Lyrebird — Assigning Meanings to Machines

David Cock
NICTA[1] *& University of New South Wales*

Abstract

This paper presents work in progress on the Lyrebird framework, consisting of a language for specifying the programmer-visible behaviour of a processor and its associated devices, a tool for automatically producing a fast simulator, and a formal semantic interpretation providing a machine model for use in an interactive theorem prover. Machine specifications are modular, providing abstract interfaces and structural parameterization (MMU-less processors, for example). Also presented is a specific example: An instantiation for the ARM1136jf-s core.

1 Introduction & Background

This work arose from the L4.verified [16] project that provided a functional correctness proof for the seL4 microkernel [11, 12]. The result of the project was a proof of correspondence between an abstract kernel specification and a C language implementation executing on a idealized 'C machine'. The abstract and C models share several simplifying assumptions about the underlying machine: First that the kernel program is invariant, second that privileged data structures are distinct from user-visible memory and third that machine-management operations (timer setup, TLB management, ...) are primitive operations distinct from the ordinary instruction set. The C model makes the additional assumption that the runtime stack is unbounded and distinct from addressable memory.

The above assumptions are all implicit: There is simply no way to express program update or stack addressing in the framework. Any further instantiation (in particular a more complete machine model) must respect these assumptions to be acceptable. With the exception of machine-management, these assumptions are all expressed as *invariants* on the execution state, which must be preserved by implementation. The kernel model makes several explicit assumptions about the machine, which we have argued informally are sufficient to satisfy the implicit assumptions we make in reasoning. These are that no write by user-level code ever modifies kernel data, and that no write, by user or kernel, ever modifies the kernel program. We justify this in turn by showing that the kernel data is never mapped to a user process, and that the kernel program is only ever mapped read-only. We need a formal model of the hardware that justifies that these precautions (correct mapping) ensure the safety properties (no writes to kernel data or program) and that this in turn implies that the kernel program is invariant. We propose a stack of low-level virtual machines building from a detailed hardware model to allow us to establish the desired properties in small steps.

The abstract-to-C simulation is formally verified and thus *correct*, yet its application to the seL4 kernel (implemented in C with small amounts of assembler) depends on the faithfulness of the abstract C machine (with opaque hardware manipulation calls) to the C runtime environment with hardware calls implemented in assembler. We need to show that the target machine (when configured correctly) implements the 'invariant program, distinct privileged data' abstract machine and that the (assembler) machine-management stubs implement the abstract machine operations. The validity of the proof rests on the accuracy of the machine model.

Having an accurate machine model is certainly necessary, yet it is by no means sufficient. In order to support such a large proof effort, the model needs to efficiently support the styles of reasoning that we apply, and to dovetail into our higher-level notions of simulation and refinement. The feasibility of the proof rests on the ease of reasoning about the model.

Finally, while the techniques employed may be generic, the ultimate proof statement will be that seL4 is correctly implemented on a specific machine (processor architecture, revision, configuration, ...). There is the temptation to target a machine amenable to reasoning, but used rarely (or not at all) in practice, thus limiting the real-word usefulness of the proof. At the other extreme, it is too easy to target a specific machine that happens to be available, with all its specific details and quirks and again limit the proof's usefulness. While for the sake of concreteness, it is probably necessary to pin down some details of the target (register width, hardware- vs. software- walked page tables, ...), it would be preferable to specify as little as possible about the underlying machine, and to write a generic proof that can be re-targeted with minimal effort (automatically by preference) to related architectures. The usefulness of the proof rests on the ease with which it can be instantiated (ported) to a specific target.

To achieve this, we need a model of the hardware that is sufficiently abstract to cover several machines, sufficiently precise to allow a complete proof on a given machine, easily re-targetable and extensively validated. To this end we present the Lyrebird specification language together with a simulator generation tool and propose a formal semantic interpretation. A particular specification contains the details necessary to complete the proof for a particular machine, and a stack of abstract virtual machines will provide an abstract hardware model to support the seL4 correctness proof. A new machine specification can be generated with modest effort by persons unfamiliar with formal logic, and the model can be validated using the generated simulator.

There are conflicting demands on our machine model. On the one hand, we need it to be precise, carefully constructed and well validated. On the other we don't want to tie ourselves to a particular machine; We want to port the model as easily as we do the code. We also need the simulator to be fast enough to be useful in development, yet be accurately synchronized with our formal model; We want to use the simulator to validate the model.

