Formal Modelling of Separation Kernel Components

Andrius Velykis and Leo Freitas

University of York, UK
andrius@velykis.lt, leo@cs.york.ac.uk

Abstract. Separation kernels are key components in embedded applications. Their small size and widespread use in high-integrity environments make them good targets for formal modelling and verification. We summarise results from the mechanisation of a separation kernel scheduler using the **Z/Eves** theorem prover. We concentrate on key data structures to model scheduler operations. The results are part of an experiment in a Grand Challenge in software verification, as part of a pilot project in verified OS kernels. The project aims at creating a mechanised formal model of kernel components that gets refined to code. This provides a set of reusable components, proof strategies, and general lemmas. Important findings about properties and requirements are also discussed.

Keywords: Kernel, grand challenge, formal models, proof.

1 Introduction

Although software is ubiquitous, it is still perceived as an "Achilles' heel" of most systems, often being a serious threat. There is increasing evidence of the successful use of formal methods for software development. In [22], 62 industrial projects over 20 years are discussed. The survey explains the effect that formal methods have on time, cost, and quality of systems and how their application is becoming cost effective, hence easier to justify, not as an academic pursuit or legal requirement, but as a business case. By the use of mathematical analysis, formal methods enable accurate definition of a problem domain with capability of proving properties of interest. Formal methods application usually produces reliable evidence for errors that are fiendishly difficult to catch. Industrial and academic researchers have joined up in an international Grand Challenge (GC) in Verified Software [21], with the creation of a Verified Software Repository (VSR) with two principal aims: (i) construction of verified software components; and (ii) industrial-scale verification experiments to drive future research in the development of theory and tool support [2].

This paper is part of a pilot project within the GC in modelling OS kernels. It summarises work done in [19]. The project follows work on modelling smartcards [12] and flash memory file stores [8]. Our objective is to provide proofs of the correctness of a formal specification and design of kernels for real-time

A. Cavalcanti *et al.* (Eds.): ICTAC 2010, LNCS 6255, pp. 230–244, 2010.

embedded systems. We start from Craig's formal models of OS kernels [6], and take into account separation kernel requirements set out by Rushby [15] and in the US National Security Agency's *Separation Kernel Protection Profile* [18].

We focus on formalisation of data structures needed by an abstract specification of a separation kernel scheduler, its main operations and algorithms. Our long experience with the Z notation [20] and one of its theorem provers (**Z/Eves** [16]) is well aligned with Craig's original model [6] also developed in Z, as well as existing resources in the VSR. Kernel development work is facilitated by reusing modelling concepts and proof tactics of mechanising simple kernel components (*e.g.,* basic types and the process table) from [10]. A complex separation kernel process table adds process separation, external identifiers and other structures to address architecture and security requirements. While the process queue is shared with minor differences between both kernels, the shedulers are architecturally different. The increased complexity and structural differences affect associated proofs — although some lemmas can be reused, in general the proofs are different in both kernels and thus both contribute different verified components to VSR. With mechanisation and formal modelling we upgrade Craig's original separation kernel model from [6] by: improving the specification adding missing invariants and new security properties; verifying API robustness and model correctness in general; *etc.* Note that all details of results presented in this paper, analysis, justification as well as formal specification and proof scripts, are available in [19].

We briefly set the kernel verification scene in the next section. Section 3 presents a case study, which involves mechanisation of key data structures for the kernel's scheduler: a process table that keeps track of user and device process information; a process queue used by scheduler operations and algorithms; the scheduler invariant itself; and proved properties of interest. In Section 4, we reflect on our results by giving some measures. Finally, Section 5 sums up our findings and sets the agenda for future work.

2 Background

Craig's book on kernels [6] includes Z specifications and refinement of simple and separation kernels developed as an exercise that is beyond academic. It serves as a starting point for our project. The objectives are to demonstrate feasibility of top-down development using formal specification and verification with refinement to code (*i.e.,* correctness by construction) [11]. Craig's original models are typeset by hand and include several manual proofs. We augment the specification that uses Z notation [20] by mechanising it with a theorem prover [16] in order to more precisely record its correctness arguments from hand-written proofs. Given Craig's expertise as a kernel developer, we try to keep as faithful as possible to his original designs, only changing it at places where identified mistakes have been made. All results [19], including models, lemmas, *etc.* are being curated in the VSR [2] at SourceForge (`vsr.sourceforge.net`).

Verification of OS Kernels. An OS kernel is key in coordinating all access to underlying hardware resources like processors, memory, and I/O devices. Application processes can access these resources via system calls and inter-process communication. Kernel development has a reputation for being a complex task for two prime reasons: (i) every system requires the kernel to provide correct functionality and good performance; and (ii) the kernel cannot make (direct) use of the abstractions it provides (*e.g.,* processes, semaphores, *etc.*).

