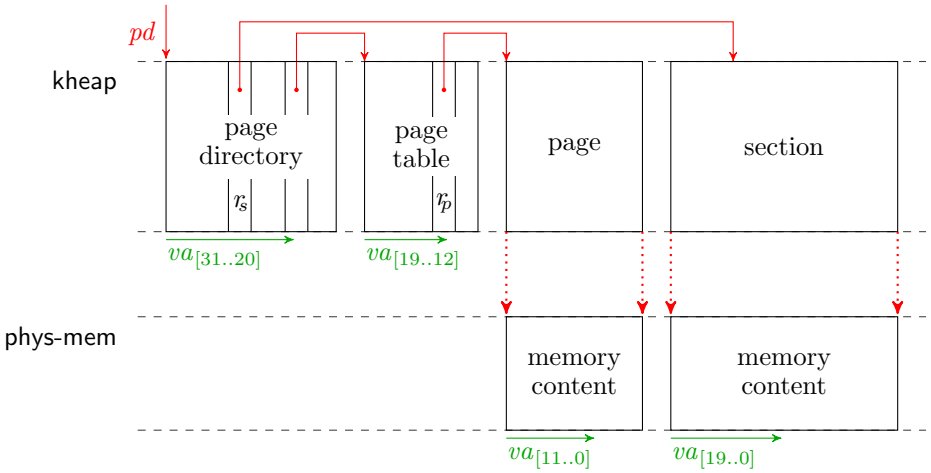

Fig. 7. Virtual memory mappings

might contain a pointer into the kernel heap. We dereference the latter in search for an address-space pool (asid-pool). If the address-space pool is present, we use the i of the capability's mapping data as index into the pool. If the entry found in the pool is the same page directory pointer pd as the one stored in the vtable-capability, then we return it as the pointer to the page directory of the current thread. If the search fails in any of the above stages, we resort to the master page directory (master-pd), another kernel variable pointing to a page directory with minimal, static mappings for kernel-only access. We formally capture this search in the function get-mmu-info but omit its definition here.

The original model of seL4's execution environment did not specify this search; we have now added it to our new model. Technically, there is still a slight deviation from the hardware's behaviour: on hardware, the kernel searches for the page directory pointer only once at kernel exit and then activates it by storing it in a special register; we, in contrast, defer the search to the time when we actually need the page directory. As far as the C code is concerned, this difference is irrelevant because accessing the special register requires assembly. Thus, we postpone the refinement of this aspect for translation validation of the compiled kernel binary. Myreen and Sewell [SMK13] have achieved first results on this although their hardware model does not yet include a MMU.

Mapping-data lookup. Fig. 7 illustrates how a 32-bit virtual address va is translated into a physical address once the page directory is known. Again, the figure shows the names of the relevant variables of the

abstract kernel state on the left and their structured values on the right. We assume a known pointer *pd* into the kernel heap kheap, referring to a page directory. Then we use the highest thirteen bits of the virtual address $va_{[31..20]}$ as an index into this page directory. A directory might contain three kinds of entries: *invalid* entries (in which case the memory access fails), entries for so-called *sections* (pointing to large contiguous memory regions) and entries for *page tables* (pointing to tables for the second stage of translation).

An entry for a section contains a pointer to a section object in the kernel heap. Section objects only serve as placeholders; the memory content at *va* is found at the corresponding offset in state variable phys-mem. Note that a section entry also contains access permissions r_s, which might restrict memory access to read-only or kernel-only (we came across permissions already in Section 3 when discussing multiple representations of one semantic value).

An entry for a page table contains a pointer to a page table object in the kernel heap. The next eight bits of the virtual address $va_{[19..12]}$ are used as an index into the page table, which has invalid entries and entries for pages. Similarly, entries for pages contain a pointer to page objects as well as access permissions. And again, the actual memory content is found in phys-mem.

We formally capture this mapping-data lookup in the function get-pd-for-thread but omit its definition here. It finds the mapping data for a given kernel heap h, a page directory pointer *pd* and a virtual address *va*. If the lookup finds an invalid entry, the function returns None; otherwise it returns a triple Some $(base, bits, perms)$, where *base* is a pointer to the page or section object, 2^{bits} is the size of the object in bytes and *perms* is a set representing the access permissions associated with *va*. Possible values are $\emptyset$, $\{\mathsf{Read}\}$ and $\{\mathsf{Read}, \mathsf{Write}\}$.

MMU Interface. Based on the internal functions get-pd-for-thread and get-mmu-info, we can finally define the external interface of the MMU:

Definition 7 (Address Translation). The function virt-to-phys computes the physical address that is associated in an abstract specification state s_A with a given virtual address *va* (if any). It uses get-pd-for-thread to find out the page directory *pd* of the current thread and with that, it interprets the result of get-mmu-info: if None, the address is not mapped and virt-to-phys returns None; otherwise, virt-to-phys returns the offset into phys-mem, where the memory content is located.

virt-to-phys s_A $va \equiv$
let $pd =$ get-pd-for-thread (kheap s_A) (asid-table s_A) (master-pd s_A) (cur-thread s_A)
in case get-mmu-info (kheap s_A) pd va **of**
  None $\Rightarrow$ None
| Some $(base, bits, perms) \Rightarrow$ Some $(base + (va \mathbin{\&\&} (2^{bits} - 1)))$

where && denotes bitwise and.

Definition 8 (Virtual-Memory Permissions). The function vm-perms computes the access permissions associated with a given virtual address. For unmapped virtual addresses, it returns the empty set; otherwise the set of permissions *perms* determined by get-mmu-info.

vm-perms s_A $va \equiv$
let $pd =$ get-pd-for-thread (kheap s_A) (asid-table s_A) (master-pd s_A) (cur-thread s_A)
in case get-mmu-info (kheap s_A) pd va **of**
  None $\Rightarrow \emptyset$
| Some $(base, bits, perms) \Rightarrow perms$

With these definitions in place, we can now base user transitions on virtual memory.

4.2. Changing the Definition of User Transitions

Using the above abstract MMU, we can now introduce virtual memory to user transitions and thus distinguish individual processes. To do so, we first take a closer look at how user transitions were defined originally.

