J Autom Reasoning (2018) 61:141–189
https://doi.org/10.1007/s10817-017-9446-0

Toward Compositional Verification of Interruptible OS Kernels and Device Drivers

Hao Chen[1,2] · **Xiongnan Wu**[2] · **Zhong Shao**[2] · **Joshua Lockerman**[2] · **Ronghui Gu**[2]

Received: 1 April 2017 / Accepted: 14 December 2017 / Published online: 23 December 2017

Abstract An operating system (OS) kernel forms the lowest level of any system software stack. The correctness of the OS kernel is the basis for the correctness of the entire system. Recent efforts have demonstrated the feasibility of building formally verified general-purpose kernels, but it is unclear how to extend their work to verify the functional correctness of device drivers, due to the non-local effects of interrupts. In this paper, we present a novel compositional framework for building certified interruptible OS kernels with device drivers. We provide a general device model that can be instantiated with various hardware devices, and a realistic formal model of interrupts, which can be used to reason about interruptible code. We have realized this framework in the Coq proof assistant. To demonstrate the effectiveness of our new approach, we have successfully extended an existing verified non-interruptible kernel with our framework and turned it into an interruptible kernel with verified device drivers. To the best of our knowledge, this is the first verified interruptible operating system with device drivers.

This is a revised and extended version of the conference paper [11] under the same title.

✉ Hao Chen
hao.chen@yale.edu

Xiongnan Wu
xiongnan.wu@yale.edu

Zhong Shao
zhong.shao@yale.edu

Joshua Lockerman
joshua.lockerman@yale.edu

Ronghui Gu
ronghui.gu@yale.edu

1 University of Electronic Science and Technology of China, Chengdu, Sichuan, China

2 Yale University, New Haven, CT, USA

Keywords Program verification · Certified OS kernels · Interrupts · Device drivers · Abstraction layer · Modularity

1 Introduction

An operating system (OS) kernel serves as the lowest level of any system software stack. The correctness of the OS kernel is the basis for that of the entire system. In a monolithic kernel, device drivers form the majority of the code base; 70% of the Linux 2.4.1 kernel are device drivers [12]. Furthermore, such drivers are found to be the major source of crashes in the Linux and Windows operating systems [8,12,19]. While recent efforts on seL4 [28] and CertiKOS [20] have demonstrated the feasibility of building formally verified OS kernels, it is unclear how to extend their work to verify the functional correctness of device drivers. In CertiKOS [20], drivers are unverified, and it is not obvious how to extend their framework to model devices and interrupts. In a microkernel like seL4 [28], device drivers are implemented in user space, and, though its proofs guarantee driver isolation, it does not eliminate bugs in its user-level drivers.

A major challenge in driver verification is the interrupt: a non-local jump to some driver code, triggered by a device. When device drivers are implemented inside the kernel (for better performance), the kernel should be interruptible; otherwise, it can lead to an unacceptable interrupt processing latency. Reasoning about interruptible code is particularly challenging, since every fine-grained processor step could contain a non-local jump, and, upon return, the machine state could be substantially changed. Even worse, it is not clear how such reasoning should be done at the C level, which is completely interrupt-unaware. Existing work either assumes that interrupts are turned off inside the kernel [20,34], or polls the interrupts at a few carefully chosen interrupt points [28].

Furthermore, interrupt hardware is not static, but is configured by software. In order to verify any interesting device drivers (serial, disk, etc.), we first need to model the interrupt controller devices (e.g., LAPIC [24], I/O APIC [23]), and formally verify their drivers. This is important because, if the interrupt controllers are not initialized properly, it may lead to undesired interrupt behaviors. Device drivers also interact with interrupt controllers to mask/unmask particular interrupt lines. These issues have been overlooked in past work, where interrupt controllers are assumed to be properly initialized and their drivers are correctly implemented [2].

Finally, verifying an interruptible operating system with device drivers also faces the following challenges.

Devices and CPU Run in Parallel Thus, the executions of CPU instructions and device transitions can interleave arbitrarily. Code verification on this highly nondeterministic machine can be challenging, since it needs to consider device state transitions, even when the CPU is executing a set of instructions unrelated to external devices. Recent work [1,2,4] tries to address this by enforcing a *stability* requirement that device states only change due to CPU operations. This requirement is, however, too strong as devices interacting with external environments are not stable: a serial device constantly receives characters through its port, a network card continuously transfers packets, an interrupt controller (IC) asynchronously receives interrupt requests, etc.

Devices may Directly Interact with Each Other Existing work assumes that a device driver monopolizes its underlying device and devices do not influence each other [2]. This assumption does not hold for many devices in practice. For example, most devices directly communicate with an interrupt controller by signaling an interrupt.

Device Drivers are Written in Both Assembly and C Existing device driver verification is either done completely at the assembly level [2,15] or the verified properties are only guaranteed to hold at the C level [37,38]. For realistic use-cases, proven properties should be translated down and then formally linked with the assembly-level proofs.

The Correctness Results of Different Components Should be Integrated Formally For example, the correctness proofs of device drivers and the OS kernel need to be formally linked as an integrated system, before one can deliver formal guarantees on the OS as a whole. Not doing so can introduce semantic gaps among different modules, a scenario which introduced actual bugs in previous verification efforts as reported by Yang and Hawblitzel [42]. Unfortunately, this formal linking process was found to be even more challenging than the correctness proofs of individual modules themselves [2]. Even OS's with user-level device drivers can suffer if the correctness proofs of their drivers are not formally linked with those of the kernel. For example, if some device driver code triggers a page fault at the user level, the behavior of the corresponding driver is linked to the behaviors of the page-fault handlers and address translation mechanism of the kernel.