No existing technique meets all our requirements, hence the need for this work. A formal model on which we can reason is an absolute necessity, so we might consider a model in a formal logic (in our case Isabelle/HOL), however, constructing such a model requires a good working knowledge of the logical formalism; It would be preferable for the model to be written by someone with an intimate knowledge of the hardware, without requiring them to learn the logic.

The approach we have taken is to define a domain-specific language, Lyrebird, built from a small set of primitives (to which we will assign a formal interpretation), that is easily compiled to give a fast, portable simulator. The flexibility of the language comes from its simplicity and carefully chosen primitives, not from a broad set of specialized extensions. One great advantage of this approach is that the model and the simulator are synchronized by definition, as the model is simply the semantic interpretation of the specification. In addition, the generated simulator is completely portable and has as simple an interface as possible, so that it can easily be incorporated into larger simulations.

The Lyrebird framework was initially used to validate the kernel API before it had been implemented in C, and thus the initial model was for unprivileged execution only. We now propose to extend this framework to provide the abstract machine stack to support the kernel proof

This paper will present: The Lyrebird language (§2) together with an sample CPU model, an example of device modelling (§3), simulator generation (§4), formal modelling and the abstract machine stack (§5), the ARM1136jf-s instantiation (§6) and an overview of ongoing work (§7).

2 The Lyrebird Language

The Lyrebird specification language allows a precise, compact description of an instruction set. All *types* (§2.1) have a width and conversions are explicit. The basic storage unit is the *register* (§2.2), from which complex structures are built in a straightforward manner. Expressions are built with an expressive set of *operators* (§2.3), all of which have a straightforward combinational logic implementation.

Specification of behaviour is divided between pure, side-effect-free *functions* (§2.4) and state-transforming *macros* (§2.5) which are in turn built from imperative *fragments* (§2.5). Decoding is expressed as a pure function of the machine state, and the behaviour of an individual *instruction* (§2.7) is given by an imperative fragment.

Duplication of effort is avoided, and terseness improved by grouping instructions into *classes* (§2.8), which generally mirror the decode tree. Instruction classes define *aliases* (§2.6) for parameters common to instructions e.g. register operands, and common fragments of behaviour e.g. condition code updates.

Figure 1 gives the small, yet complete specification of a simple RISC processor. It features universal conditional execution (similar to ARM), multiple addressing modes and visible control registers, all of which are complications arising in real processors. The following discussion will refer to this example to illustrate the constructs presented.

```
module vsr;

registerclass R { index_size 5; entries 32; size 32; }
register  Instr { size 32; }

synonym SFR { PC= 31; SR= 30; }
synonym Gd  { NV= 0b000; AL= 0b001; ZE= 0b010; NZ= 0b011; }

function getZ: int<32> SR -> bool  SR[0]==1
function setZ: bool Z, int<32> SR -> int<32>
    Z :: SR[31:1]++1 || not Z :: SR[31:1]++0
function check_guard: int<3> g, int<32> SR -> bool
  g==Gd.NV :: false || g==Gd.AL :: true ||
  g==Gd.ZE :: getZ(SR) || g==Gd.NZ :: not getZ(SR)

interface Memory { int<32> addr; int<32> data;
  transaction Read  out { addr out; data in;  }
  transaction Write out { addr out; data out; }}

init { register(R, SFR.PC)<- 0; }

cycle { Memory.Read[[register(R, SFR.PC), Instr]];  decode_execute VSR; }

instructionclass VSR {
    guard= Instr[31:29]; opcode= Instr[28:23]; Ra= Instr[22:18]; Rb= Instr[17:13];

    pre_exec { if(not check_guard(guard, register(R, SFR.SR))) { abort; } }
    decode   { opcode[5:1]==0b11111: LoadStore; otherwise: Arithmetic; } }

instructionclass Arithmetic {
    I= Instr[12]; S= Instr[11]; Rc= Instr[10:6]; Immed= Instr[10:0];
    op_1= register(R, Rb); op_2= I==0 :: register(R, Rc) || I==1 :: Immed sext 32;

    post_exec {
      if(S==1) {
        register(R, SFR.SR)<- setZ(register(R, Ra)==0, register(R, SFR.SR)); }}
    decode    { opcode==0x00 : ADD; opcode==0x01 : SUB;
                opcode==0x02 : AND; opcode==0x03 : OR; } }

instruction ADD { execute { register(R, Ra)<- op_2+op_1; } }
instruction SUB { execute { register(R, Ra)<- op_2-op_1; } }
instruction AND { execute { register(R, Ra)<- op_2&op_1; } }
instruction OR  { execute { register(R, Ra)<- op_2|op_1; } }

instructionclass LoadStore {
    offset= Instr[10:0]; address= register(R, Rb) + (offset sext 32);
    decode { opcode[0]==0: LDR;  opcode[1]==1: STR; } }

instruction LDR { execute { Memory.Read [[address, register(R, Ra)]]; } }
instruction STR { execute { Memory.Write[[address, register(R, Ra)]]; } }
```