Recall that the L4.verified project originally overapproximated user transitions, assuming that the current thread has read and write access to all memory that the kernel has assigned to user code. The kernel marks memory assigned to user code by page and section objects, which we have mentioned in our discussion of the mapping-data lookup (cf. Section 4.1). Formally, L4.verified modelled reading and writing to user-assigned

memory by the two functions user-memory and user-memory-upd. The function user-memory takes an abstract specification state s_A and a physical address pa and returns the corresponding memory content for pa iff it is assigned to user code. In particular, user-memory inspects the kernel heap kheap to find a page or section containing pa. If successful, it retrieves the memory content from the kernel state variable phys-mem; otherwise, it returns None. Note that we might perceive user-memory s_A as a partial function from physical addresses to memory contents. Conversely, the function user-memory-upd takes such a partial function um and updates variable phys-mem at all physical addresses for which um is defined with the corresponding value of um. Based on these functions, L4.verified defined user transitions of the abstract specification as follows:

Definition 9 (*Original* Abstract User Transition). A user transition user-trans$_A$ of the abstract specification consisted of four steps: First, it projected the contents um of all user-assigned physical memory out of the kernel state. Second, a select command non-deterministically chose the new register context tc' and the new memory contents um' such that the domains of um and um' are equal. Third, user-trans$_A$ partially updated the physical memory according to um'. Fourth, it returned the new register context.

user-trans$_A$ $tc \equiv$ **do**
  $um \leftarrow$ gets user-memory;
  $(tc', um') \leftarrow$ select $\{(tc', um').$ dom $um =$ dom $um'\}$;
  modify (user-memory-upd $(\lambda s.\ um')$);
  return tc'
od

When we amend user transitions with virtual memory, we preserve this general structure but we restrict the domain of user memory to translated addresses. This restriction is twofold: on the one hand, we limit the non-determinism of the select statement—it should not depend on memory content that the current thread cannot read—and on the other hand, we constrain the effects of the modify statement—it should not change memory that the current thread cannot write to. The former restriction ensures confidentiality of user threads, the latter preserves their integrity. Confidentiality and integrity are the two complementing concepts underlying *non-interference*, the formal concept of process separation.

Our restriction relies on information about the current kernel state. More specifically, we use the functions cur-thread, virt-to-phys, and vm-perms to project out the current thread t, the partial function $conv$ translating virtual to physical addresses, and the corresponding access permissions $perms$. As mentioned in Section 1, we furthermore introduce a parameter *user-op* to user transitions, which is a function computing a set of new thread states from various information about the current state. In particular, *user-op* takes as parameters t, $conv$ and $perms$ and the current thread state (register and memory contents). To restrict the domain of the thread's memory contents, we use the Isabelle/HOL function restrict-map, which takes a partial function f and a set R as parameters and computes the restriction of f to R. We thereby ensure that the result of *user-op* can only depend on memory the user has Read access to, and can only modify memory it has Write access to.

Formally, we replace Def. 9 by:

Definition 10 (*New* Abstract User Transition).

user-trans$_A$ *user-op* $tc \equiv$ **do**
  $t \leftarrow$ gets cur-thread;
  $conv \leftarrow$ gets virt-to-phys;
  $perms \leftarrow$ gets vm-perms;
  $um \leftarrow$ gets user-memory;
  $(tc', um') \leftarrow$ select (*user-op* t $conv$ $perms$
    $(tc,$ restrict-map um $\{pa.\ \exists va.\ conv\ va =$ Some $pa\ \wedge$ Read $\in perms\ va\}))$;
  modify (user-memory-upd
    $(\lambda s.$ restrict-map um' $\{pa.\ \exists va.\ conv\ va =$ Some $pa\ \wedge$ Write $\in perms\ va\}))$;
  return tc'
od

We apply effectively the same changes to the low-level specification and the `Simpl` representation. Note that we need the projections Π_H and Π_C to convert virt-to-phys and vm-perms to the respective states. On

the `Simpl` layer, for instance, the second and third command thus read as follows:

$$conv \leftarrow \mathsf{gets}\ (\mathsf{virt\text{-}to\text{-}phys} \circ \Pi_\mathsf{H} \circ \Pi_\mathsf{C});$$
$$perms \leftarrow \mathsf{gets}\ (\mathsf{vm\text{-}perms} \circ \Pi_\mathsf{H} \circ \Pi_\mathsf{C});$$

Apart from such syntactic differences, the user transitions are defined analogously at all three abstraction layers.

Summarising, the original modelling of user transitions assumed that any user transition might potentially change all physical memory that the kernel has assigned to user threads. This original modelling amounts to bypassing the page directory search (cf. Fig. 6) and the mapping-data lookup (cf. Fig. 7) and assuming that all page and section objects in the kernel heap might potentially be reached by any user thread. We have revised the user transitions such that they now examine, which thread is currently running, and limit memory accesses according to the permissions of virtual memory. Thus, our modelling allows us to distinguish individual threads and their address spaces. Moreover, we can now express non-interference of (i. e. isolation between) individual processes. Our new modelling is also more precise: it turns a previously implicit assumption of the original modelling into three proof obligations. This aspect will be the topic of the subsequent three sections.

4.3. Strengthening the Invariants about Abstract States

The overall proof architecture of L4.verified establishes data refinement by subdividing the problem into correspondence proofs between state monads of adjacent layers (e. g. Lemma 1) and proofs about state monads preserving invariants (e. g. Lemma 2). By changing the definitions of the user transitions $\mathsf{user\text{-}trans}_\mathsf{A}$, $\mathsf{user\text{-}trans}_\mathsf{H}$ and $\mathsf{user\text{-}trans}_\mathsf{C}$, we have implicitly changed the lemmas about these transitions, which causes their respective proofs to break. Below, we reestablish these lemmas for the revised definitions. In this section, in particular, we aim for invariant preservation of $\mathsf{user\text{-}trans}_\mathsf{A}$:

Lemma 5. The *new* user transitions at the abstract specification layer preserve the invariants:

$$\{\!|\lambda s_\mathsf{A}.\ \mathsf{I}_\mathsf{A}\ s_\mathsf{A} \wedge \mathsf{ut\text{-}pre}_\mathsf{A}\ s_\mathsf{A}|\!\}\ \mathsf{user\text{-}trans}_\mathsf{A}\ \mathit{user\text{-}op}\ \mathit{tc}\ \{\!|\lambda r\, s_\mathsf{A}.\ \mathsf{I}_\mathsf{A}\ s_\mathsf{A}|\!\}$$