Microkernels for embedded systems are a suitable target for formal verification due to their small size and controlled environment. Such verification is an industrial-scale exercise that is undertaken in a number of academic and commercial projects. We identify two different approaches to verification. One starts with an existing kernel (possibly code or concrete design) and verifies its properties bottom-up, *e.g.,* Microsoft's hypervisor — a separation kernel aiming at virtualisation of hardware [5]. The project has verified existing C and assembler code for the functional correctness of kernel memory models. Within the GC, there is a recently started pilot project on the verification of FreeRTOS open-source kernel [3] that involves scientists in the UK and India. Alternatively, a top-down approach starts with the formalisation of high-level requirements that then gets refined (as formally as possible) to code. This approach allows reasoning about kernel properties without being bound to an existing implementation. Its application can be seen in parts of the commercial project L4.verified [13], which formalises and verifies a high-performance general-purpose microkernel; and in the work on Xenon [14], a security hypervisor based on the Xen OS. The former uses Isabelle/HOL to specify, abstract and verify properties of a Haskell prototype of the kernel, whereas the latter is using Z and CSP to model the C code. With properties proved about such formal model, one can then apply refinement techniques to obtain concrete designs. Furthermore, abstract components facilitate development of new kernel structures, where their properties are proved without an implementation. Our project aims to create fully generic abstract kernel models and refine them to code with good levels of automation — this paper contains results from the beginning of this large scale effort.

Simple kernel components based on [6] have already been mechanised in [10,9]. There we found interesting issues, including missing and hidden invariants. Although Craig's models have great insight from an OS engineer in necessary underlying data types, a series of mistakes are introduced, both clerical and more substantial in design decisions. Craig's work also includes a C implementation for Intel's IA32 architecture that is carried out using data refinement and the Z refinement calculus [20]. This paper continues with separation kernel components, in particular the scheduler, as reported in [19].

Separation Kernel. Separation kernel architecture was first introduced by Rushby [15], where different kinds of processes are isolated to achieve desirable security properties. The US National Security Agency has produced a Separation Kernel Protection Profile (SKPP) under the Common Criteria certification framework to define requirements for separation kernels used in environments

that require high-robustness [18]. SKPP also includes the kernels' interaction with both hardware and firmware platforms, hence these components also need to be verified. In here, we assume them as *trusted entities* verified elsewhere. In our work, we follow the formal models by Craig [6], which are relatively close to the SKPP requirements, as extensively discussed in [19]. Craig assumes the kernel to be running on an *Intel's IA32/64* platform, and verbally states that memory partitioning and context switches are achieved by the underlying hardware. We need to specify this mathematically in order to support statements spanning the kernel and the hardware. The main concerns are to ensure separation of process address spaces: they must execute in isolated memory partitions; inter-process communication is only allowed via vetted communication channels; and so on. To achieve this, one needs to have memory partitioning in the kernel, where each process is allocated a dedicated area. Process communication is established via message passing over a special shared memory area. Unauthorised communication between processes is prevented by having external process identifiers, which are translated into internal representations within the kernel. The requirement to have process execution separation is achieved by a non-preemptive scheduler. It ensures only a single component is active at each given time. Craig models device processes as trusted code running within the kernel. Our work here is to mechanise, polish, and improve Craig's original model. Also, we know from collaboration that the modelling of the Xenon hypervisor [14], which is a much more complex kernel, is benefiting from ideas presented here and in [19].

3 Case Study

Separation kernel formal model development is a significant undertaking due to the high number of different components and requirements, as well as specific domain knowledge involved. This case study presents some details on mechanisation, modelling and verification of a separation kernel specification, based on hand-written models by Craig [6]. The mechanisation has four stages: (i) parsing and typechecking Z for syntactic type consistency; (ii) domain and axiomatic checking that shows well-formedness of expressions (*e.g.,* functions are applied within their domains), and soundness of axioms; (iii) feasibility lemmas providing an existence proof for the initial state, and operation preconditions showing API robustness and correct state invariant; and finally (iv) proving conjectures that represent properties of the model. We also summarise key data structures in the scheduler like a process table and scheduling queue. This paper focuses on formal modelling, hence details of the mechanical proof process are omitted here. Instead, all theorems, proofs and complete analysis are available in [19]. A very detailed report on proving Z specifications with **Z/Eves** is provided in [7].

3.1 Process Table

A core data structure in our separation kernel is a *process table* (*PTab*) that stores all process information. Previous work on simple kernels [10] has been

reused, with additional variables and invariants to address process separation security requirements. All kernel processes are referenced by their identifiers, as bounded non-empty range type of *PID*s. Also, to distinguish between user processes and "trusted" platform code (*i.e.,* device processes), we use Z free types ($PTYPE ::= uproc \mid dproc$), which in this case are enumerated type constructors that form a partition. This means that *uproc* and *dproc* are distinct, and the only elements of the set $PTYPE$. Outside the kernel, processes are referenced by external identifiers to prevent unauthorised access to internal kernel resources. Craig [6, Chap. 5] suggests having an unbounded number of external user process identifiers *UPID*, and a limited number of device identifiers *Dev*, since in embedded environments device configuration is known from the start.