Figure 1: Very Simple RISC Specification

Typesetting Convention

In presenting language constructs, we will apply the following convention: $variables$ are in italics, $\mathrm{constants}$ in serif font, types in sans serif and **keywords** bold. In addition, a dot ($\cdot$) stands for a single missing lexical element and an ellipsis ($\ldots$) for the missing portion of a list.

2.1 Types

Figure 2: Hierarchy

All Lyrebird values are strongly typed as one of the following: Arbitrarily-sized integers, int; fixed-width integers, $\mathsf{int}\langle\mathrm{n}\rangle$ (for $\mathrm{n} \in \mathbb{N}^+$) and booleans, bool. Of these, only $\mathsf{int}\langle\mathrm{n}\rangle$ and bool are *realizable* (a valid *final* type for an expression), and only $\mathsf{int}\langle\mathrm{n}\rangle$ is *implementable* (a valid final type for a register). Overloading is allowed for built-in operators and is constrained by the type hierarchy in figure 2 ($\mathsf{int}\langle n\rangle$ is a fixed-width integer of as-yet-indeterminate width):

Type inference resolves the *initial* types of expression nodes to their *final* (realizable) types. Polymorphic operators have a more general initial type e.g. $+ :: \mathsf{int}\langle n\rangle, \mathsf{int}\langle n\rangle \to \mathsf{int}\langle n\rangle$ and a relation on operand types e.g. $+\!\!+ :: \mathsf{int}\langle m\rangle, \mathsf{int}\langle n\rangle \to \mathsf{int}\langle m+n\rangle$. Type annotations may be omitted where the correct type can be inferred from context.

2.2 Registers

Registers are the basic unit of storage, declared either globally (representing persistent state) or locally to an imperative fragment (representing transient storage). The syntax for global declarations (*fig.* 1, *l.* 3) is:

$$\textbf{register}\ R\ \{\ \textbf{size}\,\mathrm{n};\ \}$$

And for local declarations (*fig.* 4, *l.* 10):

$$\textbf{register}\ \mathsf{int}\langle\mathrm{n}\rangle\ R;$$

Globally, *register classes* (*fig.* 1, *l.* 3) are declared as follows:

$$\textbf{registerclass}\ C\ \{\ \textbf{index\_size}\,\mathrm{i};\ \textbf{entries}\,\mathrm{e};\ \textbf{size}\,\mathrm{n};\ \}$$

A register class is a set of e registers of width n indexed by $x \in \mathsf{int}\langle\mathrm{i}\rangle$ with $x < \mathrm{e} \leq 2^{\mathrm{i}}$.

2.3 Operators & Expressions

A wide range of operators are available to build expressions:

$$\begin{aligned}
+, -, *, /, \&, \text{¦}, \wedge &:: \mathsf{int}\langle n\rangle, \mathsf{int}\langle n\rangle \to \mathsf{int}\langle n\rangle \\
\%, \ll, \gg, \ggg &:: \mathsf{int}\langle n\rangle, \mathsf{int} \to \mathsf{int}\langle n\rangle \\
+\!\!+ &:: \mathsf{int}\langle m\rangle, \mathsf{int}\langle n\rangle \to \mathsf{int}\langle m+n\rangle \\
-, \sim &:: \mathsf{int}\langle n\rangle \to \mathsf{int}\langle n\rangle \\
[i:j] &:: \mathsf{int}\langle n\rangle \to \mathsf{int}\langle j-i+1\rangle \\
<, >, \leq, \geq, ==, != &:: \mathsf{int}\langle n\rangle, \mathsf{int}\langle n\rangle \to \mathsf{bool} \\
\textbf{and}, \textbf{or} &:: \mathsf{bool}, \mathsf{bool} \to \mathsf{bool} \\
! &:: \mathsf{bool} \to \mathsf{bool} \\
\textbf{sext}\ n, \textbf{zext}\ n &:: \mathsf{int}\langle m\rangle \to \mathsf{int}\langle n\rangle \quad \text{where } m \leq n
\end{aligned}$$

Of particular note are $+\!\!+$ (bitwise concatenation), $\cdot[\cdot:\cdot]$ (slice), $\ggg$ (sign-preserving arithmetic shift) and **sext** & **zext** (sign- and zero-extend). One fundamental operator not covered above is the guarded option expression $b_1 :: e_1 \| b_2 :: e_2 \| \ldots \| \textbf{otherwise} :: b_o$ which evaluates to e_n if $b_n \wedge \neg \bigvee_{i=1}^{n-1} b_i$ and b_o otherwise.

