Annual Report for AOARD Grant #FA2386-11-1-4070

Formal System Verification - Extension

June Andronick, Andrew Boyton, Gerwin Klein

17 May 2012

Principal Investigators: Gerwin Klein and June Andronick

Email: gerwin.klein@nicta.com.au, june.andronick@nicta.com.au
Institution: NICTA
Mailing Address: 223 Anzac Parade, Kensington NSW 2052, Australia
Phone: +61 2 8306 0578
Fax: +61 2 8306 0406

Period of Performance: 18 May, 2011 - 17 Oct, 2012
Reporting Period: 18 May, 2011 - 17 May, 2012

1 Abstract

The AOARD project #FA2386-11-1-4070 aims at providing a provably correct *initialiser* of componentised systems. Taking as input a description of the desired components and the desired authorised communication between them, the initialiser sets up the system and provides a proof that the resulting concrete machine state of the system matches the desired authority state. Within the scope of this project, we provide (1) a formal *specification* of the initialiser, in terms of the steps needed to create the components and their communication channels; and (2) a formal proof that this specification is correct in that it either fails safely or produces the desired state.

This document is an annual report of the project, presenting the status of the work to date. Namely, we have written the initialiser specification, we have set up a verification framework enabling modular reasoning and proofs, and we have progressed substantially on the proof.

2 Introduction

This annual report describes the status of AOARD project #FA2386-11-1-4070, "Formal System Verification - Extension". This project runs from 18 May 2011 to 17 October 2012.

This project is part of a larger research vision of building truly trustworthy software systems; in particular the formal verification of large, complex embedded systems. A first step in this vision was to provide a formally verified microkernel basis. We have previously developed the seL4 microkernel, together with a formal proof (in the theorem prover Isabelle/HOL [5]) of its functional correctness [2]. This means that all the behaviours of the seL4 C source code are included in the high-level, formal specification of the kernel.

The approach taken to provide formal, code-level guarantees about the security or safety of million of lines of code is to minimise the *trusted computing base* by designing the system architecture in a componentised way, where untrusted components can be isolated from trusted ones. This MILS-style of security architecture [1] enables us to concentrate the verification effort on the trusted components. In our case, the isolation is provided by the underlying seL4 kernel, in terms of integrity and authority confinement [8], as well as confidentiality (in ongoing work).

The project reported here focuses on the critical step of initialising the system into a state satisfying the specified architecture. Namely, the project goal is to:

> *(1) construct a formal, high-level specification of the system initialiser component in the theorem prover Isabelle/HOL that is connected to the existing formal specification of the seL4 microkernel and*
> *(2) prove that this specification of the system initialiser component is correct in the sense that it will always either fail safely at initialisation time or produce a system state that formally corresponds to the specified target.*

We begin the remainder of this report with some background information on the seL4 microkernel and in particular its access control mechanism, which is critical to the system initialiser. We

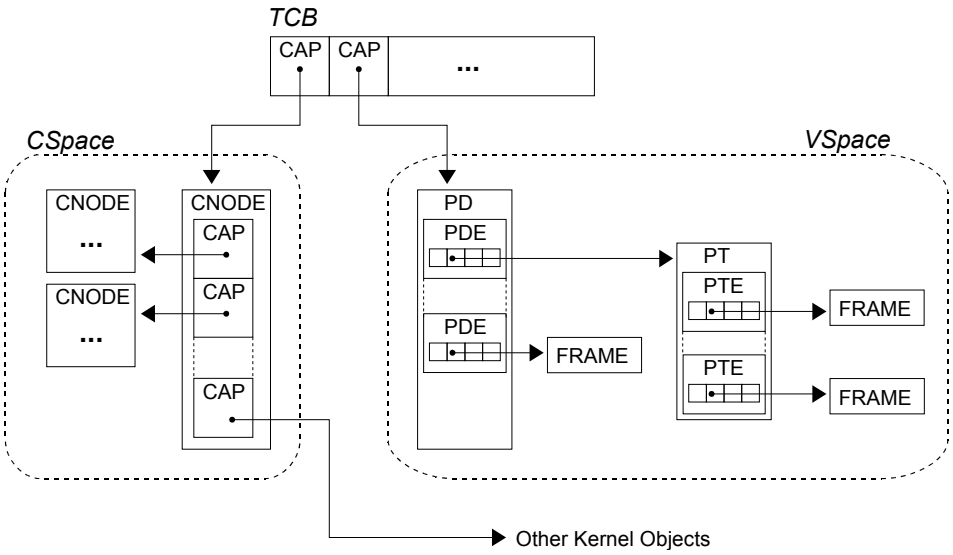

Figure 1: Internal representation of an application in seL4.

then describe the exact project scope and finally explain in detail our formal specification of the initialiser and the status of its proof of correctness. In particular, we describe our instantiation of *separation logic* that enables the decomposition of the proof into smaller statements that can be reasoned about independently. Additionally, we attach a publication describing the separation logic framework instantiated for the work presented here.

3 Background

3.1 seL4

The seL4 microkernel is a small operating system kernel designed to be a secure, safe, and reliable foundation for a wide variety of application domains. The microkernel is the only software running in the *privileged* mode (or *kernel* mode) of the processor. The privileged mode is used to protect the operating system from applications and applications from each other. Software other than the kernel runs in unprivileged mode (or *user* mode). As a microkernel, seL4 provides a minimal number of services to applications: threads, inter-process communication and virtual memory, as well as a capability-based access control. Threads are an abstraction of CPU execution that support running software. As shown in Figure 1, threads are represented in seL4 by their *thread control blocks* (TCB), that store a thread's context, virtual address space (VSpace) and capability space (CSpace). VSpaces contain a set of frames, generally organised in a hierarchical, architecture-dependent structure of page tables and page directories. CSpaces are kernel managed storage for *capabilities*. A capability is an unforgeable token that confers authority. In seL4, a thread may only invoke operations on

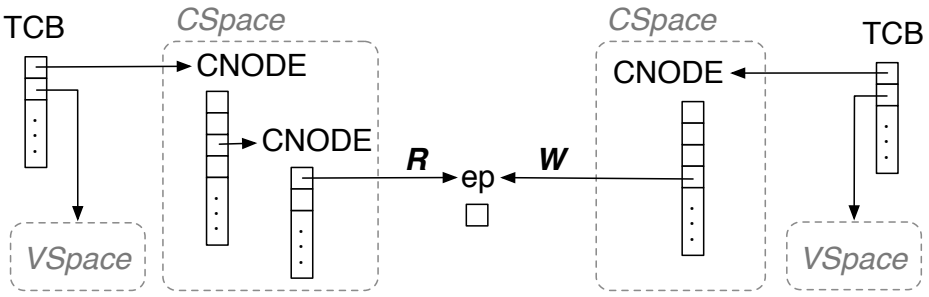

Figure 2: seL4-based system with two threads that can communicate via an *endpoint*.

objects when it has sufficient permissions. This means that every call to the kernel requires the presentation of a capability with the correct authority to perform the specified operation. A thread stores its capabilities in its CSpace, which is a directed acyclic graph of capability nodes (CNodes). These CNodes can be of various sizes and contain capabilities to other CNodes and objects. When a thread invokes seL4, it provides an index into this structure, which is transversed and resolved into a real capability. Communication between components is enabled by *endpoints*. When a thread wants to send a message to another thread, it must send it to an intermediate endpoint to which it has send access and to which the receiver has receive access, as shown in Figure 2.