PTab

$$used, free : \mathbb{F}\ PID;\ nup : UPID;\ dmap : Dev \rightarrowtail PID;\ pidext : PID \nrightarrow UPID$$
$$ptype : PID \nrightarrow PTYPE;\ extpid : UPID \nrightarrow PID;\ state : PID \nrightarrow PSTATE$$

$$free = PID \setminus used \land used = \mathrm{dom}\ state = \mathrm{dom}\ ptype \land pidext = extpid^{\sim}$$
$$\exists\, dprocs, uprocs : \mathbb{F}\ PID \bullet dprocs = ptype^{\sim}(\!|\ \{\ dproc\ \}\ |\!) = \mathrm{ran}\ dmap$$
$$\land\ uprocs = ptype^{\sim}(\!|\ \{\ uproc\ \}\ |\!) = \mathrm{ran}\ extpid$$
$$\forall\, u : UPID \mid u \geq nup \bullet u \notin \mathrm{dom}\ extpid$$

A process table (*PTab*) is specified using a Z *schema*: a labelled record data structure with invariants. *PTab* is similar to the one for a simple kernel in [10]: it has finite sets ($\mathbb{F}$) *used* and *free* for process identifiers (*PID*) that are disjoint; and it specifies partial function ($\nrightarrow$) mappings for each *used PID* to access various related process information. Functions *ptype* and *state* specify type and process-state information for each process, respectively. Available process states are defined by the free-type *PSTATE* like *PTYPE* above, and omitted here. Process table invariants require these mappings to exist for all *used* processes (*i.e.,* functions recording process information are total on *used*).

External identifiers are different for each process type and stored in separate structures. Device numbers are stored in *dmap*: a partial injective ($\rightarrowtail$) relationship, which guarantees a one-to-one mapping between device numbers *Dev* and kernel device process *PID*s. For user process identifiers, we have functions *extpid* and *pidext* as the inverse ($^{\sim}$) of each other to allow simple bi-directional identifier queries. Some ambiguity while modelling a system in Z is common practice, providing it aids clarity and simplify proof goals. Because *dmap*, *extpid* — and a few other process information mappings not included above — are used for processes of different types, just like with *used PID*s for *state* and *ptype* domains, we need to identify device and user-process sets. Since these sets are images of *ptype* for each process type, we define sets *dprocs* and *uprocs* locally using an existential quantifier for device and user processes, respectively. These sets are linked with the range of their corresponding functions — as well as the domain of the few mappings not shown here. Finally, *nup* defines next available *UPID* for user processes. Proving precondition of *PTab* operations revealed the necessity for ensuring a future unused *UPID* ($u \notin \mathrm{dom}\ extpid$) for all *UPID*s beyond (and

including) *nup*, hence the last invariant. More details on how this appeared are given below, and a full account is given in [19, Chap. 6].

Properties. *PTab* specifies an injective relationship between device process external (*Dev*) and internal (*PID*) identifiers. A corresponding property of user process identifiers (*UPID*) can be formulated as

$$PTab \vdash extpid \in UPID \rightarrowtail PID \land pidext \in PID \rightarrowtail UPID \tag{1}$$

Given a *PTab* state (*i.e.,* there exists an instance of the state where its invariant holds: we have a feasible model), both *extpid* and *pidext* are injective ($\rightarrowtail$), thus for each internal user process *PID*, there exists a unique external user process *UPID*, and vice-versa. We proved it as a theorem in **Z/Eves** using current *PTab*'s invariants. This way we achieve separation of concerns, as the theorem can be used later in proofs. As a redundant invariant in *PTab*, it could unnecessarily complicate future proofs, *e.g.,* extra proofs are needed per redundant property. We formalised various operations for process identifier management, such as process allocation and deletion.

Process Table Refinement. Mechanisation involves refinement of data structures to accommodate proofs and model changes. A detailed case study of *PTab* refinement is given in [19, Chap. 6] and describes the evolution of the original schema in [6, p. 220] to the final form as shown above. Refinement relationships are proved in [19] for each step to ensure that the original specification is still satisfied, *e.g.,* $PTab \Rightarrow PTabv4 \Leftrightarrow PTabv3 \Leftrightarrow \ldots$. Equivalence ($\Leftrightarrow$) changes represent specification refactoring without changing the original meaning, whereas a refinement $Spec \sqsubseteq Design$, here as reverse implication $Design \Rightarrow Spec$, guarantees that the stronger schema satisfies all properties of the one it refines, and is used to correct mistakes and add missing invariants. Some examples illustrating different refinement steps are given below (details and all proofs are in [19]).

Complicated mathematical constructs can be replaced with more suitable Z idioms for theorem proving without compromising corresponding *PTab* schema. Originally the set of device processes was defined as set comprehension (middle)

$$dprocs = \{\, p : PID \mid p \in used \land ptype(p) = dproc \,\} = ptype^{\sim}(\!|\, \{\, dproc \,\} \,|\!) \tag{2}$$

which is straightforward enough to understand: *dprocs* is a set of known (*used*) *PID*s with *dproc* type, as required. However, set comprehension expressions are difficult to reason about in proofs, as they require pointwise extensionality proofs (*i.e.,* to show that every element in the set satisfies its invariants). Instead, if we could characterise the same relationship with higher-level operators, we would then be likely to have higher automation levels. So, (2, middle) can be refactored as the relational image of the of inverse of *ptype* (2, right). Function *ptype* maps *PID*s to their process types (*PTYPE*). Inverting it, we get a relation (set of pairs) between *PTYPE* and their corresponding *PID*s, which might no longer be functional as more than one identifier might be of user or device type. Then,

we apply relational image ($R(\!|\ S\ |\!)$) over this resulting set of pairs, which gives a set of values in *ptype* for all its *dproc*-typed members. Thus, (2, right) gives all *PID*s of *dproc* type in *ptype* as set *dprocs*. The advantage of this equivalent formulation is that it can take advantage of a series of lemmas about inverse and relational image from the Z mathematical toolkit [17], which then leads to more automatic proofs. The equality in (2) enables us to prove equivalence between the schemas affected by this change (*i.e.,* $PTabv4 \Leftrightarrow PTabv3$ in [19, p. 46]).