2.4 Functions

A function (*fig.* 1, *l.* 8) is defined by a tuple of input types and labels, a result type and an expression:

$$\textbf{function}\ F\colon \tau_1\ l_1, \ldots, \tau_n\ l_n \to \tau_r\ e(l_1, \ldots, l_n)$$

2.5 Fragments & Macros

There are two primitive imperative statements: *register transfer*, $r \leftarrow e$ and *abrupt termination*, **abort**. Register transfer (*fig.* 1, *l.* 17) stores the current result of expression e into the register named by r and abrupt termination (*fig.* 1, *l.* 24) immediately terminates the enclosing imperative fragment. Imperative fragments are built from register transfers by sequential composition, $\ldots;\ldots$ (*fig.* 1, *l.* 19); conditional execution, $\textbf{if}(b)\ \{\ldots\}\ \textbf{else}\ \{\ldots\}$ (*fig.* 1, *l.* 32) and looping on a previously declared local register, $\textbf{for}(r\ \textbf{from}\ m\ \textbf{to}\ n)\ \{\ldots\}$.

Macros (*fig.* 4, *l.* 9) are declared with a set of label bindings and a fragment:

$$\textbf{macro}\ M(\tau_1\ l_1, \cdot, \tau_n\ l_n)\ \{\ldots\}$$

Labels can be declared as *output labels* by prefixing with &. When a macro is instantiated (*fig.* 4, *l.* 13) ($M[\![e_1|r_1, \ldots, e_n|r_n]\!]$), each label is bound to either an expression (for input labels) or a register (for output labels). Semantically, a macro instantiation is equivalent to

inserting the macro fragment with labels replaced (taking care to avoid variable capture). These *semantic macros* are superior to textual macros principally by providing type checking and making input and output parameters explicit. Their principal application is in avoiding duplication of boilerplate e.g. error checking.

2.6 Aliases

Aliases (*fig.* 1, *l.* 5) give a name to an expression and are declared as $n = e$. The result of the expression is bound early, at the alias statement. Aliases are inherited through structural scopes and through the class hierarchy. Aliases in an outer scope can be obscured by those in an inner scope, but otherwise the semantics are SSA (static single assignment). Their principle application is defining instruction operands. The *synonym* keyword (*fig.* 1, *l.* 5) declares a namespace to qualify alias labels.

2.7 Instructions

Instructions (*fig.* 1, *l.* 35) are the core of the behavioural specification and consist of alias declarations together with an imperative *execute block*:

$$\textbf{instruction}\ C \left\{ \begin{array}{l} n = e\,; \\ \ldots \\ \textbf{execute}\,\{\ldots\} \end{array} \right\}$$

2.8 Classes

Instruction classes (*fig.* 1, *l.* 21) group instructions that share a common decode path, operands or behaviour. A class declaration consists of aliases, optional *pre-execute* and *post-execute* blocks and a *decode block*.

$$\textbf{instructionclass}\ C \left\{ \begin{array}{l} n = e\,; \\ \ldots \\ \textbf{pre\_exec}\,\{\ldots\} \\ \textbf{post\_exec}\,\{\ldots\} \\ \textbf{decode}\,\{b_1 : I_1\,; \ldots\} \end{array} \right\}$$

Where b_i is a boolean expression and I_i is the name of either an instruction or a class. The decode cases, $b_i : I_i$ (*fig.* 1, *l.* 25), are interpreted as for the guarded choice expression, determining the next branch in the decode tree according to a predicate on the state. The pre- and post-execute blocks modify any execute block reached by decoding from this class: The pre-execute block is prepended and the post- appended. This modification is cumulative if several classes on the decode path specify such blocks.

Instruction decoding is made fully explicit in the model, to allow reasoning down to the level of the hardware if desired, yet it can easily be abstracted by taking instructions as the primitive machine operations, rather than cycles.

2.9 Top-Level Behaviour

The top-level behaviour of the model is defined by the **cycle** $\{\ldots\}$ (*fig.* 1, *l.* 19) and **init** $\{\ldots\}$ (*fig.* 1, *l.* 17) blocks, specifying fragments implementing a single machine cycle and machine initialization, respectively. The only events typically visible external to the model are initialization, cycle and interface transactions, described next.

