Extensible Specifications for Automatic Re-Use of Specifications and Proofs

Daniel Matichuk[1] and Toby Murray[1,2]

[1] NICTA, Sydney, Australia*
[2] School of Computer Science and Engineering, UNSW, Sydney, Australia
{firstname.lastname}@nicta.com.au

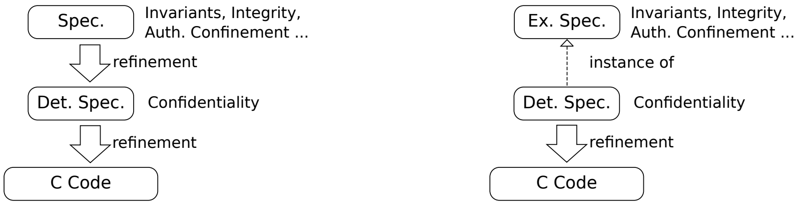
Abstract. One way to reduce the cost of formally verifying a large program is to perform proofs over a specification of its behaviour, which its implementation refines. However, interesting programs must often satisfy multiple properties. Ideally, each property should be proved against the most abstract specification for which it holds. This simplifies reasoning and increases the property's robustness against later tweaks to the program's implementation. We introduce *extensible specifications*, a lightweight technique for constructing a specification that can be instantiated and reasoned about at multiple levels of abstraction. This avoids having to write and maintain a different specification for each property being proved whilst still allowing properties to be proved at the highest levels of abstraction. Importantly, properties proved of an extensible specification hold automatically for all instantiations of it, avoiding unnecessary proof duplication. We explain how we applied this idea in the context of verifying confidentiality enforcement for the seL4 microkernel, saving us significant proof and code duplication.

1 Introduction

Formally verifying real software is expensive: proving a single property of a program's implementation can require an order of magnitude more effort than to write the implementation [4, 5]. To avoid expending this much effort on *every* property to be proved of an implementation, it is common to construct an abstract specification for the software and prove that the software's implementation formally *refines* this specification. While this is expensive, subsequent reasoning can then be performed over the abstract specification. In practice, such proofs can require only a similar amount of effort as that to write the implementation [5].

The verification of the seL4 microkernel [4] provides a useful data-point, being to our knowledge the most extensive code-level verification ever performed of a general-purpose software artifact. A microkernel is a minimal operating system kernel; seL4 implements services such as threads, virtual address spaces, IPC, and capability-based access control. An initial proof of refinement between the kernel's

* 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



(a) Without Extensible Specification (b) Using an Extensible Specification

Fig. 1. Proving Confidentiality for seL4

C implementation and an abstract specification of its behaviour consumed about 25 person-years of effort [4]. This produced a proof on the order of 200,000 lines of Isabelle/HOL, including proving the basic kernel invariants for the abstract specification. However, subsequent proofs of security properties, namely integrity and authority confinement [9], have been carried out on the abstract specification, making use of these invariants, and then transferred to the C implementation via refinement. These proofs were completed in under 10 person months [9].

Reusable, general purpose software systems, such as operating system (OS) kernels, must often satisfy multiple properties. For instance, a secure OS kernel like seL4 should enforce not only integrity and authority confinement, but also confidentiality and availability. Unfortunately, writing a specification that captures all of these properties is tricky: one can reason about integrity in the presence of nondeterminism, but doing so with confidentiality under refinement is much harder because nondeterministic specifications tend to have insecure refinements [8].

seL4's abstract specification, where integrity and authority confinement were proved, is nondeterministic. Under-specification of seL4's scheduling behaviour is a major source of nondeterminism in this specification. The scheduling routine is maximally nondeterministic, saying only that the kernel can either (nondeterministically) schedule some runnable thread, or schedule the idle thread. Thus any sensible scheduling algorithm is a valid refinement of this scheduling specification. However, this includes malicious scheduling algorithms that might leak information via their scheduling decisions, perhaps by choosing the next thread to schedule by examining some secret information. For this reason, confidentiality cannot be proved about this specification, because it abstracts away from details that are relevant to confidentiality. We must therefore prove confidentiality of a less nondeterministic specification that, for instance, precisely specifies the kernel's scheduling behaviour. We call this specification, the *deterministic* specification. This situation is depicted in Figure 1(a).