Failed precondition proofs during the mechanisation of a deterministic *UPID* allocation algorithm by Craig [6, p. 222] revealed a missing invariant. The monotonic increment of new *UPID*s did not ensure that new identifiers are unused by the process table. We refine the $PTabv4$ schema (*i.e.,* $PTab \Rightarrow PTabv4$) by adding an invariant (3) requiring the next and subsequent *UPID*s to be available.

$$\forall\, u : UPID \mid u \geq nup \bullet u \notin \mathrm{dom}\ extpid \tag{3}$$

Without it, one could not prove that *UPID* allocation operations kept the after state of *extpid* values being injective. This is an expected, yet missing invariant both in the Z models and in the English-written requirements, rather than a cosmetic model change. We benefit from streamlined specification, corrected invariants and clarity of requirements in the refined *PTab* schema, while the proofs show schema conformity to the original specification. The full account on *PTab* refinement is given in [19, Chap. 6].

3.2 Process Queue

The separation kernel scheduler stores waiting processes in *process queues* to ensure correct execution order. We model them with operations to access and manage queue elements, such as querying for elements, enqueue and dequeue processes to be scheduled, and so on. The resources being queued are process identifiers (*PID*s) from the underlying process table (*PTab*). During these queue operations *PTab* components are kept read-only.

The next (horizontal) schema *PQ* defines the queue as an injective sequence of process identifiers: it is a function from ordered pairs of indexes started from one to unique *PID* values (*i.e.,* akin to $\mathbb{N}_1 \rightarrowtail PID$). Only elements from the given *PTab*'s *used* set are allowed in the queue, as indicated by the subset ($\subseteq$) constraint over the sequence's range of *PID*s queued. In Z, schema inclusion like *PTab* is used to factor complex data structures into its constituent components.

$$PQ \mathrel{\widehat=} [\, PTab;\ pr : \mathrm{iseq}\ PID \mid \mathrm{ran}\ pr \subseteq used \,] \quad PQInit \mathrel{\widehat=} [\, PQ' \mid pr' = \langle\rangle \,]$$

The queue is initialised as empty in *PQInit*. In Z, dashed variables like pr' represent operation's after state, where PQ' represents dashed components from *PQ*. Note that in *PQInit* the underlying *PTab* is not restricted during *PQ* initialisation, hence allowing *PTab* to be initialised before (*e.g.,* with a suitable *PTabInit*). Such need arises when *PQInit* is used in scheduler initialisation (see Sect. 3.3). Because we require only an existence proof for the feasibility of such initial state, this arrangement is just right (*e.g.,* $\exists\, PQ' \bullet PQInit$). We need two

separate process queues for user processes and devices. To avoid name clash, a DQ schema is defined by renaming the queue sequence variable pr to dv as

$$DQ \mathrel{\widehat=} PQ[dv/pr] \qquad DQInit \mathrel{\widehat=} PQInit[dv'/pr']$$

Notice that the underlying $PTab$ for both PQ and DQ remains the same. This neat alignment/reuse of components is a great feature of the Z schema calculus [20, Chap. 11]. Craig suggests an injective sequence [6, Sect. 3.4] in PQ to ensure that no process can be queued multiple times. Injective sequence updates are tricky because they require uniqueness of sequence range elements, yet sequence operators (*e.g.,* concatenation $\_ \frown \_$) are not aware of such restrictions. This requires additional effort during precondition proofs to guarantee PID queueing uniqueness. For instance, to concatenate an element x to a sequence t one just needs to say $t' = t \frown \langle x \rangle$, whereas for a injective sequence s, it is also necessary to show that x is unique in s (*i.e.,* $x \notin \operatorname{ran} s$) like

$$\forall s : \operatorname{iseq} X;\ x : X \bullet x \notin \operatorname{ran} s \Rightarrow s \frown \langle x \rangle \in \operatorname{iseq} X \tag{4}$$

which is proved in [10]. Given a non-empty generic type X and an injective sequence s, if an element $x \in X$ is not mapped in s, then appending x to s keeps the result injective. Lemmas like this have been reused across various pilot projects, and form the basis of a general library, which is one of the outcomes of the Grand Challenge: reusable components and proofs. Furthermore, process queue is a generic kernel data structure and is used within other components (*e.g.,* semaphores in simple kernel [6, Sect. 3.7]). Different kernel process tables mandate redoing PQ proofs, however, proof structures and toolkit lemmas are successfully reused, hence minimising implementation effort. Further reuse is facilitated by collecting such verified components in the VSR.