2.10 External Interfaces

External interactions with memory, devices or other CPUs are implemented as *transactions* over *interfaces*. An interface (*fig.* 1, *l.* 14) defines a *datapath* and a set of transactions (*fig.* 1, *l.* 15), each of which uses a datapath element as either an input or an output.

$$\textbf{interface}\ I \left\{ \begin{array}{l} \tau_1\ n_1\,; \\ \ldots \\ \textbf{transaction}\ T\,\{n_1\,\textbf{in}\,|\,\textbf{out}, \ldots\} \\ \ldots \end{array} \right\}$$

3 Modelling Devices

The Lyrebird language was originally designed with the goal of specifying instruction set behaviour. Thanks however to its regular structure, and focus on a few well-understood primitive operations (e.g. register transfer), it was easily applied to modelling both simple and relatively complex devices. Figure 3 shows the specification of a flat memory space, showing the flexibility of the register class construct. Figure 4 show a simple paging MMU driven by tables stored in the underlying memory. This not only gives an example of the *macro* (§2.5) construct, but despite its compactness captures many of the more subtle complications of such a device (for example, what happens when a write is made to a virtual address corresponding to the table entry defining that same address' mapping? This model gives a definitive answer, consistent with most real implementations).

These models also demonstrate the implementation of interface transactions, and the structural parameterization mentioned earlier. The CPU model could be instantiated with flat memory model, or with a paging MMU which is in turn instantiated on flat memory. We might then proceed to show that if the MMU-endowed machine is initialized appropriately, with words $[2^{16}, MemTop)$ mapped to $[0, MemTop - 2^{16})$ and $[0, 2^{16})$ unmapped, it is a faithful model of the flat memory machine with $MemTop' = MemTop - 2^{16}$. We could in the same

```
module memory; init {} cycle {} parameter MemTop; // In 32-bit words

registerclass M        { index_size 30; entries MemTop; size 32; }

interface CPU          { int<30> addr; int<32> data;
  transaction Read in  { addr in; data out; }
  transaction Write in { addr in; data in;  }}

transaction CPU.Write  { if(addr<MemTop) { register(M,addr)<- data; }}
transaction CPU.Read   { data<- addr<MemTop :: register(M,addr) || otherwise :: 0; }
```

Figure 3: Flat Memory Specification

```
module mmu; init {} cycle {}

interface CPU { int<30> addr; int<32> data;
  transaction Read  in { addr in; data out; }
  transaction Write in { addr in; data in;  }}

interface Mem { int<30> addr; int<32> data;
  transaction Read  out { addr out; data in;  }
  transaction Write out { addr out; data out; }}

macro Walk(int<30> va,int<30> &pa) {
    register int<32> table_entry; vpn= va[29:14];
    Mem.Read[[vpn zext 30,table_entry]]; pa<- table_entry[29:14]++va[13:0]; }

transaction CPU.Read  { register int<30> pa; %Walk(addr,pa); Mem.Read[[pa,data]];  }
transaction CPU.Write { register int<30> pa; %Walk(addr,pa); Mem.Write[[pa,data]]; }
```

Figure 4: Table-Driven MMU Specification

way show that a (von Neumann) machine with a unified instruction and data memory is a faithful model of a (Harvard) machine with separate instruction memory, if the MMU is initialized to map all instruction memory read-only. These are examples of a more general approach we hope to develop to allow the Lyrebird framework to provide our abstract machine stack, an example of which we will give in §7.

4 Generating a Simulator

The above language primitives have been carefully chosen to allow straightforward generation of an architectural simulator. The generator proceeds structurally over the abstract syntax tree, producing an intermediate C99 model which is in turn processed by the host platform's native compiler. The initialization and cycle events, together with any inbound transactions are exported as C procedures; Outbound transactions are exported as unresolved call references. The generated simulator module is pure C99 and is completely standalone. In this manner, the simulator module can easily be incorporated into a larger system. As an example, the framework provides automatically-generated Python bindings to exercise the model as an external module.

Despite the simplistic generation, the simulator achieves good performance ($\sim$ 10MIPS on a recent desktop machine). There is nonetheless plenty of scope for performance improvement, if desired.

As mentioned, the first application of the generated simulator was to provide a user-level ARMv6 environment to exercise a pure Haskell implementation of the seL4 kernel. [11] The integration was straightforward, with transactions bound (via the GHC foreign function interface) to stubs exported from the Haskell model.