The allocation of kernel objects in seL4 is performed by *retyping* an *untyped memory* region. Untyped memory is an abstraction of a power-of-two sized, size-aligned region of physical memory. Possession of capability to an untyped memory region provides sufficient authority to allocate kernel objects in this region: a user-level application can request that the kernel transforms that region into other kernel objects (including smaller untyped memory regions). At boot time, seL4 first preallocates the memory required for the kernel itself and then gives the remainder of the memory to the initial user task in the form of capabilities to untyped memory.

3.2 seL4 Verification

The seL4 microkernel was the first, and is still, to our knowledge, the only general-purpose operating system kernel that is fully formally verified for functional correctness. This means that there exists a formal, machine-checked proof that the C implementation of seL4 is a correct refinement of its functional, abstract specification. This proof assumes the correctness of the compiler, assembly code, boot code, management of caches, and the hardware. The technique used for formal verification is interactive, machine-assisted and machine-checked proof. Specifically, we use the theorem prover Isabelle/HOL [5].

As shown in Figure 3, the verification uses several specification layers. The top-most layer in the picture is the *abstract specification*: an operational model that is the main, complete specification of system behaviour. The next layer down is the *executable specification*, representing the design of the kernel. This layer is generated from a Haskell prototype of the kernel, aimed at bridging the

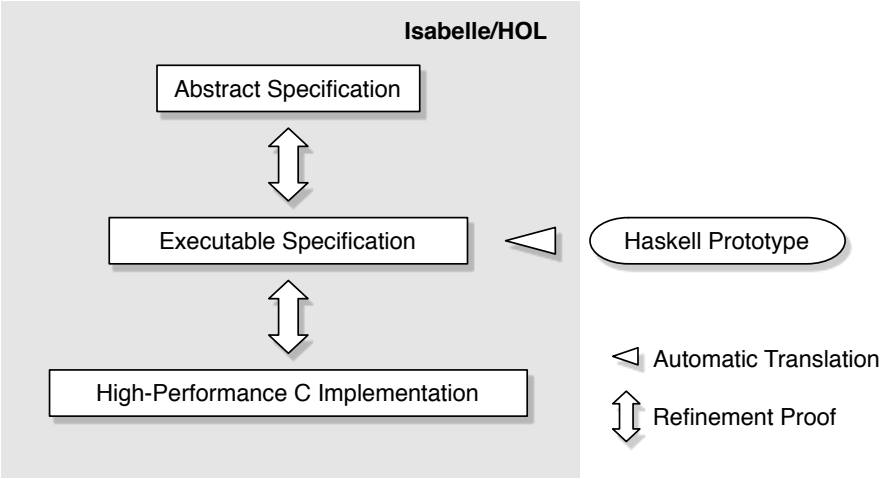

Figure 3: The refinement layers in the verification of seL4 functional correctness.

gap between kernel development needs and formal verification requirements. Finally, the bottom layer is the high-performance C implementation of seL4, parsed into Isabelle/HOL using a precise, faithful formal semantics for the C programming language [6, 9].

The correspondence established by the refinement proof enables to conduct further proofs of any *Hoare logic* properties (see Section 6) at the abstract level, ensuring that the properties also hold for the refined model. This means that if a security property is proved in Hoare logic about the abstract model (not all security properties can be), refinement guarantees that the same property holds for the kernel source code. Since proofs at the abstract level are easier to perform, this allows a significant reduction of effort in additional verifications, as illustrated by the proof that seL4 enforces integrity and authority confinement [8].

3.3 capDL

One of the design goals of the seL4 kernel was to capture all access control relevant state by capabilities. However, despite its capability-oriented design, seL4, like other microkernels, contains authority relevant to information flow and access control that is not conferred by capabilities. For example, the memory mapped into a thread's address space is not mapped using capabilities, nor are various hardware accesses such as IO ports.

Having all the protection state described by capabilities would enable reasoning about the access control and security of a system through capability distributions alone. It would also allow components and connections of user-level systems on top of seL4 to be described by their capability distribution alone. Such a description is basically a graph with objects as nodes and capabilities as edges. To reason about such specific graphs, we developed the capability distribution language capDL [4]. The capDL language unifies all information relevant to information flow and access

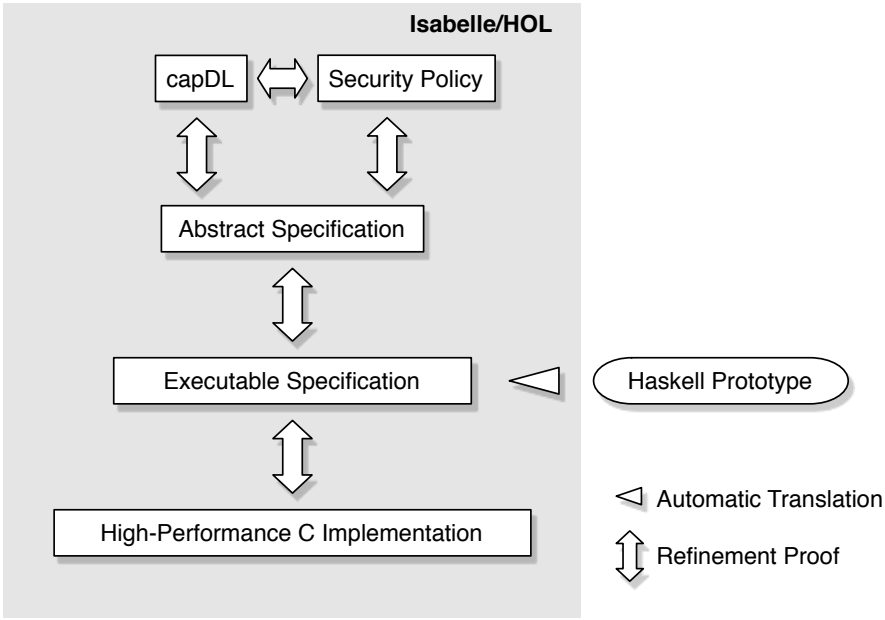

Figure 4: capDL model in the seL4 refinement chain.

control as explicit capabilities. In addition to the language itself, which describes snapshots of system states, we developed, in previous work, a kernel semantics for this language that describes the effect of each kernel operation on such states, and showed that it is a formally correct abstraction of existing models of seL4, with a complete refinement chain to the C code level, as shown in Figure 4. This high-level model of seL4, which will be referred to as the *seL4 capDL model*, fully encapsulates the protection state of the system.

Using capDL, we are able to precisely describe the protection state of a system we wish to run on seL4. For instance, the example system shown in Figure 2 can be formally described using the *capDL specification* shown in Figure 5. This capDL specification describes all system objects through a mapping from unique object identifiers to the objects themselves. Each object contains a set of capabilities, as well as other attributes, referred to as *fields*. For example, the object reference 0 maps to a TCB object whose CSpace root is the CNode at reference 2 and whose VSpace root is the page directory at reference 1. In turn, the CNode at reference 2 contains a set of capabilities, one of which is a capability to the CNode at reference 3, which itself contains the capability, with $Read$ right, to the endpoint at reference 4. Other than rights, capabilities may contain other attributes such as *guards* or other tags; objects also contain other capabilities or fields; and the capDL specification itself contains other information than the set of objects; but these are irrelevant for understanding the initialiser definition and proof.