3.3 Scheduler

Like in [15], our separation kernel employs a round-robin scheduler. It is non-preemptive, hence a running process can only suspend voluntarily or terminate. Such approach is chosen for its simplicity and current use in embedded systems. Scheduler includes operations for making a process *ready*, *suspended*, or to *terminate* processes. Synchronous device I/O in the kernel is achieved by the scheduler executing device processes at a higher priority than user processes, so that when a device call is performed, a corresponding device process is executed to handle the call, while the user process suspends and waits for a reply.

Schema *Sched* contains two ready-queues for each process type: PQ and DQ. Two special (distinct) identifiers are used for the idle (ip) and current (cr) processes. The former is used when no other processes are scheduled, whereas the latter stores the process currently executing. We also define a component to aggregate all queued process identifiers in a single set ($queued$). This auxiliary term allows us to write simpler expressions, entailing easier proofs later.

$$
\begin{array}{|l}
\textit{Sched} \\
\hline
PQ;\ DQ;\ cr, ip : PID;\ queued : \mathbb{P}\ PID \\
\hline
queued = \operatorname{ran} pr \cup \operatorname{ran} dv \land \{ cr, ip \} \subseteq used \land cr \notin queued \land ip \notin queued \\
\operatorname{ran} pr \subseteq ptype^{\sim}(\!|\ \{ uproc \}\ |\!) \land \operatorname{ran} dv \subseteq ptype^{\sim}(\!|\ \{ dproc \}\ |\!) \\
\hline
\end{array}
$$

The state invariant has that the current and idle processes are known to the kernel process table (*i.e.,* within *used*), and must not be queued for execution — no process can be executing and waiting at the same time. We ensure that processes of different types are queued accordingly: *pr* for (*uproc*) user processes, and *dv* for (*dproc*) kernel device processes. We use subset containment over the relational image of each function.This specification of the scheduler is a substantial upgrade from Craig's [6, Sect. 5.6]. The addition of a reference to *PTab* via *PQ* enabled the specification of process properties. Thus, we were able to convert informal requirements given in English in the book: that *PQ* is for user processes and *DQ* is for devices, for instance. Also with the invariants on *cr*, *ip* being known (in *used*) *PID*s only, we could specify and prove kernel security properties, and define robust operations for the scheduler (*i.e.,* operations that are proved to account for all behaviours as successful and exceptional cases).

We initialise the scheduler with empty queues and an idle process running that is passed as ($p?$) an input variable. In Z, inputs are tagged with a question mark. We keep the modular approach for kernel components and reuse previously defined operations: during scheduler initialisation, queues and process table are initialised by corresponding operations like *PQInit*.

$$SchedInit \ \widehat{=}\ [\, Sched';\ PQInit;\ DQInit;\ p? : PID \mid ip' = p? \land cr' = ip' \,]$$

$$SchedPTabInit \ \widehat{=}\ PTabInit \fatsemi (AddIdleProcess \land SchedInit[ip!/p?]) \setminus (ip!, u!)$$

Using *PTab* operations defined elsewhere [19, Chap. 6], we can construct full initialisation of *PTab* and *Sched* as *SchedPTabInit*. We use schema composition for the sequential execution of operations. In Z, the schema composition ($S \fatsemi T$) operator uses the after state of S as the before state of T, with the after state of T being the overall after state, and similarly for the before state of S. That works well providing the after state components of S is the whole of the before state of T (*i.e.,* we have homogeneous composition). This models a process table that is initialised first by *PTabInit*, then an idle process is allocated by *AddIdleProcess* (*i.e.,* a user process allocation operation in [19, p. 57]), and passed into scheduler initialisation *SchedInit*. Finally, output variables *ip*! and *u*! are hidden (*i.e.,* existentially quantified) to avoid exposing them outside the operation. It neatly reuses definitions plumbed with the schema calculus, which are equivalent to the following expanded schema.

$$SchedPTabInitExpand \ \widehat{=}\ [\, Sched' \mid used' = \{ ip' \} \land nup' = 2 \land cr' = ip' \land pr' = dv' = dmap' = \emptyset \land extpid' = \{ 1 \mapsto ip' \} \land ptype' = \{ ip' \mapsto uproc \} \land .. \,]$$

It shows state initialisation after all updates take place. The idle process ip' is allocated with expected initial values and passed into the scheduler for execution.

3.4 Scheduler Operations

Abstract data type operations in Z are specified as a relation between before and after states. This includes both successful and exceptional cases. Typically this is given as a disjunction of schemas. Usually we negate the successful case precondition, where each error case accounts for some of the negated predicates, such that the overall precondition equates to *true*, hence leading to a robust interface. This is the so-called Oxford style of Z [20]. A common and simple solution for error handling is to report the error whilst keeping the state constant. Our separation kernel model uses an *errors with memory* [1, Sect. 18.3] approach to store the error message in between operations. This way a subsequent operation can check whether the kernel is in a valid state. Furthermore, Craig [6] defines error cases to kill the kernel via an interrupt as a simplistic, albeit blunt an approach for secure exit. Nevertheless, this interrupt handling is not modelled formally. To aid this we need to extend it by defining and proving kernel error handling security properties. Note that an interrupt-like error handling approach, with state validation, recovery and logging, has been successfully modelled in Mondex project [12]. Detailed description of kernel error handling is given in [19, Chap. 5], while here we present a short summary of key operations only.