5 Generating a Model

The principal novelty of the Lyrebird framework is that in addition to producing a simulator, it can just as easily be used to produce a formal model of the programmer-visible architecture. The most straightforward approach, and the one taken initially is to translate the specification to a model in a formal language (in this case Isabelle/HOL). This work is more fully described by Tsai [22], who compared the generated model to a hand-crafted one.

The downside of the translation approach is that the

semantics it ascribes to the specification are implicit in the translator, which is generally developed in a less rigorous manner, often in a scripting language (e.g. Python) for convenience. While this approach is certainly flexible and efficient (in programmer-hours), it introduces another gap in the verification chain, precisely what we wish to avoid. The goal is to validate the specification carefully against the target machine (this being a gap that may never be fully closed) and from the machine specification up to the system specification, leave no gaps. To this end, it is desirable for the specification language to be assigned a rigorous formal semantics, with as straightforward a translation as possible into a formal system (again, Isabelle/HOL for our work) with a view to verifying the translation itself, if desired. This is the subject of ongoing research, of which we will give a short sketch here.

To begin, we define a *machine* by a state type S and a set of valid programs P. Programs are interpreted as state transformers with failure, and may refer to unbound names. Thus for names in N, and values in V, the interpretation function I has type:

$$I\colon P \to (N \rightharpoonup V) \to S \to (S \oplus \bot)$$

Programs are built from primitive operations using several combinators: sequential composition, name binding and conditional execution. The only primitive operation in the base language is register transfer. The interpretation of an instruction is a program on the underlying machine, and decoding is a function $D\colon S \to P$, from state to program. The machine specification is thus interpreted as a set of programs (instructions) on an underlying machine (whose only primitive operation is register transfer), and a map from current machine state to one of these programs (decoding). We can now treat the specified processor as a machine on the same state, its primitive operations being its instructions, which are in turn the programs (P) of the underlying machine. Programs built from these new primitives represent the processor's machine language, with the decoding details abstracted (and the implicit assumption that the program is distinct from the machine state). If we now incorporate the decoding interpretation we have a further machine, again with a single primitive, *cycle* (or step), with interpretation:

$$I(Cycle) = \lambda s.\ I\big(D(s)\big)(s)$$

These models will all be derived by interpretation of our specification, and we can reason on any of the three levels as desired. This is the method I propose to use to construct the layered virtual machines, culminating in the machine model required by the kernel.

6 The ARM model

The initial model produced using the Lyrebird framework was of the unprivileged ARMv6 instruction set with the exception of Thumb mode and FPU/VLIW/DSP instructions. The CPU model is general enough to cover any ARMv6 processor, but where concreteness was essential we used the ARM1136jf-s core for reference. The application of the model was to to provide early validation of the (Haskell model of the) seL4 kernel [11] by allowing binary code to exercise the kernel interface via syscall emulation.

The model has since been extended to include pipeline effects, an abstract and a detailed MMU model and an AVIC (interrupt controller), EPIT (timer) and UART compatible with those of the Freescale i.MX31 SoC. Extending the model to include privileged instructions and additional processor modes is an ongoing project.

7 Ongoing Work

With the language and simulation framework in a mature state, there are numerous potential applications and fruitful open questions. The most important ongoing work is the development of a formal semantics (as sketched in §5). Once such a semantic model is developed, its most obvious application will be the verification of assembly code, an important part of the ongoing L4.verified project.

At least as important as assembly verification will be establishing a concrete, formal machine model to justify the abstractions underlying both the abstract and the C model (§1). The anticipated approach is to construct layered abstract machines, building from the detailed hardware model furnished by the specification together with its semantics (single memory, low-level machine management) to the desired abstract model (isolated invariant code, high-level machine-management primitives).

Less obvious, but equally important is the need to justify our verification condition for the kernel at the hardware level. The existing kernel verification establishes that the C implementation is behaviourally equivalent to its specification, a significant and essential first step. From the point of view of an application however, we want to view the kernel as simply a part of the underlying system, providing an abstract machine with isolated virtual address spaces and all hardware management performed through the appropriate system calls. To do so will require the semantics not only be compositional (surely an automatic requirement) but support a notion of abstraction and refinement (and modular composition) compatible with that in the higher proof layers.