Concluding from the previous section, the main intention of replacing Def. 9 by Def. 10 has been refinement; so, ideally, we would be able to establish correspondence with identity relations for states and results between $\mathsf{user\text{-}trans}_\mathsf{A}^\mathsf{orig}\ \mathit{tc}$ and $\mathsf{user\text{-}trans}_\mathsf{A}^\mathsf{new}\ \mathit{user\text{-}op}\ \mathit{tc}$ for all user operations *user-op* and register contents *tc* when assuming I_A and $\mathsf{ut\text{-}pre}_\mathsf{A}$ for the respective states:

$$\begin{aligned}&\mathsf{corres}\ \{(s_a, s_c).\ s_a = s_c\}\ \{(r_a, r_c).\ r_a = r_c\}\\ &\quad(\lambda s_a.\ \mathsf{I}_\mathsf{A}\ s_a \wedge \mathsf{ut\text{-}pre}_\mathsf{A}\ s_a)\ (\lambda s_c.\ \mathsf{I}_\mathsf{A}\ s_c \wedge \mathsf{ut\text{-}pre}_\mathsf{A}\ s_c)\\ &\quad(\mathsf{user\text{-}trans}_\mathsf{A}^\mathsf{orig}\ \mathit{tc})\ (\mathsf{user\text{-}trans}_\mathsf{A}^\mathsf{new}\ \mathit{user\text{-}op}\ \mathit{tc})\end{aligned}$$

If this were the case, we could easily infer Lemma 5 from Lemma 2. Unfortunately, however, we need a stronger precondition on the states s_c for $\mathsf{user\text{-}trans}_\mathsf{A}^\mathsf{new}$ to correspond to $\mathsf{user\text{-}trans}_\mathsf{A}^\mathsf{orig}$.

In more detail, Def. 9 explicitly restricts the domain of the new user memory um' to be equal to the one of the initial user memory um. Thus, the modify statement only affects physical memory addresses *pa* assigned to user code (i. e. $pa \in \mathsf{dom}\ um$). In other words, the original modelling implicitly assumed that the collective memory accessibe by all threads via virtual memory were indeed captured by the kernel-internal accounting as user memory.

Def. 10, in contrast, restricts memory updates to those physical addresses, to which the current thread has Write permission. Thus, the new modelling turns the above assumption into a proof obligation: for Lemma 5 to hold, the set of user-writable physical addresses must not be larger than the domain of the physical memory assigned to user code; otherwise, user transitions could update physical memory assigned to kernel objects and thereby break kernel invariants. The following lemma formally reflects our informal considerations.

Lemma 6 (Correctness of Memory Assignment). If the current thread can access a physical memory address *pa* through virtual memory, the kernel has assigned it to user code.

$$\mathsf{I}_\mathsf{A}\ s_\mathsf{A} \wedge \mathsf{vm\text{-}perms}\ s_\mathsf{A}\ va \neq \emptyset \wedge \mathsf{virt\text{-}to\text{-}phys}\ s_\mathsf{A}\ va = \mathsf{Some}\ pa \longrightarrow \mathsf{user\text{-}memory}\ s_\mathsf{A}\ pa \neq \mathsf{None}$$

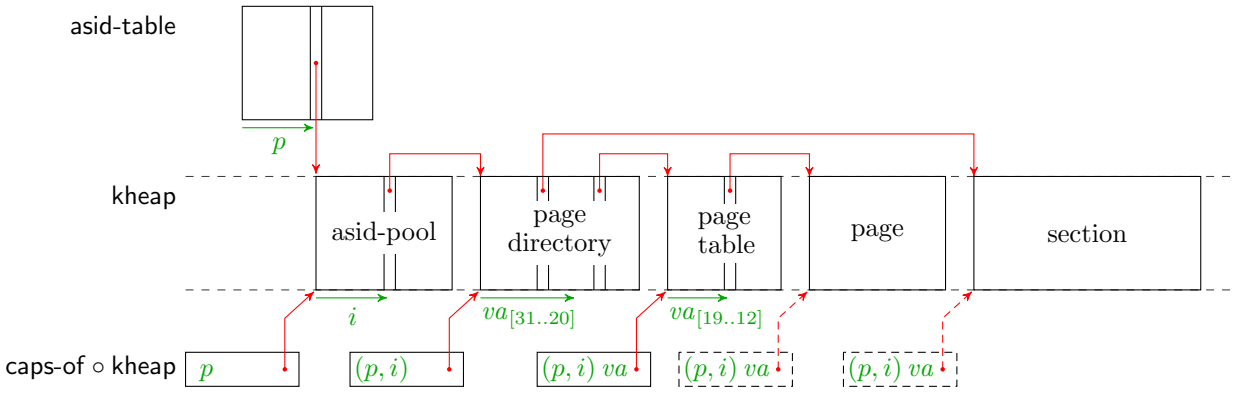

Fig. 8. Invariant: mapping structures are backed by capabilities

Unfortunately, the original invariant $\mathsf{I_A}$ was not strong enough to prove this lemma. Our verification work flow was an iterative process: we strengthened predicates contributing to $\mathsf{I_A}$ and tried to prove that the new, overall predicate $\mathsf{I_A}$ is indeed invariant. In fact, strengthening the invariant to prove this lemma induced most of the effort resulting from the changed user transitions. We estimate that we spent about seven person-months to show it. We exemplify the two most elaborate strengthened predicates.

Page Table Entries are backed by Page Objects. Fig. 7 already hints at this property: when a thread object points to a page directory, all its entries referring to a section must indeed point to a section object in the kernel heap, and respectively, if there is a page directory entry to a page table, all entries of the page table referring to a page must indeed point to a page object in the heap. Note that this predicate only applies to page directories that a thread object points to and to page tables that these page directories point to. We collectively call them *reachable* mapping structures.

The original abstract invariant $\mathsf{I_A}$ stated that if an entry in a reachable page directory refers to a page table, there must be a page table object in the kernel heap at the respective address. However, there was originally no requirement that a user-accessible page referred to by an entry of a reachable page table is backed by a page object in the kernel heap. In other words, there was no requirement that the physical memory user-accessible through virtual memory is actually assigned to user code. We strengthened the validity predicate for page tables such that user-accessible page entries are backed by page objects. This change by itself spawned almost a quarter of our proof work for Lemma 6.

Pages Objects are Backed by Capabilities. Similarly, the original invariant required that for each reachable mapping structure up to page tables, a) there exists a capability referring to it and b) all capabilities referring to it contain correct mapping information. Fig. 8 illustrates both properties—as usual, the names of the relevant state variables are to the left and the structured values to the right. Solid lines mark the original constraints and dashed lines our new additions.