For example, a process queueing operation *EnqPQOk* performs the sequence concatenation (queueing) in the successful case. The error cases, such as when the process is already queued (*ErrQueued*), are defined separately and then disjoined with the successful case. In Z, ΔPQ indicates both PQ and PQ' are included, yet without any constrains over state variables. ΞPQ is just like ΔPQ but requires everything in PQ to be kept constant.

$$
\begin{array}{lcl}
EnqPQOk & \widehat{=} & [\, \Delta PQ;\ \Xi PTab;\ p? : PID \mid pr' = pr \frown \langle p? \rangle \,] \\
ErrQueued & \widehat{=} & [\, \Xi PQ;\ \Delta ErrV;\ p? : PID \mid p? \in \mathrm{ran}\ pr \land serr' = errqueued \,] \\
EnqPQ & \widehat{=} & (EnqPQOk \land SysOk) \lor ErrQueued \lor \ldots
\end{array}
$$

A process is added for scheduling by "readying" it. This means it must be enqueued and its state in *PTab* must be set to *psready*. Such queueing must keep everything else constant (*i.e.,* queueing happens before scheduling). We reuse *EnqPQ* operation to perform the actual queueing in *EnqSchedOk*.

$$
\begin{array}{lcl}
EnqSchedOk & \widehat{=} & [\, \Delta Sched;\ EnqPQ \mid cr' = cr \land ip' = ip \land dv' = dv \,] \\
EnqUserSched & \widehat{=} & EnqSchedOk \lor ErrNotUserPID \lor \ldots
\end{array}
$$

Schema *EnqUserSched* models queueing a user process without updating the current, idle or device processes. The operation performs a number of checks to ensure that all invariants are satisfied. Included schema *EnqPQ* has a robust specification for queue-level queueing with: successful cases; case when a process is unknown to the kernel (*e.g., PID* not in *used*); or case when it is already queued (*e.g., PID* in *pr* range). Since no queue operation alters *PTab* ($\Xi PTab$), *EnqPQ* ensures that the underlying process table does not change as a result of *EnqSchedOk*. Scheduler invariants give rise to additional error cases: a process can have the wrong type; it might already be running; the idle process cannot

be queued; and so on. These error cases are defined to create a total enqueue operation *EnqUserSched*. The complete "readying" operation *MakeReady* enqueues and sets process state. We use schema composition ($\fatsemi$) to update the *pstate* function in *PTab* for a given process ($p? \in PID$) to *psready* in *SetReady*.

$$MakeReady \mathrel{\widehat=} EnqUserSched \fatsemi ((IsSysOk \land SetReady) \lor ErrKeepFail)$$

The input variable comes from *EnqPQ*. Note that process state changes only if the enqueue succeeds, hence we check for expected system state with *IsSysOk*. If *EnqUserSched* does fail (*i.e.,* either via *EnqPQ* errors, or by its own error cases), the error is propagated by *ErrKeepFail*. An analogous operation has been defined for device processes. In [4], these schemas are used as APIs for the definition of a parallel/distributed scheduling algorithm that considers the behavioural aspects of the specification, rather than what is happening in the data structures.

The main scheduler function is to determine and execute processes in an appropriate sequence. As mentioned, device processes have priority over user processes, while the idle process is run when nothing else is scheduled. We model this via separate operations, later on conjoined to represent the overall scheduling algorithm. All operations follow the same pattern: a process is selected for execution and its state is set via a *PTab* operation (*SetRunning*).

RunIdleNext

$$\Delta Sched;\ SetRunning[ip/p?];\ SysOk$$

$$dv = pr = \langle\rangle \land cr' = ip \land ip' = ip \land pr' = pr \land dv' = dv$$

Since the idle process is never queued in the scheduler, running it does not require updating scheduler queues, hence we can specify everything with operation *RunIdleNext*. Like *SetReady*, *SetRunning* updates *pstate* for a given $p?$ input to *psrunning*, which in this case is the idle process *ip*. This operation can only be executed when both process queues *dv* and *pr* are empty. The operation sets the idle process as current ($cr' = ip$), sets it to running, and does not change anything else. Finally, *SysOk* signals that operation has been successful. User and device scheduling operations are similar in that they both take the first element in a respective queue and change its state to running. For dequeue, corresponding operations for device and user processes queues are used.

$$SchedUserNext \mathrel{\widehat=} [\,\Delta Sched;\ DequeuePQ \mid dv = \langle\rangle \land pr \neq \langle\rangle \land cr' = p! \land ip' = ip \land dv' = dv\,]$$

$$RunUserNext \mathrel{\widehat=} (SchedUserNext[n/p!] \fatsemi SetRunning[n/p?]) \setminus (n)$$

Schema *SchedUserNext* sets process $p!$, which is output by *DequeuePQ* operation, as the current process. Note that *DequeuePQ* is responsible for updating user process queue pr'. The scheduling algorithm is specified in the operation invariants: output user process $p!$ is scheduled when device queue *dv* is empty, and user process queue *pr* is not. *RunUserNext* appends the *SetRunning* operation to the scheduling algorithm. Here auxiliary variable n is used to link the

output parameter from one operation to the input parameter of the next. We can formulate and prove statements about scheduling operation properties to improve assurance in them, *e.g.,* we show that *RunUserNext* always executes the first element in user process queue (*i.e., RunUserNext* $\vdash cr' = head\ pr$). The device scheduling operations *SchedDeviceNext* and *RunDeviceNext* are specified in an similar manner with *DequeueDQ* and with the precondition that the device queue is not empty ($dv \neq \langle\rangle$). The complete API for any scheduler state implements the scheduling algorithm by disjoining all top-level operations.