In the project reported here, we use formal capDL specifications, similar to the one given in Figure 5, to describe the system to be initialised.

$$
\begin{array}{l}
(\!| \mathit{object\_heap} = \\
\quad [\,0 \mapsto \mathit{Tcb}\ (\!| \mathit{cdl\_tcb\_caps} = [\,0 \mapsto \mathit{CNodeCap}\ 2\ \mathit{guard}\ \mathit{guard\_size}, \\
\qquad\qquad\qquad\qquad\qquad 1 \mapsto \mathit{PageDirectoryCap}\ 1\ \mathit{True}\ \mathit{None}, \ldots\,], \\
\qquad\qquad \mathit{cdl\_tcb\_fault\_endpoint} = \ldots, \\
\qquad\qquad \mathit{cdl\_tcb\_intent} = \ldots\ |\!), \\
\qquad 1 \mapsto \mathit{PageDirectory}\ \ldots, \\
\qquad 2 \mapsto \mathit{CNode}\ (\!| \mathit{cdl\_cnode\_caps} = [\ldots,\ 2 \mapsto \mathit{CNodeCap}\ 3\ \mathit{guard}\ \mathit{guard\_size},\ \ldots\,], \\
\qquad\qquad\qquad \mathit{cdl\_cnode\_size\_bits} = \ldots\ |\!), \\
\qquad 3 \mapsto \mathit{CNode}\ (\!| \mathit{cdl\_cnode\_caps} = [0 \mapsto \mathit{EndpointCap}\ 4\ \mathit{badge}\ \{\mathit{Read}\}, \ldots\,], \\
\qquad\qquad\qquad \mathit{cdl\_cnode\_size\_bits} = \ldots\ |\!), \\
\qquad 4 \mapsto \mathit{Endpoint}, \\
\qquad 5 \mapsto \mathit{Tcb}\ (\!| \mathit{cdl\_tcb\_caps} = [\,0 \mapsto \mathit{CNodeCap}\ 6\ \mathit{guard}\ \mathit{guard\_size}, \\
\qquad\qquad\qquad\qquad\qquad 1 \mapsto \mathit{PageDirectoryCap}\ 7\ \mathit{True}\ \mathit{None}, \ldots\,], \\
\qquad\qquad \mathit{cdl\_tcb\_fault\_endpoint} = \ldots, \\
\qquad\qquad \mathit{cdl\_tcb\_intent} = \ldots\ |\!), \\
\qquad 6 \mapsto \mathit{CNode}\ (\!| \mathit{cdl\_cnode\_caps} = [0 \mapsto \mathit{EndpointCap}\ 4\ \mathit{badge}\ \{\mathit{Write}\}], \\
\qquad\qquad\qquad \mathit{cdl\_cnode\_size\_bits} = \ldots\ |\!), \\
\qquad 7 \mapsto \mathit{PageDirectory}\ \ldots, \\
\qquad \ldots\,], \\
\ldots \\
\mathit{current\_thread} = \ldots\ |\!)
\end{array}
$$

Figure 5: capDL specification of Figure 2.

4 Project Scope

Given a formal capDL specification describing the target system, we need to produce initialiser code that runs at system startup to give a state where all the components have been created and their communication channels set up.

As explained in Subsection 3.1, when a seL4 system starts, the kernel creates an initial user task – the *root task* – with access rights to all of the memory not used by the kernel itself. More precisely, the kernel creates all the objects needed by the root task such as its TCB, capability space and virtual address space. Capabilities to untyped memory are stored in the capability space, together with capabilities to allow hardware access. At the end of this booting phase, the root task is enabled to run and starts executing.

Our work produces a custom root task to perform system initialisation. We provide both *code* describing its execution and a *proof* of its correctness (this latter work is in progress).

The code involves allocating objects, managing capabilities to the objects, copying or transferring authority, managing and mapping frames, and setting any required data (such as thread instruction pointers). Manually performing these tasks for each given system is complicated, error prone, and inflexible. For instance, creating an object requires possessing a capability to an untyped memory region of suitable size, which in turn requires possessing a slot in the root task's CSpace to store this capability, and so on. A slight change in the desired system might require storing more capabilities, which may be difficult or impossible if the root task's CSpace size is hardcoded. Instead we choose to provide a *generic tool*, that takes any capDL specification as input, and *automatically* produces

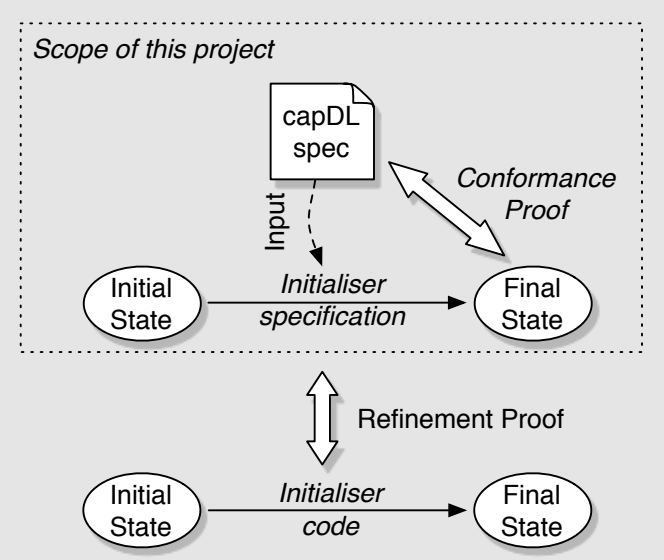

Figure 6: Overview of initialiser correctness proof.

code to create and initialise the objects.

Together with the code, the tool also generates a proof asserting that after executing the code from the initial state after kernel booting, the system is in a final state *conformant* to the original capDL specification. We use a notion of conformance instead of equality for two reasons. First, as will be described in detail in Subsection 7.3, capDL specifications use object identifiers that need to be mapped to memory addresses. Second, we have not yet examined different options for deleting the root task after system initialisation. Therefore the root task stays dormant in memory, but will be provably isolated from the rest of the system. Future work could look into removing the root task from memory after it has finished initialising the system. In this project, the final state after initialisation will therefore contain some root task objects in addition to the objects described in the capDL specification.

As formal proof at the source code level requires significant effort, we follow the approach successfully used for seL4 security proofs, by performing the proof at the abstract specification level, and relating it to the source code through the refinement proof. We therefore describe the behaviour of the initialiser by a set of high-level instructions in the seL4 capDL model. We call this behaviour description the *initialiser specification* and we prove conformance with the capDL specification at the initialiser specification level. Later work (outside the scope of this project) will involve producing an implementation of the initialisation code in C, and proving a formal refinement between the initialiser specification and the C implementation, as illustrated in Figure 6.

To summarise, within the scope of this project:

- we have defined a formal specification `init_system` as a set of high level instructions,

which takes a capDL specification `spec` as input, and sets up objects and capabilities as required by `spec`;

- we are in the process of proving that executing `init_system spec` from a state where only the root task's objects exist results in a final state where each object mentioned in `spec` maps to an object in the state's heap, and the only other existing objects are the root task's objects. The desired property is stated formally as:

  If `well_formed spec` *and* `injective` φ *and*
  `object_ids = dom (object_heap spec)` *then*

  $\{\!|$ `<< root_objects >>` $|\!\}$ `init_system spec`
  $\{\!|$ `<<` $\bigwedge^*$ `map (object_done spec` φ`) object_ids` $\wedge^*$ `root_objects >>` $|\!\}$

  This statement will be explained in depth in Section 7. It assumes a wellformedness condition about `spec` and the existence of an injection φ mapping object identifiers to memory addresses. It uses the predicate `object_done` stating that a given object is successfully created according to `spec`, and iterates this predicate over the list `object_ids` of all identifiers of the objects to be created in `spec`.

5 Specification of the System Initialiser

The initialiser specification is divided into the following well-defined separate phases.