This might suggest that the initial abstract specification for seL4 was too nondeterministic. However, proving the kernel invariants, integrity and authority confinement at this level was the right thing to do. This is because, not only is a more abstract specification easier to reason about but, by proving these properties at a more abstract level, they become far more resilient to changes in the kernel's design and implementation, and we can still derive these properties for the deterministic specification by refinement. Specifically, having proved these properties about the nondeterministic specification, we can conclude that they hold for *all* possible refinements, which include all sensible scheduling implementations for instance. If we need to tweak an implementation detail of the scheduler, we need not fear that doing so will break these properties. The same does not necessarily follow if we have proved them only about the deterministic specification, which captures the precise scheduling behaviour.

This suggests that properties should be proved at the most abstract level at which they still hold. However, in the worst case, each property would require its own specification, as well as associated proofs of refinement between them. Any changes to the most abstract specification, such as an API evolution, must be reflected in all other specifications, and all proofs updated. This is expensive when APIs continually evolve and specifications duplicate much of each other's code, as happens with the nondeterministic and deterministic seL4 specifications.

In this short paper, we present *extensible specifications*, a technique for constructing specifications that avoids these problems while still allowing properties to be proved at the highest levels of abstraction. This technique is designed primarily for very large mechanical proof efforts, with large bodies of existing proofs and specifications, where duplicating existing artifacts and performing and maintaining unnecessary proofs is undesirable. These ideas have been developed and formalised within the proof assistant Isabelle/HOL; however, they should be applicable to any other proof assistant for higher order logic.

2 Extensible Specifications

The seL4 specifications are formalised as nondeterministic state monads [2] and we explain extensible specifications with reference to this formalism.

An imperative program may be specified as a nondeterministic state monad by defining a *state type* that is a record containing a field for each global variable in the program and each relevant piece of global state (such as the state of external devices with which the program interacts). The program is then a function that takes one of these records as its input and yields an updated record and a return value as its output. Nondeterministic computations yield a set of such outputs. Traditional "do-notation", as supported by Haskell for instance, is used to phrase monadic specifications in an imperative style.

A Running Example We use a toy example program to motivate and explain extensible specifications throughout this section. This program contains two functions of interest, called `alloc` and `dealloc` whose signatures are:

```
int alloc(void);    void dealloc(int i);
```

`alloc` takes no arguments and allocates a new resource, returning an integer ID to the allocated resource. `dealloc` takes an ID as its argument that represents an allocated resource, and deallocates the resource if it is currently allocated, and does nothing otherwise. For simplicity, `alloc` is allowed to fail if there are no resources to allocate. Figure 2 depicts abstract specifications of these functions.

```
alloc-abs ≡ do                                dealloc-abs $i$ ≡ do
    $ids$ ← gets ids;                             $ids$ ← gets ids;
    assert ($ids \neq \emptyset$);                when ($i \notin ids$)
    $i$ ← select $ids$;                            (modify (ids-update ($\lambda f.\ f \cup \{i\}$)))
    modify (ids-update ($\lambda f.\ f - \{i\}$));   od
    return $i$
od
```

Fig. 2. An example abstract specification.

The state type for these specifications is a record that contains a single field, ids, which holds a set of integers representing the IDs of currently free resources.

Given a field name x, gets x reads from the state record the value stored in field x, while modify (x-update *func*) updates the x-field of the state record with the result of running the function *func* on the current value stored in that field. Given a set S, select S nondeterministically selects a value from S. Hence, alloc-abs first reads the set of free IDs and asserts that it is not empty. It then nondeterministically selects from this set the next ID to allocate, before updating the set of free IDs in the state record by removing the chosen ID from it. Finally alloc-abs returns the chosen ID to its caller.