First, starting from the entries of the address-space table asid-table, we recursively follow the pointers in the mapping structures, and for each mapping structure that we find along the path—namely address-space pools, page directories and page tables, we require that there is a capability referring to it. For easier reference to capabilities, the figure presumes a function caps-of, projecting the set of all capabilities out of a given kernel heap (capabilities can be stored in thread objects as well as in capability tables).

Second, capabilities for mapping structures can carry mapping information, which is intended for recovering the mapping path. The capability pointing to the page table in the center of Fig. 8, for instance, carries the pair (p, i) and a virtual address va. Thus, we can eventually find the page table by first using p as index into the address-space table, following this pointer, we find an address-space pool, in which we use i as an index to find a pointer to a page directory, in which we use the upper 12 bit of va to find a pointer to the page table. For any reachable mapping structure, the mapping information in all capabilities pointing to it must be correct. This is important upon object deletion, which we explain with an example.

Consider the state as shown in Fig. 8 and assume, we are to remove the last capability to the page table. In seL4, if the last capability to an object is removed, the object is destroyed because the user cannot refer to

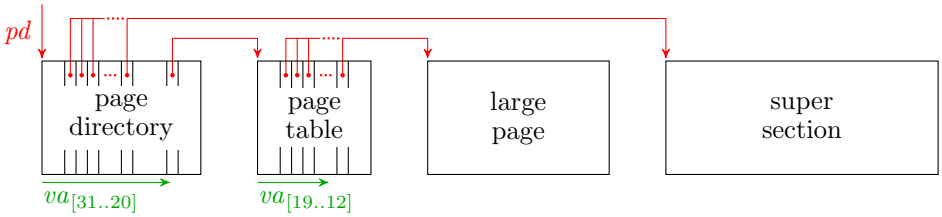

Fig. 9. Virtual memory mappings in hardware

it any more. However, there might still be kernel-internal references to the object. To avoid them becoming dangling references, the kernel needs to find and remove them before it reclaims the object's memory. For the page table in the example, this reference is stored in an entry of the parent page directory (to the left in the figure). The kernel finds this entry by following the mapping information (p, i) and va in the page table's capability and removes the reference, thereby unmapping the page table. Condition b) ensures that this lookup leads to the correct entry.

Note that after this unmap operation, the page in Fig. 8 becomes unreachable. Yet, the kernel has not removed the now invalid mapping data in the page's capability. Doing so could be a costly operation because a page table typically refers to many pages and many capabilities may in turn refer to these pages. As Conditions a) and b) are required for reachable mapping structures only, we can avoid this runtime overhead. Hence, the kernel ignores incorrect mapping data upon deletion—it starts following the mapping data, finds out that it is incorrect and assumes that an entry in its ancestors has previouly been cleared, making the object at hand unreachable. Consequently, incorrect mapping data for a *reachable object* could lead to a state violating Lemma 6 because the kernel might reclaim the object's memory without clearing the entries in the parent mapping structures—thus, for instance, leaving a page table entry dangling while the page object it once pointed to has been removed.

Originally, the absence of dangling references has only been shown up to page tables; we now extend it to pages and sections. In other words, we strengthened the original invariant $\mathsf{I_A}$ to additionally require that if a reachable page directory refers to a section or a reachable page table refers to a page, there must be a capability referring to the respective object, and all capabilities referring to a reachable section or page must bear the correct mapping information. About three quarters of our proof effort for Lemma 6 is related to this added requirement.

There were two factors which made extending this predicate even more involved for sections and pages than it had been for page directories and tables. First, seL4 allows page directories and page tables to be mapped only once while sections and pages can be mapped at multiple virtual addresses and into multiple address spaces. Consequently, we may have page capabilities with disagreeing mapping information because the page is mapped twice.

Second, the used ARM hardware supports pages of different sizes while there is only one size for page directories and page tables. In particular, the hardware MMU supports so-called *super sections* and *large pages*, which occupy the space of 16 regular sections or small pages, respectively. Hence, the MMU requires us to repeat the corresponding entries in the page directory (for super sections) or the page table (for large pages). Fig. 9 illustrates this situation.

Originally, these mapping entries were replicated on the abstract specification as well. The replicated actual entries for one logical entry, however, posed a contradiction in the way we phrased our earlier requirement that there should be a capability with the correct mapping information because there are technically 16 mapping paths, although logically regarded as one.

There are two alternatives to work around the replicated page entries; either we remove the replication in the abstract specification or we weaken the invariant to make an exception for replicated entries. We considered the former to be much cleaner; hence, we kept the stronger invariant and removed the replicated entries.

Choosing this approach, we were able to abandon a predicate about correctly replicated entries in the abstract specification, which had originally been required as part of the invariant. Instead, we have strengthened the state relation between the specifications, on the one hand, by relating the replicated entries at the low-level to the originals on the abstract specification layer, and the invariant of the low-level specification, on the other hand, by stating that entries for super sections and large pages are correctly replicated.

The work related to removing replicated entries alone accounted for about a quarter of our proof effort for Lemma 6 (i. e. a third of proving that page objects are backed by capabilities). This fact raises the question of whether treating replicated entries as a special case and weakening our requirements for them would have required less effort. Although this is possible, it is by far not obvious because introducing a special case for replicated entries would make the formulation of our requirements more complicated and consequently harder to work with. For similar reasons as avoiding a projection Π_A in Section 3, we chose to remove replicated entries, thus improving readability and clarity of the abstract specification and simplifying further proofs about it.

Since we now know that seL4's memory assignment works correctly (Lemma 6), the proof of invariant preservation (Lemma 5) for our new $\mathsf{user\text{-}trans}_\mathsf{A}$ is straightforward. In the next subsection, we discuss the corresponding proof for our new $\mathsf{user\text{-}trans}_\mathsf{H}$ at the low-level specification.

4.4. Strengthening the Invariant of the Low-Level Specification

When proving invariant preservation for our new user transition $\mathsf{user\text{-}trans}_\mathsf{H}$ at the low-level specification $\mathcal{M}_\mathsf{H}$, we have in principle the same problem as with the abstract specification $\mathcal{M}_\mathsf{A}$ above (Lemma 5): the original invariant I_H is not preserved by our new user transition $\mathsf{user\text{-}trans}_\mathsf{H}$. We need a stronger invariant and the additional proof requirement looks very similar to Lemma 6.