$$SchedNext \mathrel{\widehat=} RunIdleNext \lor RunUserNext \lor RunDeviceNext$$

We proved that the precondition of this operation is *true*, meaning that it can be executed for any scheduler state: it is a robust operation that will always succeed. Even more, we proved a conjecture to show that the operation never fails: it will always return the system call was okay (*e.g., SchedNext* $\vdash serr' = sysok$). The composing operation invariants ensure that the total operation *SchedNext* is never in a state where queueing error case preconditions may apply. Obviously, all that did not fall into place neatly. It was the result of a well-crafted model, followed by the ruthless scrutiny of the theorem prover, alongside appropriate guidance in the process via useful lemmas for acceptable levels of automation.

With the definitions of enqueue and scheduling available, we can create a suspend operation. A process is never preempted, but can suspend voluntarily to relinquish execution to other processes. One of the cases when this may happen is during inter-process communication: a process sends a message to another process and suspends itself to allow the other process to eventually execute and handle / reply the message. The suspend operation runs the next available process and places the caller back in the process queue. This is specified by reusing *SchedNext* and *MakeReady* operations.

$$RequeueUserProcess \mathrel{\widehat=} (SchedNext \fatsemi MakeReady)$$

Both *SchedNext* and *MakeReady* are robust (*i.e.,* precondition is *true*), and *SchedNext* constrains all variables in *Sched* referenced in *MakeReady* (*i.e.,* we have a homogeneous operation), hence we can safely combine them using schema composition. That is, since we have already shown *SchedNext* will always succeed, it is not necessary to check if the system is in a valid state before executing *MakeReady*. Nevertheless, as *MakeReady* can fail due to a queueing error, the full operation is not atomic: we could not recover the scheduler's state in case *MakeReady* fails. The need for atomicity of such operations is discussed in [4].

The scheduler reuses a large number of operations from *PTab* and *PQ*. The nested schemas and multiple error cases create complex operations that can be compactly presented using the schema calculus. Yet, expanding all definitions could lead to rather tricky proof steps, and a good amount of lemmas need to be in place if one is to complete the proofs with acceptable levels of automation. Fortunately, as it often happens [10,9], such lemmas are not only repeated, but also reusable from previous experiments, which minimises the overall effort.

4 Discussion

General outcomes of the separation kernel component mechanisation confirm the findings of the previous projects [10] — proper tool support and a verification framework build confidence in the formal specification. Syntax errors are eliminated, model feasibility and API robustness are verified, and missing invariants to guarantee correct operations are found. The formal model is fully proved mechanically — the proofs help establishing the correctness of the Z specification. The validity of the kernel model must be demonstrated by proving architectural and security properties. Our work significantly improves Craig's scheduler model. By translating verbal requirements to mathematical invariants and improving design of the specification, we are able to formulate and prove certain properties about the components (*e.g.,* the scheduler deadlock analysis below). The main separation kernel properties of process separation (*e.g.,* memory partitioning and communication via established channels only), however, span a number of kernel components, some of which have not been mechanised yet. Thus these properties must be analysed and proved as a future exercise.

Scheduler Deadlock Analysis. Using invariants in scheduler schema *Sched*, we prove that kernel starvation by queueing all known processes, hence none would be available for execution, is impossible: *i.e.,* $\forall\, Sched \bullet queued \subset used$ (Sect. 3.3). Nevertheless, a deadlock can occur, when, for example, the initial process (which creates other processes) suspends without "readying" other processes. In this case, the idle process would be running with all processes queued. Even so, this functionality is specific to the process, not the kernel, and we do not formally model the running processes themselves.

Other Separation Kernel Components. The process table, queue and scheduler are core data structures within a separation kernel. The next step is to mechanise and model other components, such as messaging or memory management, to achieve and verify full separation of processes. We have laid the foundations for that: external process identifiers can be allocated and translated in the process table to avoid exposure of kernel internal representation; process schedule and suspend functionalities are the core of the inter-process messaging subsystem. The formal model of underlying hardware platform would go beyond Craig's original model, but would allow proving memory access restrictions. Craig assumes hardware exceptions [6, p. 204], yet does not formally specify them. A formal model for interrupts would benefit from similar experience with Mondex smartcards [12].