It is straightforward to show certain correctness conditions about these functions. For instance, we can prove that, whenever it is successful, alloc-abs returns the unique resource ID i that is now allocated but was originally free. This statement may be written in a monadic Hoare logic variant [2] as:

$$\{\!|\lambda s.\ \mathsf{ids}\ s = X|\!\}\ \mathsf{alloc\text{-}abs}\ \{\!|\lambda i\ s'.\ i \notin \mathsf{ids}\ s' \wedge \mathsf{ids}\ s' \cup \{i\} = X|\!\} \tag{1}$$

This statement is read as follows: if, before alloc-abs executes from some pre-state s, ids $s = X$, then whenever it terminates, returning a result i in some post-state s', $i \notin$ ids $s' \wedge$ ids $s' \cup \{i\} = X$.

alloc-abs completely abstracts away from the order in which resources are allocated. Reasoning about this order requires a more concrete specification of `alloc`'s behaviour. Suppose `alloc` is implemented by having it maintain a list of unused resource IDs, from which it selects the first item (and fails when this list is empty). Figure 3 depicts a hypothetical concrete specification of this behaviour. This specification operates on a state record that extends the original by adding a new field ids-list that contains a list of currently free resource IDs. The original set ids is retained to allow existing properties like (1) to be phrased over this new specification. While not really necessary in this example, this is vital for larger specifications with a massive body of existing proof, such as seL4.

Extensible Specifications Notice that much of the code in Figure 3 is duplicated from Figure 2. Also, proving an analogous result to (1) for alloc-conc requires re-proving it directly or concluding it from a proof of refinement between alloc-abs and alloc-conc. For large specifications, either approach requires significant effort.

By defining a single *extensible specification* for `alloc` that subsumes both alloc-abs and alloc-conc (and doing the same for `dealloc`), we can avoid this

```
alloc-conc ≡ do
    $ids\text{-}list$ ← gets ids-list;
    assert ($ids\text{-}list \neq []$);
    $i$ ← return (hd $ids\text{-}list$);
    modify (ids-update ($\lambda f.\ f - \{i\}$));
    modify (ids-list-update tl);
    return $i$
od
```

```
dealloc-conc $i$ ≡ do
    $ids$ ← gets ids;
    when ($i \notin ids$)
     (do
          modify (ids-update ($\lambda f.\ f \cup \{i\}$));
          modify (ids-list-update ($\lambda l.\ i \cdot l$))
      od)
od
```

Fig. 3. A hypothetical concrete specification.

```
alloc-ext $select\text{-}ext$ $update\text{-}ext$ ≡ do
    $ids$ ← gets ids;
    $more$ ← gets more;
    $next\text{-}ids$ ←
      return ($select\text{-}ext$ $ids$ $more$);
    $g\text{-}ids$ ←
      return (guard-set $ids$ $next\text{-}ids$);
    assert ($g\text{-}ids \neq \emptyset$);
    $i$ ← select $g\text{-}ids$;
    modify (ids-update ($\lambda ids.\ ids - \{i\}$));
    modify (more-update ($update\text{-}ext$ $i$));
    return $i$
od
```

```
guard-set $ids\text{-}set$ $next\text{-}ids\text{-}set$ ≡
if $next\text{-}ids\text{-}set \subseteq ids\text{-}set \wedge next\text{-}ids\text{-}set \neq \emptyset$
then $next\text{-}ids\text{-}set$ else $ids\text{-}set$

dealloc-ext $i$ $update\text{-}ext$ ≡ do
    $ids$ ← gets ids;
    when ($i \notin ids$)
     (do
          modify (ids-update ($\lambda f.\ f \cup \{i\}$));
          modify (more-update ($update\text{-}ext$ $i$))
      od)
od
```

Fig. 4. An example extensible specification.

effort. A specification is made extensible by augmenting its state type with an extra *extended state* component of arbitrary type, and inserting carefully placed *extended computation* points in its code, which are placeholders for code that operates on the extended state. Extended computations are placed (1) where the extended state is read to resolve nondeterminism that abstracts away implementation choices, and (2) where extended state needs to be updated to ensure consistency with the program's implementation. By instantiating the extended state with a concrete state type and the extended computations with specification code that operates over this state, one produces an *instance* of the extensible specification.