Note that the predicate I_H is deliberately weak and mainly comprises statements about encoding details of the low-level specification states that are not present in the abstract specification any more. We would like to keep it that way. Recall, however, that we have added quite complex predicates to the invariant I_A of the abstract specification. It would be tedious to add equivalent predicates on low-level specification states and repeat the proof effort on this layer. Instead, we would like to inherit the necessary properties from the previous proof because, assuming refinement, any invariant proven on the abstract specification also holds on the low-level specification.

Unfortunately, this reasoning is circular: we intend to use invariant preservation for establishing data refinement from $\mathcal{M}_\mathsf{A}$ to $\mathcal{M}_\mathsf{H}$; thus assuming this refinement to establish invariant preservation is pointless. Fortunately, the refinement calculus we use gives us more fine-grained control, such that we can separate our concerns. More specifically, we do not change the definition of I_H but perceive it as only the part of the invariant that is specific to low-level states. For the proof of correspondence between $\mathsf{user\text{-}trans}_\mathsf{A}$ and $\mathsf{user\text{-}trans}_\mathsf{H}$, it is sufficient to assume the strengthened I_A invariant and the original I_H predicate.

For the actual invariance proof about user transitions $\mathsf{user\text{-}trans}_\mathsf{H}$ of the low-level specification, however, we need a stronger invariant. Recall that we have defined the virtual MMU on the abstract layer and use the projection Π_H to convert them down to the low-level specification states. Drawing from this observation, we formulate the new invariant about the low-level specification based on the assumption that there exists a corresponding, abstract state and the abstract invariant holds, i. e. the actual invariant becomes $\mathsf{I}_\mathsf{H}\ s_\mathsf{H} \wedge (\exists s_\mathsf{A}.\ (s_\mathsf{A}, s_\mathsf{H}) \in \mathsf{SR}_\mathsf{A} \wedge \mathsf{I}_\mathsf{A}\ s_\mathsf{A})$. Apart from this more elaborate term, we state invariant preservation of our revised $\mathsf{user\text{-}trans}_\mathsf{H}$ analoguous to $\mathsf{user\text{-}trans}_\mathsf{A}$ (cf. Lemma 5). Formally, we state (the additions w. r. t. L4.verified's original statement are highlighted):

Lemma 7. The new user transitions of the low-level specification preserve the invariants:

$$\{\!|\lambda s_\mathsf{H}.\ \mathsf{I}_\mathsf{H}\ s_\mathsf{H} \wedge (\exists s_\mathsf{A}.\ (s_\mathsf{A}, s_\mathsf{H}) \in \mathsf{SR}_\mathsf{A} \wedge \mathsf{I}_\mathsf{A}\ s_\mathsf{A}) \wedge \mathsf{ut\text{-}pre}_\mathsf{H}\ s_\mathsf{H}|\!\}$$
$$\mathsf{user\text{-}trans}_\mathsf{H}\ \mathit{user\text{-}op}\ \mathit{tc}$$
$$\{\!|\lambda r\ s_\mathsf{H}.\ \mathsf{I}_\mathsf{H}\ s_\mathsf{H} \wedge (\exists s_\mathsf{A}.\ (s_\mathsf{A}, s_\mathsf{H}) \in \mathsf{SR}_\mathsf{A} \wedge \mathsf{I}_\mathsf{A}\ s_\mathsf{A})|\!\}$$

Proof. With a correlated, abstract state s_A for which the invariant I_A holds, we can eventually use Lemma 6 to prove the new proof obligation that arises from the changed $\mathsf{user\text{-}trans}_\mathsf{H}$. Recall that we use the projection Π_H to extend the virtual MMU functions $\mathsf{vm\text{-}perms}$ and $\mathsf{virt\text{-}to\text{-}phys}$ to states s_H of the low-level specification. We use Lemma 3 to establish that this projection of the low-level state, $\Pi_\mathsf{H}\ s_\mathsf{H}$, is equal to the correlated abstract state s_A. Furthermore, the state relation implies that the domains of the user-memory projections at both specification layers $\mathcal{M}_\mathsf{A}$ and $\mathcal{M}_\mathsf{H}$ are equal.

Given that we now state the existence of an abstract state to be invariant, we are required to prove it. This fact follows from the correspondence between the user transitions (cf. Lemma 8 and Section 4.5). This correspondence relates the abstract states in pre and post condition.

As already mentioned in Section 4.3, we have furthermore strengthened the original predicate I_H w. r. t.

correctly replicated entries for super sections and large pages. This change became necessary after we had removed the predicate about correctly replicated entries in the abstract invariant and had changed the state relation between the specification layers. □

At the first sight of Lemma 7, it might appear cumbersome to just assume the existence of an abstract state s_A correlating with the pre state s_H. In practice, this assumption is harmless because it already is a proof obligation for forward simulation between the specification layers $\mathcal{M}_\mathsf{A}$ and $\mathcal{M}_\mathsf{H}$. Hence, adding this assumption to Lemma 7 has no consequences for the forward-simulation proof. Note that we could also rephrase the above invariant by instantiating the existential quantifier with $\Pi_\mathsf{H}\ s_\mathsf{H}$:

Proposition.

$$\mathsf{I_H}\ s_\mathsf{H} \wedge (\exists s_\mathsf{A}.\ (s_\mathsf{A}, s_\mathsf{H}) \in \mathsf{SR_A} \wedge \mathsf{I_A}\ s_\mathsf{A}) \longleftrightarrow \mathsf{I_H}\ s_\mathsf{H} \wedge (\Pi_\mathsf{H}\ s_\mathsf{H}, s_\mathsf{H}) \in \mathsf{SR_A} \wedge \mathsf{I_A}(\Pi_\mathsf{H}\ s_\mathsf{H})$$

Proof. The above statement directly follows from Lemma 3. □

An apparent consequence of the added existential quantifier (or equivalently the Π_H projection) is that the refinement proof between low-level specification $\mathcal{M}_\mathsf{H}$ and `Simpl` representation $\mathcal{M}_\mathsf{C}$ is no longer independent from the refinement between the specifications $\mathcal{M}_\mathsf{A}$ and $\mathcal{M}_\mathsf{H}$. In practice, this circumstance is harmless as well because ultimately, we are interested in the fact that $\mathcal{M}_\mathsf{C}$ refines $\mathcal{M}_\mathsf{A}$, which requires both proofs.