The approach we propose is to phrase the desired behaviour as such an abstract machine, using the seman-

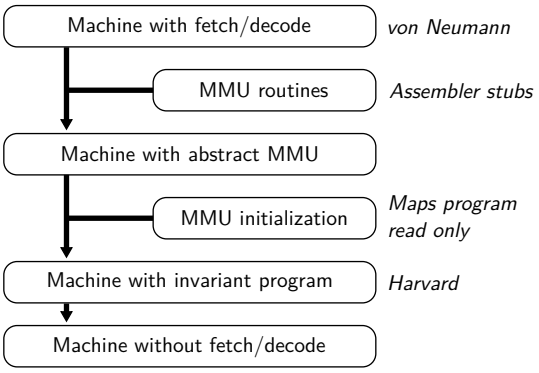

Figure 5: Abstract machine stack

tic model provided by the Lyrebird specification together with the abstract kernel model to prove that the underlying machine (suitably abstracted as above) together with the kernel implementation (in C with assembly) faithfully implements it. This is, in effect, a *reflection* of the kernel semantics across the machine interface to a give a user-level, rather than a kernel-level view of the system.

For the approaches suggested above to improve our satisfaction with the overall verification, the model will need to be carefully validated. At a minimum, the model should be shown to be consistent with others in the literature. [13] This will present a challenge, requiring reasoning with models produced in related but distinct logical formalisms (HOL4 & Isabelle/HOL).

The abstract machine stack that we propose is built both from Lyrebird specifications and more abstract specifications in HOL (e.g. the abstract kernel). Figure 5 shows the stack we propose to establish the invariant code property. The underlying (von Neumann) machine is extended with MMU routines and initialization to simulate a (Harvard) machine with an invariant program.

8 Related Work

There are many published formalizations of instruction sets. Bowen presented a complete formal model of the Motorola 6800 instruction set as early as 1987 [9]. Contemporary work includes that of Fox et. al. who have published a formalization of the ARMv6 instruction set [13] together with a multiprocessor memory model [4] and a Hoare logic for assembler [21]. The VAMP [7] was the first formally specified processor to be fully implemented. Approaches to instruction set models have also been proposed in the typed assembly language/proof-carrying code community [19, 3].

Lyrebird improves on existing formalisms by providing a specification language that is accessible to non-specialists while retaining a clear formalism, and allowing generation of flexible high-performance simulators.

Layered models are an established approach to building high-confidence systems. An example was developed for the VAX VMM project [18, 15]. This approach was advocated and put into practice by the Verisoft project [5, 10]. A memory layer for C was presented by Tuch et. al. [24, 23] as part of the L4.verified project and extended to handle virtual memory by Kolanski [17]. Lyrebird will allow us to extend this model down to the hardware level.

The canonical hardware description languages are VHDL [2] and Verilog [1]. These languages have several disadvantages for the style of modelling Lyrebird allows: They are large languages with steep learning curves and they model the system at a low level. As a result of their size, and having been developed prior to any formal models [20] rather than in parallel, they do not have a single natural semantic interpretation. SystemC [14] aims to address these shortcomings by allowing higher-level ('system-level') modelling. However, as SystemC is embedded within C++, it fails to adequately address the need for formalization (although steps in this direction have been proposed [25]). ArchC [6] is a modelling language built on SystemC that offers a similar specification style to Lyrebird but suffers from SystemC's lack of a formal basis. The M5 simulator [8] is portable to new architectures via a description language that provided some of the initial inspiration for this work. Lyrebird improves on the M5 ISA description by eliminating dependence on C/C++ code templates, instead describing all behaviour in the same language. Lyrebird will improve on existing simulators by having a clear formal interpretation.

9 Summary

We have presented the work to date, and work ongoing, on the Lyrebird framework for machine modelling. The specification language is simple, flexible and will be endowed with a natural semantic interpretation; It has already been assigned a semantics by translation. We hope to use this interpretation to establish a stack of virtual machines that establish the assumptions necessary for the seL4 correctness proof. The principal advantage of this approach is that the model and simulator are derived from a common specification, ensuring that they are consistent, and that the simulator can be used to validate the model, and hence the assumptions made by the kernel correctness proof.

Notes

[1]NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] Verilog register transfer level synthesis. *IEC 62142-2005 First Edition 2005-06 IEEE Std 1364.1* (Jun 2005), 1–116.

[2] IEEE standard VHDL language reference manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (Jan 2009), 1–626.

[3] AHMED, A., APPEL, A. W., RICHARDS, C. D., SWADI, K. N., TAN, G., AND WANG, D. C. Semantic foundations for typed assembly languages. *Trans. Progr. Lang. & Syst. 32*, 3 (2010), 1–67.

[4] ALGLAVE, J., FOX, A., ISHTIAQ, S., MYREEN, M. O., SARKAR, S., SEWELL, P., AND NARDELLI, F. Z. The semantics of power and ARM multiprocessor machine code. In *4th Workshop on Declarative Aspects of Multicore Programming* (New York, NY, USA, 2008), ACM, pp. 13–24.