- Firstly, the initialiser processes the information provided by the kernel (in the initial state), and the capDL specification itself.
- Secondly it creates all the objects specified in the capDL specification.
- Finally, it initialises each of these objects by type, including installing the capabilities into the capability storage objects, and sets all threads to be runnable.

This clear separation into phases is designed to assist in the formal proof of the initialiser's correctness, as it makes it relatively easy to specify the state of the system at any given point. Each phase is now described in detail, illustrated by the example given in Figure 2 and Figure 5.

The initialiser first analyses the boot information provided by seL4. For instance, the initial state of our example system is illustrated in Figure 7. The initialiser establishes the size and location of the capabilities to untyped memory and stores a list of free capability slots for storing the capabilities to the objects it will create. The number of free capability slots provided to the root task is specified at compile time for seL4. It is therefore possible to make sure that there will be enough free slots available for initialisation of a specific system.

The initialiser then creates all the objects required by the capDL specification. Objects are created sequentially from the untyped memory regions, as illustrated in Figure 8. Each untyped

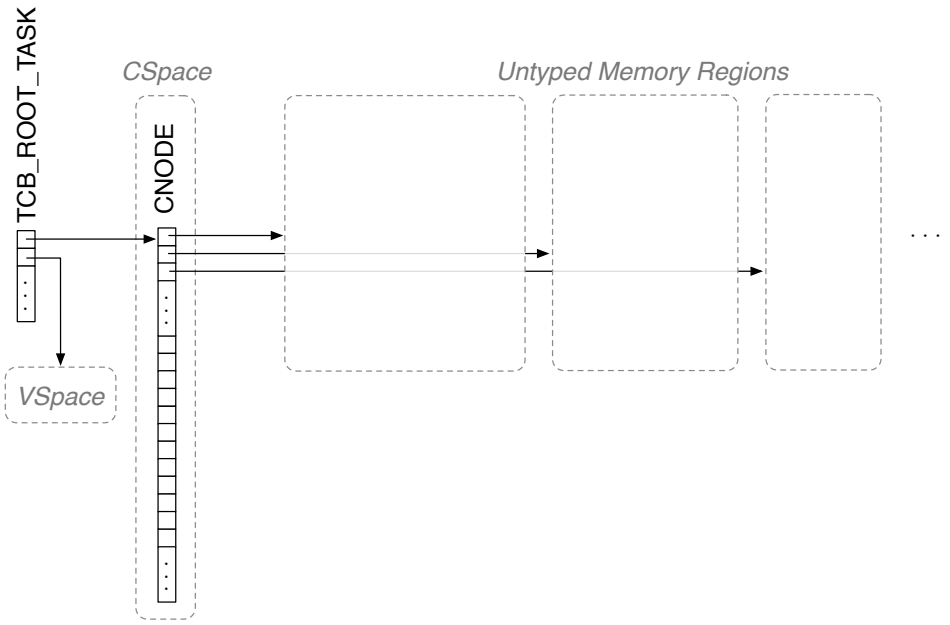

Figure 7: Initial state (after kernel booting) of the system described in Figure 2 and Figure 5.

memory region can be retyped incrementally to create new objects. Any ordering of the creation of objects, and any choice of the untyped memory region they are created from, is safe. Some orderings may however make inefficient use of space as seL4 requires all objects to be aligned to their size. Ordering the creation of objects from the largest object to the smallest, and from the largest untyped memory region to the smallest, is conjectured to be optimal, but proving it is outside the scope of this work. The capabilities to each of these newly created objects are stored in the free capability slots of the initialiser, and the locations of the capabilities to these objects are kept in the bookkeeping done by the initialiser.

Objects are created in a default state. Their content must be initialised, which is done per object type: the virtual address spaces are first initialised, then the thread control blocks, and finally the capability spaces.

As explained in Subsection 3.1, the virtual address space of each thread in seL4 consists of a page directory containing either page tables or large frames. Each page table in turn contains a number of frames. The initialisation of virtual address spaces consists in mapping the required page tables and large frames into the page directories, and mapping the frames into the page tables, as directed by the capDL specification.

Thread initialisation is performed via seL4 system calls that set the required data, such as the capabilities to the VSpace and CSpace, for each of the TCBs.

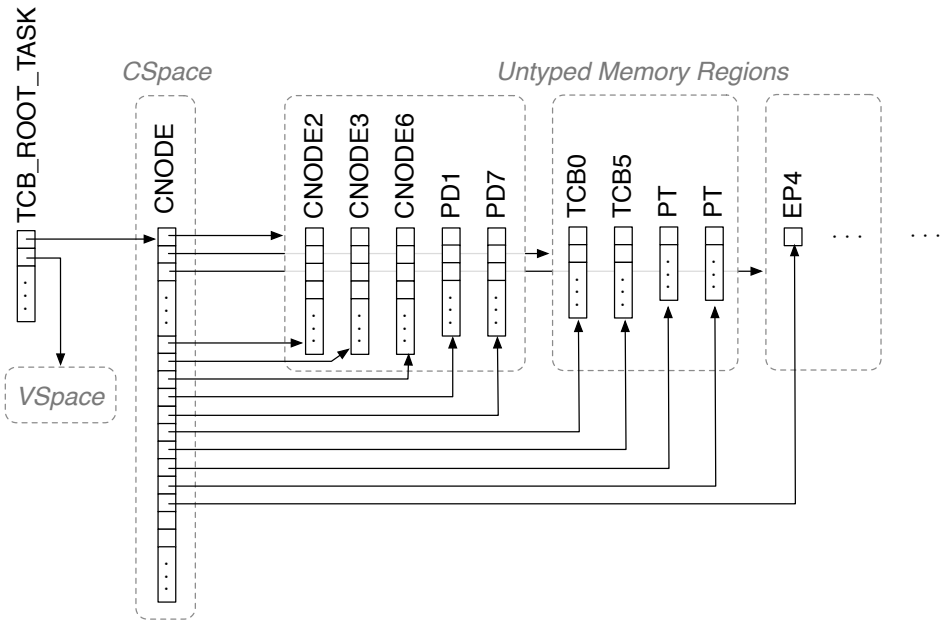

Figure 8: Creation of objects for the system described in Figure 2 and Figure 5.

The capability spaces are initialised similarly to the virtual address spaces. The complication with capability spaces is that, unlike virtual address spaces which are fixed, two-level data structures, capability spaces can be arbitrary directed acyclic graphs. This causes a problem when the initialiser must *move* (instead of *copying*) its own capabilities into some of the CNodes. Capabilities are moved instead of copied due to a distinction in seL4 between *master* capabilities (which are the original capabilities acquired at object creation), and *derived* capabilities (which are copies of these original capabilities). Master capabilities confer more authority as they allow for the complete deletion of an object. When initialising a system, we want to provide some of the objects with master capabilities (with potentially diminished permissions if required by the capDL specification). Therefore, if the initialiser moves its own (master) capability to a CNode A into another CNode, it can no longer access CNode A to add more capabilities to it. Rather than defining a complex dependency ordering between the various moves of CNode capabilities, the initialiser duplicates all its CNode capabilities by copying them into its own CSpace. The master capabilities can then be moved where needed in the system's CNodes (with potentially diminished permissions) and the initialiser can still access all the CNodes using the duplicated capabilities.

The last action of the initialiser is to set each thread state to runnable. It then becomes dormant in memory, isolated from the rest of the system.