While we lose the independence of the two refinement steps, we save much verification effort by assuming the existence of an abstract state. Recall that the alternative is an uninspiring repetition: an extensive proof at the low-level specification $\mathcal{M}_\mathsf{H}$, which mainly repeats the line of argument from invariant preservation at the abstract specification $\mathcal{M}_\mathsf{A}$. Most notably, we would not only repeat the effort for Lemma 6. In addition, we would have to show many properties of invariance at $\mathcal{M}_\mathsf{H}$, which have originally only been shown at $\mathcal{M}_\mathsf{A}$. The original refinement proof $\mathcal{R}_\mathsf{A}$ between $\mathcal{M}_\mathsf{A}$ and $\mathcal{M}_\mathsf{H}$ required five person-years [CKS08], to a large extent for invariant preservation at $\mathcal{M}_\mathsf{A}$. Consequently, the overall proof effort for establishing a self-contained invariant at $\mathcal{M}_\mathsf{H}$ is likely to comprise two person-years or more. Thanks to the flexibility of the refinement calculus we use, we can avoid this extensive verification effort.

Summarising this subsection, we have established invariant preservation for our revised $\mathsf{user\text{-}trans_H}$. Thus, there remains only one problem to be solved for reestablishing data refinement with L4.verified's proof architecture: showing correspondence of the revised user transitions. We discuss this problem below.

4.5. Proving Correspondence of User Transitions

By revising the user transitions $\mathsf{user\text{-}trans_A}$, $\mathsf{user\text{-}trans_H}$ and $\mathsf{user\text{-}trans_C}$, we have invalidated L4.verified's original correspondence lemmas (cf. Def. 4 and Lemma 1). In this section, we prove them anew. Syntactically, Lemma 1 remains almost unchanged (we only add *user-op* parameters):

Lemma 8. The new user transitions at the two specification layers, $\mathsf{user\text{-}trans_A}$ and $\mathsf{user\text{-}trans_H}$, respectively, correspond w. r. t. the state relation $\mathsf{SR_A}$ and equality of return values:

$$\begin{aligned}&\mathsf{corres}\ \mathsf{SR_A}\ \{(r_a, r_c).\ r_a = r_c\}\\ &\quad(\lambda s.\ \mathsf{I_A}\ s \wedge \mathsf{ut\text{-}pre_A}\ s)\ (\lambda s.\ \mathsf{I_H}\ s \wedge \mathsf{ut\text{-}pre_H}\ s)\\ &\quad(\mathsf{user\text{-}trans_A}\ \mathit{user\text{-}op}\ tc)\ (\mathsf{user\text{-}trans_H}\ \mathit{user\text{-}op}\ tc)\end{aligned}$$

Despite the syntactic similarity, the original proof is effectively void because the definitions of the user transitions have changed in structure. Fortunately, reproving the lemma is straightforward: we unfold the definitions and show correspondence for each computational step. For the added steps, correspondence follows from Lemma 3.

The low-level counterpart of Lemma 8 in contrast, the correspondence between the revised user transitions $\mathsf{user\text{-}trans_H}$ and $\mathsf{user\text{-}trans_C}$, hinges on the fact that the modified virtual memory is a subset of the user memory—just like the invariance proof for $\mathsf{user\text{-}trans_H}$ (Lemma 7). Hence, we assume here as well the existence of a correlated, abstract state s_A for which the invariant $\mathsf{I_A}$ holds:

Lemma 9. The new user transitions on the low-level specification and on the `Simpl` representation corre-

spond w. r. t. the state relation $\mathsf{SR_C}$ and with equality of return values:

$$\begin{array}{l}\mathsf{corres}\ \mathsf{SR_C}\ \{(r_a, r_c).\ r_a = r_c\}\\ \quad (\lambda s_\mathsf{H}.\ \mathsf{I_H}\ s_\mathsf{H} \wedge\ (\exists s_\mathsf{A}.\ (s_\mathsf{A}, s_\mathsf{H}) \in \mathsf{SR_A} \wedge\ \mathsf{I_A}\ s_\mathsf{A}))\ (\lambda s_\mathsf{C}.\ \mathsf{True})\\ \quad (\mathsf{user\text{-}trans_H}\ \mathit{user\text{-}op}\ tc)\ (\mathsf{user\text{-}trans_C}\ \mathit{user\text{-}op}\ tc)\end{array}$$

Proof. In principle, this proof is straightforward and much like the one for Lemma 8. A more interesting step, though, is showing that the user-memory updates at both layers correspond. Recall that at the `Simpl` layer, we have a single kernel heap for both, kernel objects and user memory, while on the specification layers, the kernel objects and the physical memory of the underlying machine are separated. Hence, a user-memory update could theoretically interfere with kernel objects in `Simpl` but not in the low-level specification. We rule this possibility out by showing the equality of the user-memory domains, drawing on the invariant about the correlated, abstract state, which we transfer down using Lemmas 3 and 4. □

This proof concludes our effort in reestablishing data refinement for seL4 after revising user transitions w. r. t. virtual memory. In the following, we look at related work and then conclude.

5. Related Work

The overall context of our work is operating-system verification. We are indebted to Klein [Kle09] for a comprehensive overview of past verification attempts. Two more recent results are more closely related to ours: the completed refinement proofs for the small real-time operating system Olos [DSS09], and for the simple virtualisation platform *baby hypervisor* [AHPP10]. Despite conceptual differences, both kernels are, like seL4, implemented in a mix of C and assembly based on an event-driven programming model with a single kernel stack.

Olos comprises 300 lines of code [Sch11]. Using Isabelle/HOL, Schmidt has verified its correctness in 2 person-years. Thus, Olos is small compared to the 8,700 C lines of seL4 and a proof effort of about 11 person-years. Schmidt's proof is based on a very detailed computational model including interaction with and computation of peripheral devices as well as an assembly semantics for user transitions. Though very detailed, her model of computation is less flexible than ours and assumes that all software in the system should eventually be verified [DSS10]. We, however, envisage larger trustworthy systems, where only a small trusted computing base is formally verified. In contrast to the step-wise refinement of L4.verified, the C code of Olos has been verified in a single refinement step. In addition, In der Rieden and Tsyban [dRT08] have proven the assembly parts correct but on a lower layer, without formally linking both proofs. The top-level statement of Olos correctness rests on simulation with a projection function from implementation to specification states, similar to our $\mathsf{Fin}\ \mathcal{M}_\mathsf{C}$. Although there is no formal notion of an observable state for Olos, it could be defined as the whole specification state. Processes are modelled as separate assembly machines, which can invoke the kernel and can be manipulated by the kernel. Thus, process separation is built into the implementation model. Process separation relies on another proof [dRT08, APST10]; although on a lower abstraction layer and not formally connected to the first. Furthermore, their hardware platform [BJK+06] is much simpler (e. g. single-level page tables without memory sharing).