In this paper, we propose a novel compositional approach that tackles all of the above challenges. There are two key contributing ideas. One is to build up a certified "virtual" device hierarchy, and the other is a new abstract interrupt model, built upon a realistic hardware interrupt model through contextual refinement. We use these to build an extensible framework that systematically enforces the isolation among different operating system modules, which is important for scalability of any verification effort and critical for reasoning about interruptible code.

Our paper makes the following new contributions:

- We present a new extensible architecture for building certified OS kernels with device drivers. Instead of mixing the device drivers with the rest of the kernel (since they both run on the same physical CPU), we treat the device drivers for each device as if they were running on a "logical" CPU dedicated to that device. This novel idea allows us to build up a certified hierarchy of extended abstract devices over the raw hardware devices, meanwhile, systematically enforcing the isolation among different "devices" and the rest of the kernel.
- We present a novel abstraction-layer-based approach for expressing interrupts, which enables us to build certified *interruptible* OS kernels and device drivers. Our formalization of interrupts includes a realistic hardware interrupt model, and an abstract model of interrupts which is suitable for reasoning about interruptible code. We prove that the two interrupt models are contextually equivalent.
- We present, to the best of our knowledge, the first verified interruptible OS kernel and device drivers that come with machine-checkable proofs. The implementation, modeling, specification, and proofs are all done in a unified framework (realized in the Coq proof assistant [40]), yet the machine-checkable proofs verify the correctness of the assembly code that can run on the actual hardware.

This paper presents a step (of our ongoing work) towards the formal verification of a faithful operating system, which contains certain limitations that could be explored as further research topics. First, the specification of hardware needs to be manually checked, but could be discharged in the future by the verification of device gate-level implementation. Second, concurrency caused by multi-processors and thread preemption are not discussed, thus need to be further investigated with a general shared-memory concurrency framework, such as

[21]. In addition, the framework does not support the verification of real-time behaviors of devices due to the lack of real-time notion in our machine model and events.

The rest of this paper is organized as follows. Section 2 briefly explains our abstraction-layer-based verification technology using a concrete example. Section 3 gives an overview on how we extend the layer-based framework to build our device hierarchy while enforcing isolation from the rest of the kernel. Section 4 defines a formal machine model extended with raw hardware devices. Section 5 presents the device objects, hardware interrupt model, and abstract interrupt model, and shows how we prove contextual refinement between the two interrupt models. Section 6 presents case studies of our verified drivers using the techniques developed in this paper. Section 7 describes our concrete Coq implementation in detail. Sections 8 and 9 give an evaluation of our new techniques and describe the lessons we learned, the limitations, and future work. Finally, we discuss related work and then conclude.

The artifact, including the specifications, the proofs, the framework, and automation library are available at http://flint.cs.yale.edu/certikos/publications/device/index.html for further references.

2 Overview of Certified Abstraction Layers

In this section, we give an overview of our abstraction-layer-based approach on verifying system software, first introduced in [20]. As in any other system verification, we associate every code module (a piece of code) a specification, and prove that the code meets its specification, or more formally, there is a *forward simulation* [32] from the module implementation to its specification. A specification of a module is a logical abstract representation of the module's behavior with the concrete implementation details hidden. For example, to specify operations on a doubly linked list stored in memory, we may logically interpret the complex in-memory data structure as a simple logical list and specify its *push* and *pop* operations as a simple list *append* and *remove* operations. To support this, the framework needs to provide a systematic way to hide the private memory state from its client, and replace them with *abstract states* to specify the full functionality of each operation in the interface in terms of the *abstract states*. Furthermore, a complex system like a kernel module is normally implemented in a combination of the C and assembly language. Thus, the framework should be able to be instantiated in both languages and provide a way to certifiably compile the C-based framework into the assembly-based one. The *certified abstraction layers* provide exactly such support.

2.1 C and Assembly Languages Used

Our framework supports both a C-like language and an x86 assembly language called Clight and LAsm, respectively.

Clight [10] is a subset of C and is formalized in Coq as part of the CompCert project [30]. Its formal semantics relies on a memory model [31] that is not only realistic enough to specify C pointer operations, but also designed to simplify reasoning about non-aliasing of different variables. From the programmer's point of view, Clight avoids most pitfalls and peculiarities of C such as nondeterminism in expressions with side effects. On the other hand, Clight allows for pointer arithmetic and is a true subset of C: valid Clight programs are valid C programs with the same semantics. Such simplicity and practicality turn Clight into a solid choice for certified programming. Furthermore, the CompCert verified compiler provides strong guarantees on code obtained by compilation of Clight programs. However, Clight provides little support for abstraction, and proving properties about a Clight program requires intricate reasoning about data structures. This issue is addressed by our layer infrastructure. Our Clight code is automatically generated from standard C code through a tool called clightgen provided by CompCert. We directly verify the generated Clight code. Thus, correctness of the clightgen does not affect the correctness of the verified code.

LAsm is a super set of the CompCert x86 assembly language with more machine-dependent registers and instructions needed for implementing low level system software.

2.2 Layer Interface

A layer interface L consists of the *abstract states*, *primitives*, a set of *invariants* on the abstract states, and *proofs* that all the primitives in the layer interface preserves the layer invariants. An *abstract state* could be a logical state that does not correspond to any physical state in the machine, but in most cases, it is a logical state that is abstracted from a concrete state in the registers or memory. Each *primitive* operates on the abstract states and is associated with an atomic specification. It is abstracted from a concrete, verified piece of the actual code. Since the *invariants* are preserved by all the primitives, the abstract states can only be accessed through calling one of the primitives, and execution of the primitives are atomic, the invariants hold at any moment during the system execution.