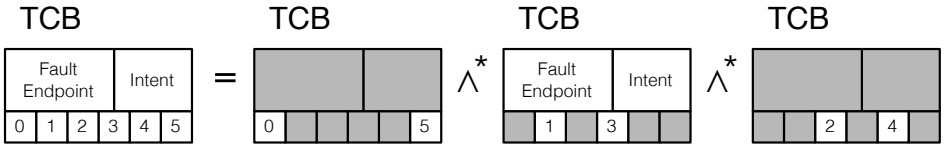

Figure 9: The decomposition of a thread control block into three parts, one containing capability slots 0 and 5, another containing the fields (the fault endpoint and intent) and capability slots 1 and 3 and another containing capability slots 2 and 4.

6 Abstract Separation Logic

When reasoning about the system initialiser specification, we wish to precisely capture the semantics of each operation. Axiomatic semantics are commonly represented in *Hoare logic*, using *Hoare triples*. These triples, written $\{\!|P|\!\}\ f\ \{\!|Q|\!\}$ expresses that if P is true before an operation f, then Q will be true afterwards.[1]

Many of the operations in the initialiser specification either iterate over a list of objects, or over the capability slots of an object. Expressing that an operation affects only a single capability slot, and chaining such operations together is difficult in traditional Hoare logic. For instance, to express the fact that a loop correctly initialises all the thread control blocks, we need to express that the initialisation of each TCB is correct *and that it does not affect the other TCBs*. We also need to express that the composition of these operations performs as expected. Similarly, when reasoning about the initialisation of specific capability slots *within an object*, we need to express that the capability slots are correctly initialised and that the other capability slots in the object are unaffected. Such reasoning in Hoare logic requires explicitly description of the unmodified parts of the state.

Separation logic — an extension to Hoare logic first introduced by Reynolds [7] — is designed for modular reasoning about operations that only manipulate portions of the state. In this work we have defined a separation logic to reason independently about various objects of a system state and various parts within objects. This separation logic is an instantiation of an abstract separation algebra we developed [3]. We partition all the objects of the logical abstract state — the memory heap — into smaller disjoints heaps, to allow us to specify that an operation affects only a portion of the memory heap. We also allow partitioning within objects to specify that an operation affects only a particular slot of an object and leaves other slots and fields unchanged. For example, we can partition a thread control block into three portions, as shown in Figure 9. This logical partitioning allows modular reasoning about the initialisation specification, that is we can effectively initialise each part separately.

Formally, we express that P and Q are true for separate parts of a heap h, by the separation conjunction $P \wedge^* Q$.[2] We also define the lifted separation conjunction operator $\bigwedge^* Ps$ which joins

[1] Hoare logic is normally used to prove partial correctness — if f terminates, then Q will be true. Termination can be proven separately, if desired.

[2] $P \wedge^* Q$ is defined as $(P \wedge^* Q)\ h = (\exists p\ q.\ p \#\# q \wedge h = p + q \wedge P\ p \wedge Q\ q)$, that

a list of separation predicates `Ps` with separation conjunction (i.e. $\bigwedge^*$ `[P1, P2] = P1` $\wedge^*$ `P1`). This definition of separation conjunction allows us to prove concise rules about loops, such as the following rule about `mapM_X`, which runs the operation `f` on each object in the list `xs`.

$$\forall P\ x.\ \{\!|Q\ x \wedge^* P|\!\}\ f\ x\ \{\!|R\ x \wedge^* P|\!\} \Longrightarrow$$
$$\{\!|\bigwedge^*\ map\ Q\ xs \wedge^* P|\!\}\ mapM\_x\ f\ xs\ \{\!|\bigwedge^*\ map\ R\ xs \wedge^* P|\!\}$$

The arbitrary `P` in this rule specifies that any predicate on the rest of the state is preserved, which means that the rest of the heap must be unchanged.

The state used in separation logic predicates has two parts, the memory heap containing the objects, and an additional *component map*, which states which parts of an object are "owned" by an object (and which is used in the definition of the heap partitioning). To apply separation predicates to the kernel state we use the syntax `<P>`, and to apply separation predicates to the initialiser state we use the syntax `<< P >>`.

By allowing partitions within objects, we can specify that an operation like `set_cap` changes only a single slot, and leaves everything else (all other objects and all other slots) unchanged. We express this property using the following Hoare triple.

$$\{\!|<ptr \mapsto_c old\_cap \wedge^* P>|\!\}\ set\_cap\ ptr\ cap\ \{\!|<ptr \mapsto_c cap \wedge^* P>|\!\}$$

We have defined predicates that express when an object identifier maps to an object ($ptr \mapsto_o object$), when a capability reference maps to a capability in an object ($cap\_ref \mapsto_c cap$), and also when an object identifier maps to an object containing just slots and no fields ($ptr \mapsto_S obj$), fields and no slots ($ptr \mapsto_f obj$), or just a single slot ($cap\_ref \mapsto_s obj$). This allows us to decompose the reasoning about the initialisation of an object into reasoning about initialising the capabilities for an object separately from initialising the fields of an object.

We use this separation logic to reason about the semantics of the initialiser specification and about the capDL model of seL4 itself. We have proven the rule for `set_cap` about the seL4 capDL model and other leaf functions, and are using these to prove concise semantics for the behaviour of the seL4 kernel operations in specific contexts. These semantics, which are preserved by refinement and thus true of the seL4 code itself, are used when proving the initialisation specification.

7 Proof Sketch

Now that we have described the specification of the initialiser and the verification framework that enables us to reason compositionally about each of its phase, we can explain the ongoing proof that the initialiser specification is correct, in that it builds the system as described in the capDL specification given as input. This proof requires some restrictions on the capDL specification and involves proving some invariants about the system state.

---

is, there exists two disjoint memory heaps, p and q, that add to give h, and for which P is true of p and Q is true of q.

7.1 Assumptions on the capDL Specification

The capDL language allows the description of infeasible systems, such that those with objects having infinite number of capability slots, or more slots than the object size allows. Our tool can obviously not configure systems described by such capDL specifications. The proof of correctness therefore assumes that the capDL specification is *well_formed*. The definition of wellformedness is being refined in response to the requirements of the evolving proof. Thus far we only require the number of capability slots to be finite, but we expect to have stronger requirements by the end of this work. These constraints on the capDL specification can be established before attempting to generate an initialiser.

```
well_formed spec ≡ ∀obj. finite (dom (slots_of obj spec))
```

7.2 Invariants

The proof relies on some invariants about the state of the system. The invariants are properties about either the internal state of the kernel, i.e. about kernel objects such as threads or communication channels, or the user state of the initialiser, such as its bookkeeping information, which must be kept consistent with the kernel state.

Invariants are properties that the proof requires and that are typically identified as the proof develops. Thus far, we have identified several invariants and we expect to have more by project completion.

In terms of the kernel state, we require that the initialisation program is the only runnable thread. This way the proof can rely on the fact that the execution of the initialiser is not interleaved with executions of other threads. This in turns requires that there is no inter-process communication (IPC) between threads during initialisation. Indeed, if a thread is blocked waiting for a message to arrive at an endpoint, and if IPC is allowed, then sending a message to this endpoint would turn the blocked thread into a runnable one. The invariants are stated as follows, where we specify that there is no pending IPC by ensuring that there are no pending capabilities.

```
kernel_state_invs s ≡
(∀cap∈ran (caps_of_state s). ¬ is_pending_cap cap) ∧
all_active_tcbs s = {root_thread} ∧ current_thread s = Some root_thread
```

The proof of these invariants is in progress and is expected to follow from a seL4 invariant stating that the root task is the highest priority thread and from the fact that if the root task is the only task running, it can ensure that there is no inter-process communication by simply not initiating any itself.