Figure 4 depicts extensible specifications for `alloc` and `dealloc`. As we will show, each may be instantiated to behave like the abstract and concrete specifications above, avoiding the duplication between them.

Importantly, we can also prove properties about these extensible specifications. Such properties hold for *all* instances of these specifications. This avoids proof duplication and/or having to maintain refinement proofs between specifications like alloc-abs and alloc-conc. We can easily rephrase (1) to be over alloc-ext:

$$\{\!|\lambda s.\ \mathsf{ids}\ s = X|\!\}\ \mathsf{alloc\text{-}ext}\ \mathit{select\text{-}ext}\ \mathit{update\text{-}ext}\ \{\!|\lambda i\ s'.\ i \notin \mathsf{ids}\ s' \wedge \mathsf{ids}\ s' \cup \{i\} = X|\!\}$$

Due to the similarity between alloc-ext and alloc-abs, the proof of (1) is easily adapted to alloc-ext. This is true for any property of the original specification.

Because it holds for all instances of alloc-ext, the above Hoare triple automatically applies to the instantiations shown below for alloc-abs and alloc-conc.

alloc-ext and dealloc-ext operate over a state type that extends the original state type with an extra field, more, of arbitrary type. They are parametrised by subroutines that perform extended computation (*select-ext* and *update-ext*). While we have included them in this paper for ease of presentation, passing the extended computations as parameters can be avoided by defining them as abstract operations on a *type-class* that the extended state implements. This approach should also work for other higher order logic proof assistants, like Coq.

We obtain alloc-ext from alloc-abs by adding two blocks of extended computation. The first reads the extended state and uses it to help resolve the nondeterministic selection of the next ID to allocate, while the second updates the extended state following this selection.

alloc-ext is carefully constructed so that it behaves like alloc-abs with respect to the non-extended state. alloc-ext calls the extended computation *select-ext* to obtain the set of IDs from which to subsequently select the ID to allocate. Importantly, alloc-ext then makes use of the function guard-set whose purpose is to force alloc-ext to behave like alloc-abs no matter what *select-ext* did: guard-set ensures that the subsequent assert fails only when ids is empty and the subsequent select always chooses an ID from ids. guard-set does this by checking that the set chosen by *select-ext* is a non-empty subset of ids and replacing it by ids if it is not. By building alloc-ext this way, we ensure that any property unrelated to the extended state that alloc-abs satisfies also holds for all instantiations of alloc-ext.

The original specifications of Figure 2 can be recovered by instantiating the extended state to be of type *unit* (the type with one element), and the extended computations to be no-ops. For alloc-ext, we have *select-ext* return the entire set ids, which ensures that the subsequent select behaves exactly like the nondeterministic select in alloc-abs. Finally, we have *update-ext* leave the extended state unchanged. Hence alloc-abs $=$ alloc-ext ($\lambda ids\ more.\ ids$) ($\lambda i\ more.\ more$) and dealloc-abs i $=$ dealloc-ext i ($\lambda i\ more.\ more$).

We can also instantiate alloc-ext to behave like alloc-conc from Figure 3, to reason about the concrete behaviour of `alloc`. We instantiate the extended state to include a list of currently unallocated resource IDs. We define the functions ids-list-ext, which reads this list from the more field, and ids-list-update-ext, which updates it; we omit these definitions for brevity. We then have *select-ext* return the singleton set containing the head of this same list, and have *update-ext* modify this list by replacing it with its tail (i.e. by removing its first item).

alloc-ext will behave deterministically, by performing the select from the singleton set given by *select-ext*, so long as this set is contained in ids, to prevent guard-set causing selection from the entirety of ids. alloc-ext behaves the same as alloc-conc in this case so long as ids-list is empty if and only if ids is. The invariant valid-list ensures this, and is trivial to prove of the deterministic instantiation.