2.3 Code Module

A code module M corresponds to a concrete piece of code in Clight or LAsm assembly. Note that a module M implemented on top of a layer interface L may call any of the primitives defined in L. However, the standard Clight semantics is unaware of either the abstract states or the abstract primitives defined in the layer interface. While we would like to support the new abstract states and primitives, we seek to minimize the impact on the existing proof infrastructure for program and compiler verification. Thus, we do not modify the semantics of basic operations of Clight, but access the abstract states exclusively through the Clight's external function mechanism provided in CompCert. In addition, the external function mechanism is also used to model the interaction with the devices, such as input/output. Indeed, CompCert models compiler correctness through traces of events which can be generated only by external functions. CompCert axiomatizes the behaviors of external functions without specifying them, and only assumes they do not behave in a manner that violates compiler correctness. We use the external function mechanism to extend Clight with our primitive operations, and supply their specifications to make the semantics of external functions more precise. The semantics of LAsm is also instrumented accordingly to support the primitive calls in the assembly code. The verified Clight source code can be compiled by our extended CompCertX compiler [20] to the corresponding LAsm assembly in such a way that all proofs at the Clight level are preserved at the LAsm level. Then, the compiled LAsm modules and their proofs can be linked with the ones directly developed in LAsm.

2.4 Certified Layer

A *certified layer* is a new language-based module that consists of a triple (L_1, M, L_2) plus a mechanized proof object showing that the layer implementation M that is built on top of the layer interface L_1 (the *underlay interface*), denoted $[\![M]\!]L_1$, is a *contextual refinement* of the desirable layer interface L_2 above (the *overlay interface*), as shown in Fig. 1. A *deep specification* of L_2 captures everything *contextually observable* about running M over its

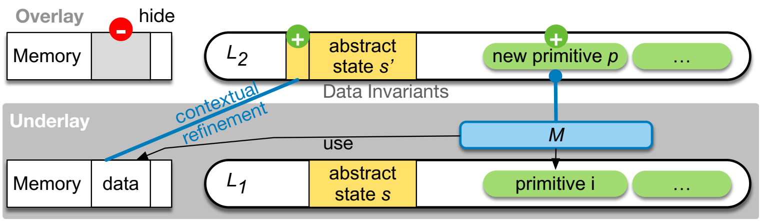

Fig. 1 Layer-based contextual refinement

underlay L_1. Once a certified layer (L_1, M, L_2) with its deep specification is built, there is no need to ever look at M again, since any property about M can be proven using L_2 alone.

The contextual refinement is proven by showing a forward simulation from L_2 to $[\![M]\!]L_1$ over a refinement relation. Thus, for every contextual refinement, we need to find a refinement relation R that can relate the system's states (including the abstract states) between the layer interface L_1 and L_2. In the above doubly linked list example, R needs to relate the in-memory doubly linked list of L_1 to the abstract logical list in L_2. In the case when there is no data abstraction between L_1 and L_2, R is simply an identity relation. To prove forward simulation, we need to prove that for every state (s_1, s_2) in R, and for every primitive p in L_2, if p takes the state from s_2 to s'_2, then there exists zero or more steps in M which can take the state s_1 to s'_1, where (s'_1, s'_2) is also in R.

In addition, the contextual refinement also needs to guarantee that the context code running on the overlay interface does not accidentally damage the underlay in-memory data by directly accessing the relevant memory. As shown in Fig. 1, we achieve this by utilizing the CompCert memory permissions [31] to hide the relevant memory region at overlay, which prevents the context code from accessing the relevant memory. These logical permissions do not correspond to any physical protection mechanism, but are used to ensure that the abstract machine at overlay gets stuck if any code tries to directly access this portion of memory. The safety proof of our entire system (the system never gets stuck) guarantees that such a situation never happens. Thus, the only way to affect the abstracted memory by any context code running over the overlay interface L_2 is to explicitly calling relevant primitives in L_2.

Traditionally, refinement is proven by an upward simulation from implementation to specification to ensure soundness, i.e., any property we prove on the overlay specification is guaranteed to hold at underlay implementation. In our framework, we also prove the upward simulation. But as a proof technique, we first prove the downward simulation and later turn it into an upward one using the fact that our machine semantics is deterministic with respect to the external events. We model all non-deterministic behaviors by encapsulating all potential non-deterministic aspects into event logs. And we prove our strong contextual refinement property with respect to all possible combinations of event logs. This is how we guarantee that our proof holds for all possible scenarios in the non-deterministic executions.

On the other hand, if the number of simulating underlay steps is not constrained, a downward simulation could potentially be fulfilled by bogus implementations, We always insist writing the deep specifications for the overlay interface to capture everything contextually observable for the underlay implementation. Thus, the form of our high level specifications at overlay is not some random high level relations vaguely restricting our underlay behaviors, but actual precise specifications of what should be the exact outcome after running corresponding abstract primitive, explained in terms of abstract states. Thus, under same preconditions enforced by the primitive specification, the behavior of overlay primitive and underlay implementation should be exactly the same over the same external events.

As in Fig. 1 and the following figures, we use $\boxed{L}$ to represent a layer interface, which consists of an abstract states s and a list of primitives ($\boxed{prim}$). The layer implementation is presented as $\boxed{M}$ which contains one or multiple LAsm or Clight functions. The contextual refinement relation of a module to a layer interface is denoted as M −● L, which requires to preserve the refinement relation (—) between the abstract states of two layers. We always put the more abstract layer on top of more concrete layers.