Finally we identified an invariant specifying the relationship between the initialisation code's state and that of the kernel. This invariant specifies the correctness of the bookkeeping performed by the initialiser in terms of the locations of capabilities and identities of the objects they refer to. The phrasing of this invariant is still being developed. The invariant should be established during object creation, and maintained by subsequent operations.

7.3 Final Theorem

The final result of this project will be a theorem about the initialiser specification stating that, at the end of the initialisation, all the objects in the system either existed at the start (those for the root task itself, created during kernel booting), or that they are conformant with the capDL specification.

In capDL specifications, systems are described as a mapping from object identifiers to objects. We treat these object identifiers simply as identifiers, rather than as memory addresses. When objects are created, their location in memory is decided by an allocator that runs at system initialisation. The initialiser ensures that there is an injective (one-to-one) function φ between the identifiers of the objects in the capDL specification and the memory addresses of corresponding objects in the initialised system. In other words, no two objects are mapped to the same memory address. The use of an online allocator allows more adaptability to hardware and specification changes. However, it also means that the capDL specification cannot specify the memory addresses for objects. It would be possible to instead allocate memory addresses for objects offline. The capDL specification would then use these memory addresses as object identifiers, and the system initialiser would create objects at the specified memory addresses. This is presently outside the scope of this work.

To reason about the relation between the capDL specification and the concrete kernel state, the injection between object identifiers and memory addresses is stored in *ghost state* upon object construction by the initialiser. The ghost state represents information used by the proof of the specification, but not the specification itself. The injection maps the object graph of the capDL specification to the one of the kernel state. Technically, such a mapping is called an injective homomorphism, or monomorphism. This monomorphism is used to map all the object identifiers in the capDL specification, as well as all object identifiers within the capabilities of each object — the latter defined by the function `spec2s` (which takes the injection φ as a parameter). Indeed, each of the objects in the capDL specification will be created at a particular memory address and the capabilities in the kernel will refer to the memory addresses of the corresponding objects, not the identifiers used in the capDL specification.

We define the function `object_done` that indicates that a given object in the capDL specification has been created, in that it corresponds to the respective object in the kernel state as illustrated in Figure 10. The formal definition is the following.

```
object_done spec φ spec_object_id s ≡
∃ kernel_object_id kernel_object spec_object.
   φ spec_object_id = Some kernel_object_id ∧
   (kernel_object_id ↦o kernel_object) s ∧
   object_heap spec spec_object_id = Some spec_object ∧
   intent_reset (spec2s φ spec_object) = intent_reset kernel_object
```

It can be read as follow: the object identified by `spec_object_id` in the capDL specification `spec` is said to be *done* in state `s` if there exists a corresponding `kernel_object_id` mapped to `spec_object_id` via the injection φ such that the object pointed to by `spec_object_id` is (almost) the same as the one pointed to by `kernel_object_id`. It is only *almost* the same because each thread control block contains an *intent* field in the specification, used to encode the kernel

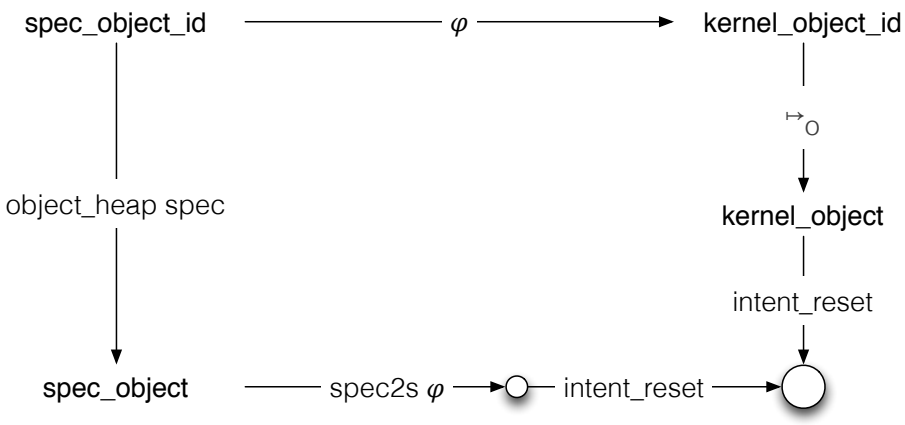

Figure 10: Diagrammatic representation of `object_done` definition.

instruction that the thread wishes to execute, and which our proof does not make any guarantee about its content. The intent is therefore overwritten to a default value through `reset_intent` before comparing objects.

The final result of the system initialiser is:

Theorem.

If `well_formed spec` *and* `injective φ` *and*
`object_ids = dom (object_heap spec)` *then*

```
{|<< root_objects >>|} init_system spec
{|<< ⋀* map (object_done spec φ) object_ids ∧* root_objects >>|}
```

It states that each object of the capDL specification has been correctly initialised, and that the other objects created by the kernel booting remain unchanged.

7.4 Decomposition of the Final Theorem

The separation algebra defined in Section 6 defines a predicate `ptr ↦o object` stating that an object `object` is at a memory address `ptr`. As in the separation logic objects can be decomposed into their fields and capability slots, the property `object_done` can be similarly decomposed into stating that an object is created in a default state, that its fields are correctly initialised, and that its capability slots are correctly initialised. The definition that an object's capability slots are initialised can be similarly decomposed into stating that each individual capability slot is initialised. Such a decomposition is generally a difficult result to prove and often not true, but our separation algebra gives us this result with relative ease.

This decomposition allows elegant expression of the properties of each phase of initialisation, and of each individual operation of initialisation.

8 Conclusion and Current Status

This project has made significant progress towards enabling provably correct configuration of systems according to desired architectures. The critical task of initialising components and communication channels is tedious and error prone. This work addresses it by providing a generic initialiser, able to automatically build seL4-based componentised systems from precise configuration descriptions, and, by project completion, a formal correctness proof.

To date, the achievements of the project are the following:

- a complete formal specification of system initialisation;
- a separation algebra suitable for compositional reasoning about the semantics of both seL4 and the initialiser specification;
- a definition of the correctness property required from the initialiser in terms of what it means for an object to be correctly initialised;
- a decomposition of the correctness property of the whole initialisation into properties about the various phases focusing on smaller parts of the system;
- identification of an initial set of required invariants for the proof;
- initial progress on the proof of the various focused phases.

Whilst the process is somewhat iterative, most of the infrastructure is in place and the framework for the proof is nearing completion. With the framework stabilising, the proof has begun to progress. We have successfully used our separation algebra to reason about most of the kernel calls used in the initialiser specification, and we expect the proofs for the other kernel calls to be similar.

More generally, we believe that we have a developed a good foundation for developing proofs about other user level programs running on seL4.

Acknowledgements We are grateful to Timothy Bourke for his feedback on drafts of this paper.

References

[1] Jim Alves-Foss, Paul W. Oman, Carol Taylor, and Scott Harrison. The MILS architecture for high-assurance embedded systems. *Int. J. Emb. Syst.*, 2:239–247, 2006.

[2] 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.

[3] Gerwin Klein, Rafal Kolanski, and Andrew Boyton. Mechanised separation algebra. In *3rd ITP*, 2012. To appear.

[4] Ihor Kuz, Gerwin Klein, Corey Lewis, and Adam Walker. capDL: A language for describing capability-based systems. In *1st APSys*, pages 31–36, New Delhi, India, Aug 2010.

[5] Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[6] Michael Norrish. C-to-Isabelle parser, version 0.7.2. http://ertos.nicta.com.au/software/c-parser/, Jan 2012.

[7] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. 17th IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[8] 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-Verlag.