valid-list $s \equiv$ distinct (ids-list s) $\wedge$ set (ids-list s) $=$ ids s

It states that each ID in ids-list is distinct, and each ID in ids-list is in ids and vice-versa. Under this invariant, the instantiated extensible specification exhibits the hypothetical concrete behaviour precisely. Specifically, $\mathsf{valid\text{-}list}\ s \longrightarrow \mathsf{alloc\text{-}conc}\ s = \mathsf{alloc\text{-}ext}\ (\lambda \mathit{ids}\ \mathit{more}.\ \{\mathsf{hd}\ (\mathsf{ids\text{-}list\text{-}ext}\ \mathit{more})\})\ (\lambda\text{-}.\ \mathsf{ids\text{-}list\text{-}update\text{-}ext}\ \mathsf{tl})\ s$ and $\mathsf{dealloc\text{-}conc}\ i = \mathsf{dealloc\text{-}ext}\ i\ (\lambda i.\ \mathsf{ids\text{-}list\text{-}update\text{-}ext}\ (\lambda l.\ i{\cdot}l))$.

These kinds of invariants, which assert consistency between the instantiated extended state and the non-extended state, are commonly required for reasoning that a concrete instantiation of an extensible specification behaves correctly with respect to the extended state. They are also the same kind of invariants required to prove refinement between $\mathsf{alloc\text{-}abs}$ and $\mathsf{alloc\text{-}conc}$, for instance; although extensible specifications avoid the need for this additional refinement proof.

3 Proving Confidentiality for seL4

We now explain how we applied extensible specifications to assist proving confidentiality of the seL4 microkernel. Our requirement for proving confidentiality was having a deterministic specification, with proofs of the thousands of lemmas that establish the correctness of the individual kernel functions, and proofs of the kernel invariants, integrity and authority confinement on this specification. These results have already been proven for seL4's nondeterministic abstract specification. Extensible specifications allow us to obtain these results for the deterministic specification as well without unnecessary effort.

The abstract seL4 specification is approximately 5,500 lines of Isabelle/HOL, and the proofs of integrity and authority confinement, and the kernel correctness lemmas and invariants, comprise around 65,000 lines of Isabelle/HOL. However, nondeterminism is used to abstract away from implementation details in only three main places: the precise order in which hardware address space identifiers (ASIDs) are allocated, the order in which capabilities are recursively deleted during a *revoke* system call, and to abstract away the scheduling algorithm, as explained earlier. If we were to take the approach depicted in Figure 1(a) and manually define a deterministic abstract specification for seL4, it would duplicate around 98% of the abstract specification. We would also have to prove refinement between this new specification and the original, and maintain this proof going forward. This refinement theorem would have to be repeatedly applied to prove the thousands of correctness lemmas, as well as the kernel invariants, integrity and authority confinement for the deterministic specification.

We avoided these problems by altering about 2% of the abstract specification to make it extensible, as depicted in Figure 1(b) where the dashed arrow indicates that the deterministic specification is an instance of the extensible one. We repeated the process shown in Section 2 for each nondeterministic function in the specification. Section 2's example is a simplification of the cases for hardware ASID allocation and capability revocation: each of these involves replacing a nondeterministic selection with a deterministic one, based on some extra state that the deterministic specification must track. We are currently designing a confidentiality-preserving scheduler for seL4. Once complete, we will modify the current extensible specification to allow scheduling decisions to be implemented by extended computations. Rephrasing the invariants and correctness lemmas, authority confinement and integrity properties and adapting their proofs to the extensible specification altered only $\sim$1, 000 lines of Isabelle/HOL ($\sim$1.5%). These results then hold for the original specification and the deterministic one without further effort.

By making the seL4 abstract specification extensible, we have avoided duplicating tens of thousands of lines of proof and specification code, and performing unnecessary refinement proofs. This would have required significant effort, and made the resulting artifacts a nightmare to maintain as seL4's API evolves.

4 Related Work

The basic ideas of extensible specifications are certainly not new. The extended state, and its abstract extended operations, which parametrise alloc-ext for instance, define the interface of an abstract data type [7] that instances of the extensible specification implement. Extensible specifications also resemble a lightweight form of Aspect-Oriented Programming [3], where our concrete extended computations resemble *advices* and the sites at which they are placed resemble *join points*. When making a specification extensible, the appropriate join points are sites where extended state needs to be read to resolve nondeterminism, and sites where extended state needs to be updated.