2.5 Verification of Clight and LAsm functions

Given that majority of the system software are developed in C (Clight in our case), we need a good framework-level automation support to verify that C modules meet their specifications. In our framework, this proof is achieved semi-automatically through Coq tactic libraries implemented in the Coq's tactical language L_{tac}. The primary proof tactic *cauto* consists of many components.

One main component is a verification condition generator that decomposes all the Clight expressions and statements, and produces conditions as sub-goals for the further expression evaluation and statement execution based on the big-step semantics defined in CompCert. The only exception is the loop, whose verification conditions cannot be generated by simply applying the semantic rules. We developed a separate logic for loops, which requires the user to provide the loop invariants that are preserved on every iteration of the loop execution. Our proof is termination sensitive. Thus, the logic also requires a termination metric, a well-founded order of the type of the provided metric, and a proof that the metric decreases at every iteration of the loop according to the provided well-founded order.

The language Clight strictly follows the C standard and disallows the undefined behaviors described in the standard C semantics. Thus, these all become the preconditions in the semantic rules of the Clight language. For a reasonably realistic C module, the set of verification condition generated is extremely large. Thus, discharging the conditions after they are fully generated would be very inefficient. Instead, the *cauto* tactic integrated many of the theory solvers to discharge the sub-goals on the fly as soon as they become provable.

First, to prevent the integer overflow, the Clight semantics requires every intermediate value in the middle of expression evaluations to be within the range regarding its type. In this way, most of the Clight code generates a huge set of arithmetic sub-goals for checking value ranges. However, the standard *omega* tactic is too weak to prove most of the goals. We have incorporated the *cauto* tactic a powerful arithmetic solver that can handle divisions, modular operations, bit-wise operations, machine finite precision integers, etc.

Clight semantics also utilizes partial maps and Coq lists to represent the local variable environments and arguments. Furthermore, we extensively use partial maps and Coq lists in the abstract states to abstract many of the concrete data structures in memory. To support those, the tactic contains theory solvers to discharge proof goals for properties related to partial maps and Coq lists. The tactic also contains a number of domain specific libraries which handle items such as device transitions and logs.

The automation library is easy to learn and use, and is exercised extensively by many students and researchers in our group to prove thousands of lines of C code in our verified OS kernel.

We have also developed an automation library to semi-automatically prove the modules directly implemented in LAsm. The automation support for LAsm is not as mature and powerful as the support for Clight, as the assembly code is much less structured in nature

```c
struct ConsoleBuffer {
 char buffer[CB_SIZE];
 unsigned int rpos;
 unsigned int wpos;
};

// in-memory circular buffer
struct ConsoleBuffer cons_buf;

// console buffer module M
void cons_buf_init() {
 cons_buf.rpos = 0;
 cons_buf.wpos = 0;
}

char cons_buf_read () {
 unsigned int rv = CB_EMPTY;
 if (cons_buf.rpos != cons_buf.wpos) {
  rv = cons_buf.buffer[cons_buf.rpos];
  cons_buf.rpos = (cons_buf.rpos + 1) %
     CB_SIZE;
 }
 return rv;
}

void cons_buf_write (char c) {
 cons_buf.buffer[cons_buf.wpos] = c;
 cons_buf.wpos = (cons_buf.wpos + 1) %
     CB_SIZE;
 if (cons_buf.rpos == cons_buf.wpos) {
  cons_buf.rpos = (cons_buf.rpos + 1) %
     CB_SIZE;
 }
}
```

Fig. 2 Console circular buffer implementation in C

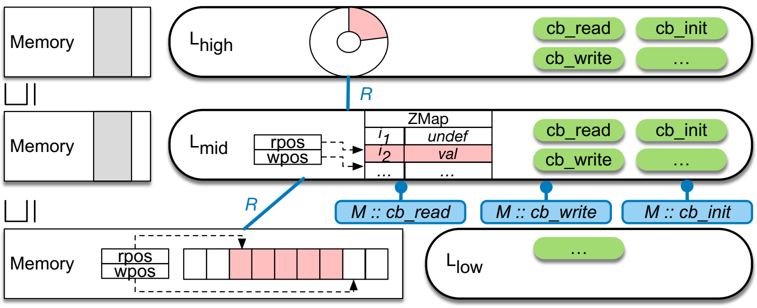

Fig. 3 The layer hierarchy of circular console buffer

compared to the Clight programs. In practice, it is not a big issue since the part of system code directly implemented in assembly is relatively small.

2.6 Example: Verification of Console Circular Buffer

To better illustrate how the certified abstraction layers work, in this subsection, we demonstrate how we can utilize the techniques to verify a console circular buffer implementation used in our verified device drivers. As shown in Fig. 2, in memory, the circular buffer is implemented as a circular array (to store the received input characters) with two additional fields (to mark the head (`rpos`) and the tail (`wpos`) of the circular buffer as shown in Fig. 3). Since the console buffer module M in Fig. 2 does not utilize any layer primitives, we can view that M could run on a layer interface L_{low} with empty abstract state and primitives.