The size of the *baby hypervisor* is given as "2.5k C code tokens", which is the same order of magnitude as Olos. Correctness has been proven using the *Verifying C Compiler* (VCC) [CDH+09], which is based on first-order logic and an automated verification backend. The authors specify the proof size with "7.7k annotation tokens" but refrain from estimating the manual work involved. An interesting aspect of their work is that they have instrumented VCC, which is tailored for C code verification, to prove mixed-language system software and formally specify refinement. A drawback of this approach is, however, that the assertion language is very close to C, which makes data refinement hard to express. The top-level statement expresses simulation as an infinite loop with 20 lines of C code. Informally, the statement says that the hypervisor implementation emulates the execution of the base architecture, which is specified by the implementation of a simulator for that processor. In other words, user transitions are modelled based on an assembly semantics with virtual memory; however, they deploy the same hardware platform as Olos, which is much simpler than ours. This is an appropriate specification for a pure hypervisor but certainly not for a microkernel like seL4, where the majority of the functionality is provided by its API.

Further, there is research related to our particular concern with virtual memory. Vaynberg and Shao [VS12] have verified a small virtual memory manager for simplified hardware in the interactive theorem

prover Coq. They advocate a modular approach to kernel verification, assuming a strictly modular software architecture. Although modularity is a necessity in commodity kernels with millions of lines of source code, it is atypical in microkernel construction, where the whole kernel compares in size and functionality to one of a commodity kernel's modules and where the conflicting goal of maximum performance is essential. Especially in this respect, it is unfortunate that Vaynberg and Shao chose simplified hardware, making a meaningful performance comparison of seL4 and their CertiKOS impossible.

Another direction of research is complimentary to ours. Barthe *et al.* [BBCL11] have formalised an abstract model of a hypervisor in Coq. Based on this model, they have formally verified properties of isolation and availability. Their model of virtual memory is more limited than ours; for instance, a page requires a unique owner in their system while in seL4, a page can be mapped into several address spaces. In principle, it should be possible to show refinement from their model to our abstract specification.

Similarly, Kolanski and Klein's mapped separation logic [KK09, Kol11] could be connected to our work. The mapped separation logic provides a unified view of virtual and physical memory for kernel verification. Our view of virtual memory exported to the user level should coincide nicely with the abstract interface that Kolanski and Klein assume in their work. Consequently, we may reuse this logic in future work for user-level virtual memory to cope with aliasing on page level.

6. Conclusion

Summarising, we have strengthened the original data refinement statement about seL4. Our technical achievements are threefold:

First, we have exploited the full potential of the original L4.verified proof by extending the observable state to the full abstract state (cf. Section 3). The original statements of correspondence (cf. Section 2.2), which have been proven in L4.verified as auxiliary lemmas for top-level statement of data refinement, are much stronger than the comparatively weak original top-level statement. Our extension of the observable state fills this previous gap, exploiting the full potential of the auxiliary corres lemmas and reflecting their strength in the top-level statement.

Second, informal expectations for correctness of an operating system kernel go far beyond a formal statement of data refinement. An important criterion for kernel correctness is that processes can be isolated. Our new model of user transitions (cf. Section 4) prepares the ground for a non-interference theorem by introducing the notion of virtual memory based on the kernel data structures, by distinguishing individual processes and by restricting the memory accesses of each process to its own virtual memory. Again, this change strengthens the original data refinement statement.

Third, our model of user transitions is customisable, thus facilitating proofs about a componentised software architecture with few trusted components in the midst of a largly untrusted code base. Most notably, the parameter *user-op* allows for different models of user transitions on a per process basis. Despite its simplicity, we consider this an important benefit over similar verification efforts that require all software in the system to eventually be verified, which will not scale to typically deployed code bases with several millions of lines.

The latter two achievements prepare future work. Most notably, our work has been used in a larger verification effort around information flow [MMB+12, MMB+13], which demonstrates that we have met our objective, to prepare the ground for a non-interference theorem, which naturally needs to distinguish individual processes. Previously, Andronick *et al.* [AGE10] and Klein [Kle10] laid out a roadmap to a software system with proven safety guarantees and currently, ongoing work complements general verification results (our work and the information-flow theorem) with system-specific proofs about small trusted components.

A different direction of future work is the validation of our model with respect to a detailed architectural model. Currently, our work models virtual memory at the level of the C source code, simply assuming that our model is correct. The underlying problem is the general tension encountered in low-level programs close to hardware: while written in a high-level language like C, they depend on low-level properties of the specifically targeted hardware. Consequently, even a verified compiler proven to adhere to the ANSI C semantics might break the low-level assumptions. Thus, Myreen and Sewell [SMK13] verify that the compiled binary indeed complies with the source code and its assumptions. Adding the MMU to their hardware model—which is validated against real processors—would enable us to link our MMU model to binary level and thus provide even stronger confidence.

Most publications in the context of L4.verified have focussed on the refinement technology [CKS08,

WKS+09], high-level reflections on the verification artefact [KEH+09,KAE+10] and more abstract, high-level properties established using the refinement statement [SWG+11,MMB+13]. This article, in contrast, aims at complementing these publications by a more in-depth elaboration on the technical details that verification engineers encounter in their day-to-day work when verifying an operating system kernel. The characteristic of these problems is that they are quite specific to the application domain. Perhaps the best example is the tight interdependency between the different parts of invariants—efficient, low-level programs are, for performance reasons, not written modularly and usually rely on complex invariants. We have presented examples from our proof at a conceptual level and expect that similar problems are likely to arise with similar verification projects in this application domain.

Apart from the particular results related to seL4, we have shown that the refinement calculus we use supports the transfer of invariants from an abstract to a concrete layer, which are themselves required for the refinement proof (cf. Section 4.4). Furthermore, we substantiate earlier claims [KEH+09, §5.3] regarding the cost of change for a large, verified code base: *Local changes*—like changing the user transitions in itself (the corres proofs were almost trivial) or the added ghost code to record object sizes—cause only little cost. *Strengthening Invariants* is significantly more expensive. In fact, the effort reported on in this paper was almost entirely spent in proving invariants. We estimate the overall effort with about 1.5 person-years, split almost evenly between invariant proofs for the extended observable state and for Lemma 6. Note that several of our decisions were guided by the long-term vision of proven trustworthiness. It might be possible to achieve the immediate results presented herein faster by sacrificing our longer-term goal of a clear and readable specification with strong, proven invariants that aid further proofs.