While presented here in the context of state monads, extensible specifications also resemble mechanisms for specification and proof re-use within the B method (e.g. [1]) and Event-B (e.g. [10]). With these sorts of methods, a generic pattern can also carry assumptions that instances of it must meet in order to inherit results proved for the pattern. We can do likewise for extensible specifications by attaching assumptions to the type-class of the extended state. Monadic extensible specifications are arguably cleaner and simpler than these other methods, largely because they inherit the elegance and power of higher order logic for abstracting over extended computation.

Sophisticated verification systems that support automated stepwise refinement also offer similar benefits to extensible specifications. For instance, Chalice [6] allows the differences between two specifications to be encoded using *skeleton* syntax, and refinement between them automatically proved via SMT. This avoids duplicating code between specifications, and having an explicit refinement proof. We expect SMT could also be used to prove properties for the concrete specification already shown of the more abstract one. Unlike extensible specifications where proof re-use comes for free, here it relies on SMT solving.

5 Conclusion

We have presented extensible specifications, a lightweight technique for constructing specifications that can be instantiated and reasoned about at multiple levels of abstraction. By using extensible specifications one avoids having to write and maintain a different specification for each property being proved of a program, whilst still allowing properties to be proved at the highest levels of abstraction.

Properties proved of an extensible specification hold automatically for all instantiations of it, avoiding unnecessary proof duplication. This technique has been vital in assisting the ongoing proof of confidentiality for the seL4 microkernel, where it saved duplicating tens of thousands of lines of proof and specification code, and for maintaining these artifacts as the kernel has continued to evolve during this proof effort. Our experience applying extensible specifications to seL4 suggests that they are practically applicable and scale to real-world verification efforts.

Acknowledgements

Thanks to Gerwin Klein, David Greenaway and Mark Staples for valuable feedback on earlier drafts of this paper.

References

1. S. Blazy, F. Gervais, and R. Laleau. Reuse of specification patterns with the B method. In *Proceedings of the 3rd international conference on Formal specification and development in Z and B*, ZB'03, pages 40–57. Springer-Verlag, 2003.
2. D. Cock, G. Klein, and T. Sewell. Secure microkernels, state monads and scalable refinement. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *21st TPHOLs*, volume 5170 of *LNCS*, pages 167–182, Montreal, Canada, Aug 2008. Springer-Verlag.
3. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *ECOOP'97*, volume 1241 of *LNCS*, pages 220–242, 1997.
4. 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, Big Sky, MT, USA, Oct 2009. ACM.
5. G. Klein, T. Murray, P. Gammie, T. Sewell, and S. Winwood. Provable security: How feasible is it? In *13th HotOS*, pages 28–32, Napa, CA, USA, May 2011. USENIX.
6. K. R. M. Leino and K. Yessenov. Stepwise refinement of heap-manipulating code in Chalice, 2011. Unpublished manuscript, available at http://research.microsoft.com/en-us/um/people/leino/papers/krml211.pdf
7. B. Liskov and S. Zilles. Programming with abstract data types. *SIGPLAN Not.*, 9(4):50–59, Mar 1974.
8. T. Murray and G. Lowe. On refinement-closed security properties and nondeterministic compositions. In *8th AVoCS*, volume 250 of *ENTCS*, pages 49–68, Glasgow, UK, 2009.
9. T. Sewell, S. Winwood, P. Gammie, T. Murray, J. Andronick, and G. Klein. seL4 enforces integrity. In M. C. J. D. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *2nd ITP*, volume 6898 of *LNCS*, pages 325–340, Nijmegen, The Netherlands, Aug 2011. Springer-Verlag.
10. R. Silva and M. Butler. Supporting reuse of Event-B developments through generic instantiation. In *ICFEM '09*, volume 5885 of *LNCS*, pages 466–484. Springer-Verlag, 2009.