Proofs & Benchmarks. In [19] we have already mechanised close to five separation kernel specification sections (out of 12 [6, Chap. 5]). The remaining part constitutes of memory management and messaging components, as well as kernel interface spanning all components. Our mechanisation of three core components

have a total of 263 paragraphs comprising schemas, types, and axioms, with ∼40% of these being related to operation feasibility proofs (*i.e.,* operation preconditions). These generated 254 verification conditions: ∼50% are about model feasibility; 35% are about proof automation rules in **Z/Eves**; and ∼10% are related to model refactoring. Also, we proved 12 properties of interest (∼5%), several of which have been presented in the case study (Sect. 3). Furthermore, the general theories created by other pilot projects contain well over 120 reusable theorems about various mathematical data types [9]. Separation kernel proofs were discharged with over 1300 proof commands, 23% of which require creative steps involving quantifier's witnesses, or knowledge on how the theorem prover works. The remaining 77% involved proof exploration steps of moderate difficulty and straightforward/blind tasks, given the right model. These numbers enable comparison with previous GC pilot projects, which accumulated similar information. The Z mechanisation of Mondex [12, p. 117–139] has 25% less paragraphs, yet has over 350% more proof: that is largely due to the underlying refinement calculation involved. Various automation lemmas from Mondex were reused. This improved our share of push-button proof steps to 40%, whereas they were about 27% in Mondex. This suggests that Mondex proofs were more difficult, yet the separation kernel proofs automation levels benefited from that work. Such difference is expected as we did not do any refinement proofs yet: it will be worthwhile comparing those later. The work presented here took approximately 900 man-hours, which also included the first author mastering the **Z/Eves** prover without previous experience, and writing up the thesis in [19].

5 Conclusions

The Grand Challenge's pilot projects inspire us to model and verify various application domains. One aim beyond actual mechanisation is to make it easier for the next team who want to work with kernels. For that, we provide data types and useful lemmas that are central to modelling kernel scheduling. Basic verified data structures were developed for a simple kernel [10], with a collection of general lemmas from the Verified Software Repository (VSR) being reused [9].

In this paper we improve the specification and formal model of the separation kernel in [6]. Separation kernel data structures address security requirements and are more complex than their simple kernel counterparts. Most of the work consisted of identifying properties about data types, calculating preconditions for (*i.e.,* feasibility of) each operation, and verifying everything via formal proof along the way. This mechanisation revised and improved the specification of the process scheduler and its associated components: it corrected modelling errors on data types, as well as missing error cases of operations, and mistaken invariants from [6], some of which were discussed here (and discussed in full in [19]). Together with new extracted general lemmas and detailed verification process report (all available in [19]), we believe this to be an important contribution in building theories for mechanised formal modelling of OS kernels.

Future Work. We aim to complete the formal kernel model and prove the process separation, as well as to perform formal refinement of kernel components to a concrete model and implementation code. We intend to explore how such kernel model is used in other projects [14], and different scheduling algorithms [4].

Acknowledgements. We are grateful to Iain Craig for his useful account of formal kernel specification and modelling. The first author is now employed by the EPSRC (EP/H024050/1) AI4FM project at Newcastle University, UK.

References

1. Barden, R., et al.: Z in Practice. Prentice Hall (1994)
2. Bicarregui, J., et al.: The verified software repository. Formal Aspects of Computing **18**(2) (2006) 143–151
3. Berry, R.: A free real-time operating system (FreeRTOS)
4. Boerger, E.: Refinement of distributed agents. In: Dagstuhl Seminar 09381. (2009)
5. Cohen, E., et al.: VCC: A practical system for verifying concurrent C. In: Theorem Proving in Higher Order Logics. Volume 5674 of LNCS., Springer (2009) 23–42
6. Craig, I.D.: Formal Refinement for Operating System Kernels. Springer (2007)
7. Freitas, L.: Proving Theorems with **Z/Eves**. T. Report, University of Kent (2004)
8. Freitas, L., et al.: Posix and the verification grand challenge: A roadmap. In: 13th ICECCS, IEEE Computer Society (2008) 153–162
9. Freitas, L.: Extended Z mathematical toolkit – Verified Software Repository. Technical Report CRG13, University of York (2008)
10. Freitas, L.: Mechanising data-types for kernel design in Z. In: Formal Methods: Foundations and Applications. Volume 5902 of LNCS., Springer (2009) 186–203
11. Hall, A., Chapman, R.: Correctness by Construction: Developing a Commercial Secure System. IEEE Software **19**(1) (2002) 18–25
12. Jones, C., Woodcock, J., eds.: Formal Aspects of Computing: Special Issue on the Mondex Verification. Volume 20:1. Springer (2008)
13. Klein, G., et al.: seL4: Formal verification of an OS kernel. In: 22nd ACM Symposium on Operating Systems Principles (SOSP), ACM (2009)
14. McDermott, J., Freitas, L.: Formal security policy of Xenon. In: FMSE. (2008)
15. Rushby, J.M.: Design and verification of secure systems. ACM SIGOPS Operating Systems Review **15**(5) (1981) 12–21
16. Saaltink, M.: **Z/Eves** 2.2 User's Guide. Technical report, ORA (1999)
17. Saaltink, M.: **Z/Eves** 2.2 Mathematical Toolkit. Technical report, ORA (2003)
18. SKPP: U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness, v.1.0.3. National Security Agency (June 2007)
19. Velykis, A.: Formal modelling of separation kernels. Master's thesis, Department of Computer Science, University of York (2009)
20. Woodcock, J., Davies, J.: Using Z. Prentice-Hall (1996)
21. Woodcock, J.: First steps in the verified software grand challenge. IEEE Computer **39**(10) (2006) 57–64
22. Woodcock, J., Larsen, P.G., Bicarregui, J., Fitzgerald, J.: Formal Methods: Practice and Experience. ACM Computing Surveys **41**(4) (2009)