Now we define a new layer interface L_{high} with an abstract state d abstractly representing the circular buffer, and a list of abstract primitives cb_init, cb_read, and cb_write that operate on the abstract buffer. First, we define $d = (\text{cons\_buf} : \text{list } \mathbb{Z})$, i.e., we simply abstract the in-memory circular buffer as a simple logical list. Next, we give specifications to the set of primitives as shown in Fig. 4. Here, we use inference rules ($\frac{Premise_1 \ \ Premise_2 \ \ldots}{Conclusion}$) to express the specification and the notations $[\cdot]$ and $++$ to represent a singleton list and list concatenation, respectively. The specification shown in Fig. 4 is much cleaner and simpler

Fig. 4 Specifications of abstract console buffer primitives

$$\frac{d' = d[\mathsf{cons\_buf} \leftarrow \mathsf{nil}]}{\mathsf{cb\_init}(d) = d'} \quad \text{(cb\_init)}$$

$$\frac{c :: tl = d.\mathsf{cons\_buf} \qquad d' = d[\mathsf{cons\_buf} \leftarrow tl]}{\mathsf{cb\_read}(d) = (d', c)} \quad \text{(cb\_read\_char)}$$

$$\frac{\mathsf{nil} = d.\mathsf{cons\_buf}}{\mathsf{cb\_read}(d) = (d, CB\_EMPTY)} \quad \text{(cb\_read\_empty)}$$

$$\frac{l = d.\mathsf{cons\_buf} \qquad \mathsf{length}\ l < CB\_SIZE \\ d' = d[\mathsf{cons\_buf} \leftarrow l{+}{+}[c]]}{\mathsf{cb\_write}(d, c) = d'} \quad \text{(cb\_write\_char)}$$

$$\frac{c :: tl = d.\mathsf{cons\_buf} \qquad \mathsf{length}\ l = CB\_SIZE \\ d' = d[\mathsf{cons\_buf} \leftarrow tl \mathbin{++} [c]]}{\mathsf{cb\_write}(d, c) = d'} \quad \text{(cb\_write\_overflow)}$$

than the actual implementation which simplifies future reasoning of code modules that use the data structure.

Next, we can define a refinement relation R to relate the concrete circular buffer in the memory and the abstract list, then prove the contextual refinement between $[\![M]\!]L_{low}$ and L_{high} over the simulation relation R. The forward simulation proof can be achieved on the primitive-by-primitive basis. One can imagine that due to the non-trivial R, this simulation proof is also complex.

The complexity may further explode when this logical complexity gets mixed with the complexity of handling the accesses to the CompCert memory. CompCert memory model is an axiomatized model where the properties are defined through a big list of axioms without a specific implementation. Any concrete implementation of this memory model needs to satisfy all the axioms. Thus, one cannot perform any simple evaluations on the memory, but needs to keep applying appropriate axioms to derive any desired properties. This severely limits the room for proof automation and significantly increases the proof size and memory consumption for proof compilation as the proof gets more complex. To separate the complexity that comes from the CompCert memory model from the actual complexity of the proof, in our layered approach, we always make the gap between the underlay and overlay interface as small as possible when it comes to data abstraction, i.e., when a piece of memory gets abstracted into an abstract state at the overlay interface (Fig. 3).

In the case of the circular console buffer, instead of directly jumping from the in-memory implementation to a logical list, we introduce an intermediate layer interface where the representation of the circular buffer in the abstract state is very similar to the one in the memory. We define the intermediate layer interface L_{mid} with the abstract state d and the primitive specifications as shown in Fig. 5. Here, for any type T, $\mathsf{ZMap}.t\ T$ is the type of partial map from integer keys to the values of type T. One can easily observe that the representations shown in Fig. 5 are extremely similar to the actual implementations shown in Fig. 2. Given the similarity, one can easily come up with a refinement relation R which maps the concrete values in the memory to their appropriate logical values in d. The simulation proof over R is also relatively easy and there is no other complex factors interfering with the ones from handling the CompCert memory.

Once the contextual refinement between $[\![M]\!]L_{low}$ and L_{mid} is proven, the contextual refinement between L_{mid} and L_{high} can be proven with no code module involved. Thus, this part of proof is completely logical and the refinement relation R_{cons_buf} (shown in Fig. 6)

$$\begin{array}{lll} d = (& \mathsf{cons\_buf\_concrete} : \mathsf{ZMap}.t\ \mathbb{Z}, & \triangleright\ \textit{Concrete console buffer} \\ & \mathsf{rpos} : \mathbb{Z}, & \triangleright\ \textit{The head of the buffer} \\ & \mathsf{wpos} : \mathbb{Z}). & \triangleright\ \textit{The tail of the buffer} \end{array}$$

$$\frac{d' = d[\mathsf{rpos} \leftarrow 0][\mathsf{wpos} \leftarrow 0]}{\mathsf{cb\_init}(d) = d'} \quad \text{(cb\_init)}$$

$$\frac{\begin{array}{c} i = d.\mathsf{rpos} \qquad i \neq d.\mathsf{wpos} \\ c = d.\mathsf{cons\_buf\_concrete}[i] \qquad d' = d[\mathsf{rpos} \leftarrow (i+1)\ \mathsf{mod}\ \mathit{CB\_SIZE}] \end{array}}{\mathsf{cb\_read}(d) = (d', c)} \quad \text{(cb\_read\_char)}$$

$$\frac{d.\mathsf{wpos} = d.\mathsf{rpos}}{\mathsf{cb\_read}(d) = (d, \mathit{CB\_EMPTY})} \quad \text{(cb\_read\_empty)}$$