[9] Harvey Tuch, Gerwin Klein, and Michael Norrish. Types, bytes, and separation logic. In Martin Hofmann and Matthias Felleisen, editors, *34th POPL*, pages 97–108. ACM, 2007.

Attachments

Mechanised Separation Algebra

Gerwin Klein, Rafal Kolanski, and Andrew Boyton

[1] NICTA, Sydney, Australia*
[2] School of Computer Science and Engineering, UNSW, Sydney, Australia
{first-name.last-name}@nicta.com.au

Abstract. We present an Isabelle/HOL library with a generic type class implementation of separation algebra, develop basic separation logic concepts on top of it, and implement generic automated tactic support that can be used directly for any instantiation of the library. We show that the library is usable by multiple example instantiations that include structures such as a heap or virtual memory, and report on our experience using it in operating systems verification.

Keywords: Isabelle, Separation Logic

1 Introduction

The aim of this work is to support and significantly reduce the effort for future separation logic developments in Isabelle/HOL by factoring out the part of separation logic that can be treated abstractly once and for all. This includes developing typical default rule sets for reasoning as well as automated tactic support for separation logic. We show that both of these can be developed in the abstract and can be used directly for instantiations.

The library supports users by enforcing a clear axiomatic interface that defines the basic properties a separation algebra provides as the underlying structure for separation logic. While these properties may seem obvious for simple underlying structures like a classical heap, more exotic structures such as virtual memory or permissions are less straight-forward to establish. The library provides an incentive to formalise towards this interface, on the one hand forcing the user to develop an actual separation algebra with actual separation logic behaviour, and on the other hand rewarding the user by supplying a significant amount of free infrastructure and reasoning support.

Neither the idea of separation algebra nor its mechanisation is new. Separation algebra was introduced by Calcagno et al [2] whose development we follow, transforming it only slightly to make it more convenient for mechanised instantiation. Mechanisations of separation logic in various theorem provers are plentiful, we have ourselves developed multiple versions [5,6] as have many others. Similarly a

* 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 funded by AOARD grant #FA2386-11-1-4070

number of mechanisations of abstract separation algebra exist, e.g. by Tuerk [7] in HOL4, by Bengtson et al [1] in Coq, or by ourselves in Isabelle/HOL [5].

The existence of so many mechanisations of separation logic is the main motivation for this work. Large parts of developing a new separation logic instance consist of boilerplate definitions, deriving standard properties, and often re-invented tactic support. While separation algebra is used to justify the separation logic properties of specific developments [5], or to conduct a part of the development in the abstract before proceeding to the concrete [1, 7], the number of instantiations of these abstract frameworks so far tends to be one. In short, the library potential of separation algebra has not been exploited yet in a practically re-usable way. Such lightweight library support with generic interactive separation logic proof tactics is the contribution of this paper.

A particular feature of the library presented here is that it does not come with a programming language, state space, or a definition of hoare triples. Instead it provides support for instantiating your own language to separation logic. This is important, because fixing the language, even if it is an abstract generic language, destroys most of the genericity that separation algebra can achieve. We have instantiated our framework with multiple different language formalisations, including both deep and shallow embeddings. The library is available for download from the Archive of Formal Proofs [4].

In Sec 2 we show the main interface of the separation algebra class in Isabelle/HOL and describe how it differs from Calcagno et al. Sec 3 describes the generic tactic support, and Sec 4 describes our experience with example instances.

2 Separation Algebra

This section gives a brief overview of our formulation of abstract separation algebra. The basic idea is simple: capture separation algebra as defined by Calcagno et al [2] with Isabelle/HOL type class axioms, develop separation logic concepts in the abstract as far as can be done without defining a programming language, and instantiate simply using Isabelle's type class instantiations. This leads to a lightweight formalisation that carries surprisingly far.

Calcagno et al define separation algebra as *a cancellative, partial commutative monoid* $(\Sigma, \cdot, u)$*. A partial commutative monoid is given by a partial binary operation where the unity, commutativity and associativity laws hold for the equality that means both sides are defined and equal, or both are undefined.* [2]

For a concrete instance, think of the carrier set as a heap and of the binary operation as map addition. The definition induces separateness and substate relations, and is then used to define separating conjunction, implication, etc. Since the cancellative property is needed primarily for completeness and concurrency, we leave it out at the type class level. If necessary, it could be introduced in a second class on top. The definition above translates to the following class axioms.

```
x ⊕ 0 = Some x      x ⊕ y = y ⊕ x      a ++ b ++ c = (a ++ b) ++ c
```

where `op ⊕::'a ⇒ 'a ⇒ 'a option` is the partial binary operator and `op ++::'a option ⇒ 'a option ⇒ 'a option` is the $\oplus$ operator lifted to strict partiality

with None ++ x = None. From this the usual definitions of separation logic can be developed. However, as to be expected in HOL, partiality makes the $\oplus$ operator cumbersome to instantiate; especially the third axiom often leads to numerous case distinctions. Hence, we make the binary operator total, re-using standard + as syntax. Totality requires us to put explicit side conditions on the laws above and to make disjointness ## a parameter of the type class leading to further axioms. The full definition of separation algebra with a total binary operator is

```
class sep_algebra = zero + plus +
  fixes op ##::'a ⇒ 'a ⇒ bool
  assumes x ## 0  and  x ## y ⟹ y ## x  and  x + 0 = x
  assumes x ## y ⟹ x + y = y + x
  assumes ⟦x ## y; y ## z; x ## z⟧ ⟹ x + y + z = x + (y + z)
  assumes ⟦x ## y + z; y ## z⟧ ⟹ x ## y
  assumes ⟦x ## y + z; y ## z⟧ ⟹ x + y ## z
```

This form is precisely as strong as Calcagno et al's formulation above in the sense that either axiom set can be derived from the other. The last two axioms are encapsulated in the original associativity law. The more intuitive form x ## y ⟹ x + y ## z = (x ## z ∧ y ## z) is strictly stronger.

While 7 axioms may seem a higher burden than 3, the absence of lifting and type partiality made them smoother to instantiate in our experience, in essence guiding the structure of the case distinctions needed in the first formulation.

Based on this type class, the definitions of basic separation logic concepts are completely standard, as are the properties we can derive for them. Some definitions are summarised below.

```
P ∧* Q   ≡ λh. ∃x y. x ## y ∧ h = x + y ∧ P x ∧ Q y
P ⟶* Q  ≡ λh. ∀h'. h ## h' ∧ P h' ⟶ Q (h + h')
x ≼ y    ≡ ∃z. x ## z ∧ x + z = y
□        ≡ λh. h = 0
⋀* Ps    ≡ foldl (op ∧*) □ Ps
```

On top of these, we have formalised the standard concepts of pure, intuitionistic, and precise formulas together with their main properties. We note to Isabelle that separating conjunction forms a commutative, additive monoid with the empty heap assertion. This means all library properties proved about this structure become automatically available, including laws about fold over lists of assertions.

From this development, we can set up standard simplification rule sets, such as maximising quantifier scopes (which is the more useful direction in separation logic), that are directly applicable in instances.

The assertions we cannot formalise on this abstract level are maps-to predicates such as the classical $p \mapsto v$. These depend on the underlying structure and can only be done on at least a partial instantiation.

Future work for the abstract development could include a second layer introducing assumptions on the semantics of the programming language instance. It then becomes possible to define locality, the frame rule, and (best) local actions generically for those languages where they make sense, e.g. for deep embeddings.

3 Automation

This section gives a brief overview of the automated tactics we have introduced on top of the abstract separation algebra formalisation.