[5] ALKASSAR, E., HILLEBRAND, M. A., LEINENBACH, D., SCHIRMER, N. W., AND STAROSTIN, A. The Verisoft approach to systems verification. In *VSTTE 2008* (Berlin, Heidelberg, 2008), Springer–Verlag, pp. 209–224.

[6] AZEVEDO, R., RIGO, S., BARTHOLOMEU, M., ARAUJO, G., ARAUJO, C., AND BARROS, E. The ArchC architecture description language and tools. *International Journal of Parallel Programming 33* (Oct 2005), 453–484.

[7] BEYER, S., JACOBI, C., KRÖNING, D., LEINENBACH, D., AND PAUL, W. J. Putting it all together – formal verification of the VAMP. *Int. J. Softw. Tools for Technology Transfer 8*, 4 (2006), 411–430.

[8] BINKERT, N. L., DRESLINSKI, R. G., HSU, L. R., LIM, K. T., SAIDI, A. G., AND REINHARDT, S. K. The M5 simulator: Modeling networked systems. *MICRO 26* (2006), 52–60.

[9] BOWEN, J. P. Formal specification and documentation of microprocessor instruction sets. *Microprocessing and Microprogramming 21*, 1–5 (1987), 223–230.

[10] DALINGER, I., HILLEBRAND, M., AND PAUL, W. On the verification of memory management mechanisms. In *Correct Hardware Design and Verification Methods*. 2005, pp. 301–316.

[11] DERRIN, P., ELPHINSTONE, K., KLEIN, G., COCK, D., AND CHAKRAVARTY, M. M. T. Running the manual: An approach to high-assurance microkernel development. In *ACM SIGPLAN Haskell WS* (Portland, OR, USA, Sep 2006).

[12] ELKADUWE, D., DERRIN, P., AND ELPHINSTONE, K. Kernel design for isolation and assurance of physical memory. In *1st IIES* (Glasgow, UK, Apr 2008), ACM SIGOPS, pp. 35–40.

[13] FOX, A. Formal specification and verification of ARM6. In *16th TPHOLs* (2003), pp. 25–40.

[14] GROTKER, T. *System Design with SystemC*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[15] KARGER, P. A., ZURKO, M. E., BONIN, D. W., MASON, A. H., AND KAHN, C. E. A retrospective on the VAX VMM security kernel. *Trans. Softw. Engin. 17*, 11 (Nov 1991), 1147–1165.

[16] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. seL4: Formal verification of an OS kernel. In *22nd SOSP* (Big Sky, MT, USA, Oct 2009), ACM, pp. 207–220.

[17] KOLANSKI, R., AND KLEIN, G. Types, maps and separation logic. In *22nd TPHOLs* (Munich, Germany, Aug 2009), S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds., vol. 5674 of *LNCS*, Springer, pp. 276–292.

[18] LEVY, H. M., AND LIPMAN, P. H. Virtual memory management in the VAX/VMS operating system. *IEEE Comp. 15*, 3 (Mar 1982), 35–41.

[19] MICHAEL, N. G., AND APPEL, A. W. Machine instruction syntax and semantics in higher order logic. In *17th International Conference on Automated Deduction* (London, UK, 2000), Springer–Verlag, pp. 7–24.

[20] MÜLLER, W., BÖRGER, E., AND GLÄSSER, U. The semantics of behavioral VHDL '93 descriptions. In *1994 European Design Automation Conference* (Los Alamitos, CA, USA, 1994), Comp. Soc. Press, pp. 500–505.

[21] MYREEN, M. O., FOX, A. C. J., AND GORDON, M. J. C. Hoare logic for ARM machine code. In *3rd International Conference on Fundamentals of Software Engineering*. 2007, pp. 272–286.

[22] TSAI, D. Formal model of an ARM microprocessor in Isabelle/HOL. Undergraduate thesis, School Comp. Sci. & Engin., University NSW, 2006.

[23] TUCH, H. Formal verification of C systems code: Structured types, separation logic and theorem proving. *JAR: Special Issue on Operating System Verification 42*, 2–4 (Apr 2009), 125–187.

[24] TUCH, H., KLEIN, G., AND NORRISH, M. Types, bytes, and separation logic. In *34th POPL* (Nice, France, Jan 2007), M. Hofmann and M. Felleisen, Eds., pp. 97–108.

[25] VARDI, M. Y. Formal techniques for SystemC verification. In *44th Design Automation Conference* (New York, NY, USA, 2007), ACM, pp. 188–192.