$$\frac{\begin{array}{c} i = d.\mathsf{wpos} \qquad i' = (i+1)\ \mathsf{mod}\ \mathit{CB\_SIZE} \qquad d.\mathsf{rpos} \neq i' \\ d' = d[\mathsf{cons\_buf\_concrete}[i \mapsto c]] \qquad d'' = d'[\mathsf{wpos} \leftarrow i'] \end{array}}{\mathsf{cb\_write}(d,c) = d'} \quad \text{(cb\_write\_char)}$$

$$\frac{\begin{array}{c} i = d.\mathsf{wpos} \qquad i' = (i+1)\ \mathsf{mod}\ \mathit{CB\_SIZE} \qquad d.\mathsf{rpos} = i' \\ i'' = (i'+1)\ \mathsf{mod}\ \mathit{CB\_SIZE} \qquad d' = d[\mathsf{cons\_buf\_concrete}[i \mapsto c]] \\ d'' = d'[\mathsf{wpos} \leftarrow i'][\mathsf{rpos} \leftarrow i''] \end{array}}{\mathsf{cb\_write}(d,c) = d''} \quad \text{(cb\_write\_overflow)}$$

Fig. 5 Intermediate specifications of console buffer primitives

```
Fixpoint match_cons_buf (cons_buf: list Z) (cons_buf_concrete: ZMap.t Z) (rpos wpos:
    Z) : Prop :=
  match cons_buf with
   | nil => rpos = wpos
   | bv :: cons_buf' =>
       ZMap.get rpos cons_buf_concrete = bv /\
       match_cons_buf cons_buf' cons_buf_concrete ((rpos + 1) mod CB_SIZE) wpos
  end.

Inductive R_cons_buf: L_high.Abs -> L_mid.Abs -> Prop :=
| MATCH_CONS_BUF:
    forall d_high d_mid,
      match_cons_buf d_high.cons_buf d_mid.cons_buf_concrete d_mid.rpos d_mid.wpos ->
      R_cons_buf d_high d_mid.
```

Fig. 6 The definition of refinement relation between L_{high} and L_{mid} in Coq

in this case only needs to relate two sets of abstract states. In Fig. 6, Abs is the type of the abstract states in each layer interface. The overall layer hierarchy of entire console buffer is shown in Fig. 3. This kind of two-stage proof strategy significantly reduces the complexity of the proof and lifts the main complex proof effort to pure logical level.

3 Certified Abstraction Layers with Device Drivers and Interrupts

Instead of verifying an operating system from scratch, we start from an existing verified kernel that is developed in our group, named mCertiKOS. It is developed and verified using the abstraction layer-based approach illustrated in Sect. 2. The kernel currently runs on the 32 bit x86 architecture. It provides a multi-processing environment for user-level applications using separate virtual address spaces. It implements both message passing and shared memory

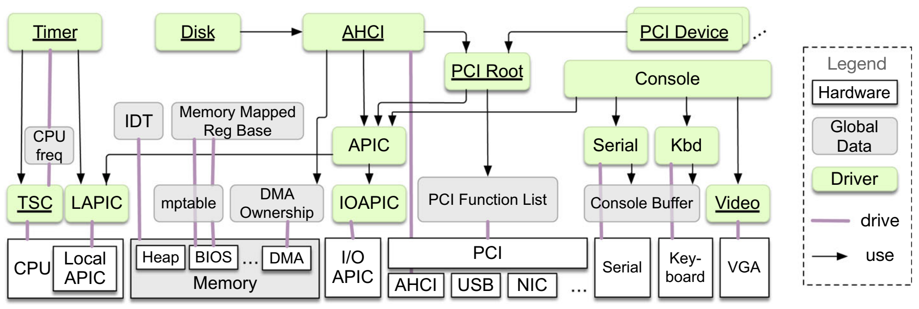

Fig. 7 The device driver hierarchy of mCertiKOS

inter-process communication protocols. As a hypervisor, it can also boot recent versions of unmodified Linux operating systems inside a virtual machine. Unlike large commercial operating systems like Linux or Unix, mCertiKOS kernel only implements a small subset of the POSIX-like API, e.g., process creation and control, physical and virtual memory management, and inter-process communication. It does not implement signals, pipes, etc. The current file system implementation in mCertiKOS is not verified.

Figure 7 shows the device hierarchy of mCertiKOS. Here the white boxes represent raw hardware devices; the green boxes denote the device drivers, and the gray boxes are the data structures used by the drivers. The purple/black lines show how these device and driver components are related. Note that the drivers in mCertiKOS are not verified; they are implemented in about 1600 lines of C and assembly code, and would be considered as part of the trusted computing base (if they are kept inside the kernel).

We take mCertiKOS's lowest level machine model, *LAsm*, and extend it with device models. We model devices as finite state transition systems interacting with the processor and the external environments. Since devices run concurrently with the processor, parts of the device state change without the processor explicitly modifying them. Though these "volatile" device states can change nondeterministically, the processor itself only ever observes a "current" state when it reads the device data via an explicit I/O operation. The processor does not, and in fact *cannot*, care about any states that the device may enter between these observed states. Therefore, instead of designing fine-grained small-step transition systems that model all possible interleaved executions amongst the processor and devices, our devices simply perform an atomic big-step transition whenever they are observed, i.e., when there is a device read/write operation from the CPU.