There are three main situations that make interactive mechanical reasoning about separation logic in HOL frameworks cumbersome. Their root cause is that the built-in mechanism for managing assumption contexts does not work for the substructural separation logic and therefore needs to be done manually.

The first situation is the application of simple implications and the removal of unnecessary context. Consider the goal `(P ∧* p ↦ v ∧* Q) h ⟹ (Q ∧* P ∧* p ↦ -) h`. This should be trivial and automatic, but without further support requires manual rule applications for commutativity and associativity of `∧*` before the basic implication between `p ↦ v` and `p ↦ -` can be applied. Rewriting with AC rules alleviates the problem somewhat, but leads to unpleasant side effects when there are uninstantiated schematic variables in the goal. In a normal, boolean setting, we would merely have applied the implication as a forward rule and solved the rest by assumption, having the theorem prover take care of reordering, unification, and assumption matching.

While in a substructural logic, we cannot expect to always be able to remove context, at least the order of conjuncts should be irrelevant. We expect to apply a rule of the form `(P ∧* Q) h ⟹ (P' ∧* Q) h` either as a forward, destruction, or introduction rule where the real implication is between `P` and `P'` and `Q` tells us it can be applied in any context. Our tactics *sep_frule*, *sep_drule*, and *sep_rule* try rotating assumptions and conclusion of the goal respectively until the rule matches. If `P` occurs as a top-level separation conjunct in the assumptions, this will be successful, and the rule is applied. This takes away the tedium of positional adjustments and gives us basic rule application similar to plain HOL. The common case of reasoning about heap updates falls into this category. Heap update can be characterised by rules such as `(p ↦ - ∧* Q) h ⟹ (p ↦ v ∧* Q) (h(p ↦ v))` if `h` is a simple heap map. If we encounter a goal with an updated heap `h(p ↦ v)` over a potentially large separating conjunction that mentions the term `p ↦ v`, we can now make progress with a simple *sep_rule* application.

Note that while the application scenario is instance dependent, the tactic is not. It simply takes a rule as parameter.

The second situation is reasoning about heap values. Again, consider a simple heap instantiation of the framework. The rule to apply would be `(p ↦ v ∧* Q) h ⟹ the (h p) = v`. The idea is similar to above, but this time we extend Isabelle's substitution tactic to automatically solve the side condition of the substitution by rotating conjuncts appropriately after applying the equality. It is important for this to happen atomically to the user, because the equality will instantiate the rule only partially in `h` and `p`, while the side condition determines the value `v`. Again, the tactic is generic, the rule comes from the instantiation.

The third situation is clearing context to focus on the interesting implication parts of a separation goal after heap update and value reasoning are done. The idea is to automatically remove all conjuncts that are equal in assumption and

conclusion as well as solve any trivial implications. The tactic `sep_cancel` that achieves this is higher-level than the tactics above, building on the same principles.

Finally, we supply a low-level tactic `sep_select n` that rotates conjunct `n` to the front while leaving the rest, including schematics, untouched.

With these basic tactics in place, higher-level special-purpose tactics can be developed much more easily in the future. The rule application and substitution tactics fully support backtracking and chaining with other Isabelle tactics.

One concrete area of future work in automation is porting Kolanski's machinery for automatically generating mapsto-arrow variants [5], e.g. automatically lifting an arrow to its weak variant, existential variant, list variant, etc, including generating standard syntax and proof rules between them which could then automatically feed into tools like `sep_cancel`. Again, the setup would be generic, but used to generate rules for instances.

4 Instantiations

We have instantiated the library so far to four different languages and variants of separation logic.

For the first instance, we took an existing example for separation logic that is part of the Isabelle distribution and ported it to sit on top of the library. The effort for this was minimal, less than an afternoon for one person, and unsurprisingly the result was drastically smaller than the original, because all boilerplate separation logic definitions and syntax declarations could be removed. The example itself is the classic separation logic benchmark of list reversal in a simple heap of type `nat ⇒ nat option` on a language with a VCG, deeply embedded statements and shallowly embedded expressions.

The original proof went through almost completely unchanged after replacing names of definitions. We then additionally shrunk the original proof from 24 to 14 lines, applying the tactics described above and transforming the script from technical details to reasoning about semantic content.

While a nice first indication, this example was clearly a toy problem. A more serious and complex instance of separation logic is a variant for reasoning about virtual memory by Kolanski [5]. We have instantiated the library to this variant as a test case, and generalised the virtual memory logic in the process.

The final two instantiations are taken from the ongoing verification of user- and kernel-level system initialisation in the seL4 microkernel [3]. Both instantiations are for a shallow embedding using the nondeterministic state monad, but for two different abstraction levels and state spaces. In the more abstract, user-level setting, the overall program state contains a heap of type `obj_id ⇒ obj option`, where `obj` is a datatype with objects that may themselves contain a map `slot ⇒ cap option` as well as further atomic data fields. In this setting we would like to not just separate parts of the heap, but also separate parts of the maps within objects, and potentially their fields. The theoretical background of this is not new, but the technical implementation in the theorem prover is nontrivial. The clear interface of separation algebra helped one of the authors with no prior

experience in separation logic to instantiate the framework within a week, and helped an undergraduate student to prove separation logic specifications of seL4 kernel functions within the space of two weeks. This is a significant improvement over previous training times in seL4 proofs.

The second monadic instance is similar in idea to the one above, but technically more involved, because it works on a lower abstraction level of the kernel, where in-object maps are not partial, some of the maps are encoded as atomic fields, and the data types are more involved. To use these with separation logic, we had to extend the state space by ghost state that encodes the partiality demanded by the logic. The instantiation to the framework is complete, and we can now proceed with the same level of abstraction in assertions as above.

Although shallow embeddings do not directly support the frame rule, we have found the approach of baking the frame rule into assertions by appending `∧* P` to pre- and post-conditions productive. This decision is independent of the library.

5 Conclusion

We have presented early work on a lightweight Isabelle/HOL library with an abstract type class for separation algebra and generic support for interactive separation logic tactics. While we have further concrete ideas for automation and for more type class layers with deeper support of additional separation logic concepts, the four nontrivial instantiations with productive proofs on top that we could produce in a short amount of time show that the concept is promising.

The idea is to provide the basis for rapid prototyping of new separation logic variants on different languages, be they deep or shallow embeddings, and of new automated interactive tactics that can be used across a number of instantiations.

Acknowledgements We thank Matthias Daum for his comments on this paper.

References

1. J. Bengtson, J. Jensen, F. Sieczkowski, and L. Birkedal. Verifying object-oriented programs with higher-order separation logic in Coq. In *ITP*, volume 6898 of *LNCS*, pages 22–38. Springer, 2011.
2. C. Calcagno, P. W. O'Hearn, and H. Yang. Local action and abstract separation logic. In *Proc. 22nd LICS*, pages 366–378. IEEE, 2007.
3. G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *22nd SOSP*, pages 207–220. ACM, Oct 2009.
4. G. Klein, R. Kolanski, and A. Boyton. Separation algebra. *Archive of Formal Proofs*, May 2012. http://afp.sf.net/entries/Separation_Algebra.shtml.
5. R. Kolanski. *Verification of Programs in Virtual Memory Using Separation Logic*. PhD thesis, School Comp. Sci. & Engin., Jul 2011.
6. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In M. Hofmann and M. Felleisen, editors, *34th POPL*, pages 97–108, Nice, France, Jan 2007. ACM.
7. T. Tuerk. A formalisation of smallfoot in HOL. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, LNCS, pages 469–484. Springer, Aug 2009.