Acknowledgment

We are most indebted to Xin Gao for his contributions to the proof. Furthermore, we thank David Greenaway and Peter Höfner for their thorough internal reviews as well as June Andronick, David Cock and Toby Murray for comments on drafts of this article.

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.

This work was in part supported by AOARD grant FA2386-10-1-4105.

References

[AGE10] June Andronick, David Greenaway, and Kevin Elphinstone. Towards proving security in the presence of large untrusted components. In Gerwin Klein, Ralf Huuck, and Bastian Schlich, editors, *5th SSV*, Vancouver, Canada, Oct 2010. USENIX.

[AHPP10] Eyad Alkassar, Mark Hillebrand, Wolfgang Paul, and Elena Petrova. Automated verification of a small hypervisor. In Gary Leavens, Peter O'Hearn, and Sriram Rajamani, editors, *VSTTE 2010*, volume 6217 of *LNCS*, pages 40–54. Springer, 2010.

[APST10] Eyad Alkassar, Wolfgang Paul, Artem Starostin, and Alexandra Tsyban. Pervasive verification of an OS microkernel: Inline assembly, memory consumption, concurrent devices. In Peter O'Hearn, Gary T. Leavens, and Sriram Rajamani, editors, *VSTTE 2010*, volume 6217 of *LNCS*, pages 71–85, Edinburgh, UK, Aug 2010. Springer.

[BBCL11] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Formally verifying isolation and availability in an idealized model of virtualization. In Michael Butler and Wolfram Schulte, editors, *17th FM*, volume 6664 of *LNCS*, pages 231–245. Springer, 2011.

[Bil12] Nelson Billing. Formal functional specification of a security critical component in Isabelle/HOL. Bachelor thesis, University NSW, School Comp. Sci. & Engin., Apr 2012.

[BJK+06] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Putting it all together—formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(4):411–430, 2006.

[CDH+09] 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 Stefan Berghofer, Tobias Nipkow, Christian Urban, and Markus Wenzel, editors, *22nd TPHOLs*, volume 5674 of *LNCS*, pages 23–42, Munich, Germany, 2009. Springer.

[CKS08] 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, Montreal, Canada, Aug 2008. Springer.

[dRE98] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Number 47. United Kingdom, 1998.

[dRT08] Tom In der Rieden and Alexandra Tsyban. CVM — A verified framework for microkernel programmers. In *3rd SSV*, ENTCS, pages 137–153, Sydney, Australia, Feb 2008. Elsevier.

[DSS09] Matthias Daum, Norbert W. Schirmer, and Mareike Schmidt. Implementation correctness of a real-time operating system. In *IEEE Int. Conf. Softw. Engin. & Formal Methods*, pages 23–32, Hanoi, Vietnam, 2009. IEEE Comp. Soc.

[DSS10] Matthias Daum, Norbert W. Schirmer, and Mareike Schmidt. From operating-system correctness to pervasively verified applications. In *IFM*, volume 6396 of *LNCS*, pages 105–120, Nancy, France, 2010. Springer.

[KAE+10] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an operating system kernel. *CACM*, 53(6):107–115, Jun 2010.

[KEH+09] 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 *SOSP*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[KK09] Rafal Kolanski and Gerwin Klein. Types, maps and separation logic. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *22nd TPHOLs*, volume 5674 of *LNCS*, pages 276–292, Munich, Germany, Aug 2009. Springer.

[Kle09] Gerwin Klein. Operating system verification — an overview. *Sādhanā*, 34(1):27–69, Feb 2009.

[Kle10] Gerwin Klein. From a verified kernel towards verified systems. In Kazunori Ueda, editor, *8th APLAS*, volume 6461 of *LNCS*, pages 21–33, Shanghai, China, Nov 2010. Springer.

[Kol11] Rafal Kolanski. *Verification of Programs in Virtual Memory Using Separation Logic*. PhD thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Jul 2011. Available from publications page at `http://ssrg.nicta.com.au/`.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *15th SOSP*, pages 237–250, Copper Mountain, CO, USA, Dec 1995.

[MMB+12] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, and Gerwin Klein. Noninterference for operating system kernels. In Chris Hawblitzel and Dale Miller, editor, *The Second International Conference on Certified Programs and Proofs*, pages 126–142, Kyoto, Dec 2012. Springer.

[MMB+13] Toby Murray, Daniel Matichuk, Matthew Brassil, Peter Gammie, Timothy Bourke, Sean Seefried, Corey Lewis, Xin Gao, and Gerwin Klein. seL4: from general purpose to a proof of information flow enforcement. In *IEEE Symp. Security & Privacy*, pages 415–429, San Francisco, CA, May 2013.

[Sch05] Norbert Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452 of *LNAI*, pages 398–414. Springer, 2005.

[Sch06] Norbert Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.

[Sch11] Mareike Schmidt. *Formal Verification of a Small Real-Time Operating System*. PhD thesis, Saarland University, Saarbrücken, Apr 2011.

[SMK13] Thomas Sewell, Magnus Myreen, and Gerwin Klein. Translation validation for a verified OS kernel. In *PLDI*, pages 471–481, Seattle, Washington, USA, Jun 2013. ACM.

[SWG+11] Thomas Sewell, Simon Winwood, Peter Gammie, Toby Murray, June Andronick, and Gerwin Klein. seL4 enforces integrity. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, *2nd ITP*, volume 6898 of *LNCS*, pages 325–340, Nijmegen, The Netherlands, Aug 2011. Springer.

[TK05] Harvey Tuch and Gerwin Klein. A unified memory model for pointers. In *12th Int. Conf. Logic for Progr., Artificial Intelligence & Reasoning*, pages 474–488, Montego Bay, Jamaica, Dec 2005.

[Tuc08] Harvey Tuch. *Formal Memory Models for Verifying C Systems Code*. PhD thesis, School Comp. Sci. & Engin., University NSW, Sydney 2052, Australia, Aug 2008.

[VS12] Alexander Vaynberg and Zhong Shao. Compositional verification of a baby virtual memory manager. In Chris Hawblitzel and Dale Miller, editors, *CPP*, volume 7679 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2012.

[WKS+09] 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, *22nd TPHOLs*, volume 5674 of *LNCS*, pages 500–515, Munich, Germany, Aug 2009. Springer.