Next, the machine model needs to be extended with the hardware interrupt model. The processor responds to an interrupt by temporarily suspending the current execution and then jumping to another routine (i.e., an interrupt handler). Interrupts can be triggered by both hardware and software. Software interrupts (e.g., exceptions, system calls) are relatively easy to reason about, since their behaviors are always deterministic. For example, a page fault exception occurs whenever the accessed address belongs to an unmapped page or a page with wrong permission, and a system call is triggered by an explicit instruction. However, hardware interrupts (IRQs) are unpredictable; when we execute some code with interrupts turned on, at every fine-grained processor step, the machine state (e.g., registers and memory) may undergo significant changes. Recent work on verified operating systems (including mCertiKOS) neglects this kind of reasoning, ignoring one of the largest kernel threat-surfaces [4,20,27]. Finally, modeling interrupts is important because it also opens the way toward enabling interrupts within the kernel.

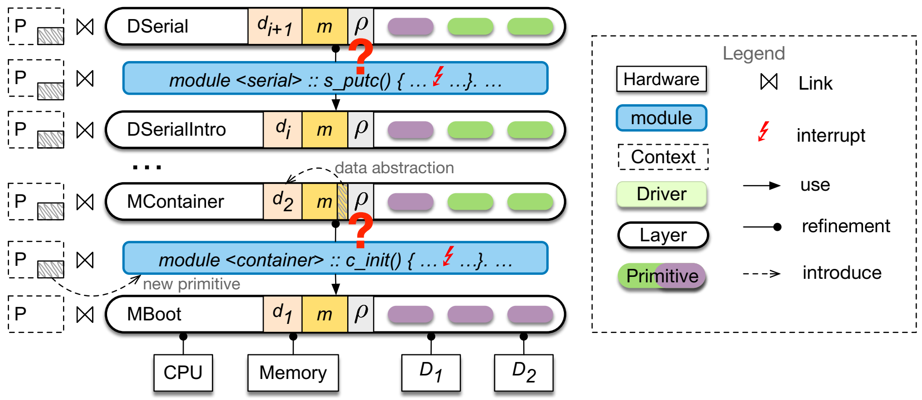

Fig. 8 Abstraction layers w. interrupts: a failed attempt

On top of this lowest-level machine model, each kernel module can be related to either device drivers (denoted as *DD*) or the rest of the kernel (denoted as *K*, representing non-device-related kernel components). To introduce, verify, and abstract each such kernel module into an abstract object with atomic logical primitive transitions, we need to prove the following isolation properties:

– For each function in *K* or user space, which has interrupts turned on, the interrupt must not affect the behavior of the function. Although the code can be interrupted at any moment, and the control flow transferred to a place outside the function, it will eventually return with states (which the function relies upon) unchanged.
– Devices which directly change the memory through Direct Memory Access (DMA), do not change any memory that the execution of any function in *K* depends on.
– For each interruptible device driver function in *DD*, any interrupt not related to the current device must not change any state related to the current device.
– In case that all interrupts related to a device are masked out, no interrupts can affect the state of the interrupt handler for the device.

For a particular fixed set of functions, the proof of the above properties may not seem hard. However, they have to be proven repeatedly for all possible combinations of currently introduced sets of functions and devices. This immediately makes the verification of an interruptible operating system with device drivers unscalable.

Furthermore, it is not obvious how to apply techniques presented in Sect. 2 to handle hardware interrupts. Figure 8 shows one such attempt. Here, P denotes the kernel/user-level context code; MBoot, MContainer, DSerialIntro, and DSerial denote several kernel and driver layers. With interrupts turned on in the kernel, it is immediately unclear how to show contextual refinement among different layers. For a kernel function like c_init, it cannot be easily refined into an atomic specification as the code can be interrupted at any point during the execution by a device interrupt, unless all possible interleaving of interrupts are encoded into the specification itself. Similarly, for a device driver function like puts, the code can be interrupted at any moment by interrupts triggered from other devices or the device itself.

In this paper, we propose a systematic way that strictly enforces isolation among different entities by construction. Our approach consists of the following two key ideas.

First, rather than viewing drivers as separate modules that interact with the CPU via in-memory shared-state, we instead view each driver as an extended device. We utilize

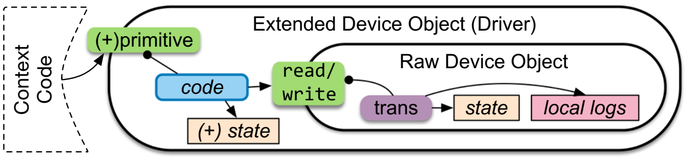

Fig. 9 The driver as an extended device

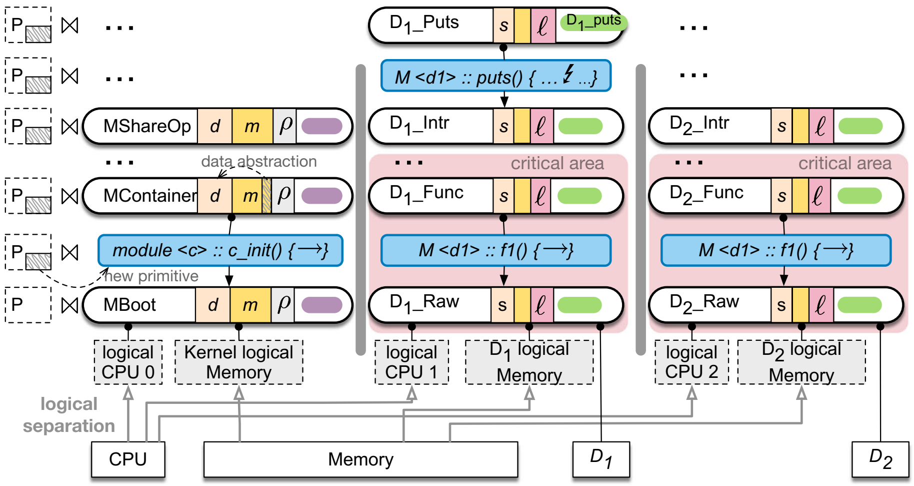

Fig. 10 Building certified abstraction layers with hardware interrupts: our new approach

abstraction layers and contextual refinement to gradually abstract the memory shared between a device and its driver into the internal abstract states of a more general device. Furthermore, we use the same technique to abstract those driver functions that manipulate these data into the abstract primitives of a higher level device. After this, our approach ensures that those abstract states can no longer be accessed by the other entities, through, e.g., memory reads and writes, but, rather, can only be manipulated via explicit calls to the device interface. We repeat these procedures so we can incrementally refine a raw device into more and more abstract devices by wrapping them with the relevant device drivers (see Fig. 9). In the rest of the paper, we call this extended abstract device a *device object*, to distinguish it from the raw hardware device. Note that in our model, device objects are indeed treated similarly to raw devices, and both have quite similar interfaces.

Second, we introduce and verify the interrupt handler for each device at the lowest machine model, which is not yet suitable for reasoning about interruptible code. This is possible because, for each device, we require that either the interrupt be disabled or its corresponding interrupt line be masked inside the interrupt handler of the device. Next, we introduce a new abstract machine with a more abstract interrupt model, that provides strong isolation properties amongst different device objects and the kernel, in which any future (context) code with interrupts turned on can be reasoned about naturally. We prove a strong contextual refinement property between these two abstract machines: any context code running on the machine with the abstract interrupt model (overlay) retains an equivalent behavior when it is running on top of the machine with the concrete hardware interrupt model (underlay).

Figure 10 shows the layer hierarchy of our interruptible kernel with device drivers. We treat the driver code as if it runs on its own device's "logical CPU," and each logical CPU operates on its own separate internal states. Thus, the approach provides a systematic way of assuring isolation among different device objects (running on its own local logical CPUs) and the rest of the kernel.

On the kernel side (the layer hierarchy on the left hand side of Fig. 10), the contextual refinement is achieved in the same way as shown in Sect. 2 since the hardware interrupts (from the other logical CPUs with separate states) no longer affect the execution of any kernel primitive (like c_init), i.e., the kernel is completely interrupt-unaware.

Similarly, the device driver functions are no longer affected by the hardware interrupts triggered from other devices. For each device D running on top of its own logical CPU, we first introduce and verify part of the driver in the *critical area*, i.e., the low-level device functions that should not be interrupted by the same device, and the interrupt handler of the device. Next, we use contextual refinement to introduce a new layer that has a more abstract interrupt model. On this layer, we can introduce and verify even interruptible driver code (e.g., puts) while still enforcing strong isolation and providing clean interface to the kernel.

4 Machine Model with Devices

In this section, we present our machine model, which is based on the Intel x86 architecture. We start from the *LAsm* machine model, and extend it to model devices and interrupts.

Our devices are modeled as finite state transition systems interacting with the CPU and the external environments. Each read/write (input/output) operation initiated from the CPU triggers an atomic big-step transition in the corresponding device. Device transitions (i.e., trans in Fig. 9) are affected by two types of interactions, one by the CPU and another by external events.

Device Transitions caused by the CPU The CPU may trigger a device transition through I/O instructions or memory-mapped I/O operations. These operations can be categorized into the following two actions:

Definition 1 (CPU Operation on a Device)

$$\begin{array}{lll} \mathcal{O} ::= & \mathsf{input}\ n & \triangleright \text{ Read value from the register at address } n \\ & \mid\ \mathsf{output}\ n\ v & \triangleright \text{ Write value } v \text{ to the register at address } n \end{array}$$

For every device, we define an atomic transition function δ^{CPU}, which takes the current device state s and a CPU operation o, and returns the new state s'. Note that δ^{CPU} is not a CPU transition, instead, it is strictly a *device transition* triggered by a CPU I/O operation.

Device Transitions caused by External Events Device transitions can also be caused by events from the external environment, such as the keyboard or network, with specific transitions depending on the kind of event. When modeling these external events, we take a minimalistic approach: though the devices can receive all kinds of different external events, we only model those that change the observable behavior of the device. Thus, the events do not map one to one to the transitions in the device hardware but rather to the CPU observations on the hardware. We model the device interfaces, not the device internals. The device interface contains all the information that a programmer can know about its states. Some example events are:

Definition 2 (Device External Events)

$$
\begin{array}{lll}
E ::= & & \\
\triangleright \textit{ UART device} & & \\
\quad \mid \mathsf{Recv}\ (s : \mathsf{list\ char}) & & \triangleright \textit{ UART receives string s} \\
\quad \mid \mathsf{NoSendingCompAck} & & \triangleright \textit{ Sending is not complete} \\
\quad \mid \mathsf{SendingCompAck} & & \triangleright \textit{ UART completes the sending} \\
\triangleright \textit{ Keyboard device} & & \\
\quad \mid \mathsf{KeyPressed}\ (c : \mathbb{Z}) & & \triangleright \textit{ A specific key is pressed} \\
\quad \mid \mathsf{KeyReleased}\ (c : \mathbb{Z}) & & \triangleright \textit{ A specific key is released} \\
\ldots & &
\end{array}
$$

External events are unpredictable, as their causes are not controlled by the OS. We determinize the behavior of each device by parametrizing it with the set of all possible list of events ℓ^{env} that will be processed sequentially when the CPU performs I/O operations on this device. The atomic transition function δ^{env} takes an external event e as input and changes the device states accordingly.

Note that events, even within a single device, can commute. For example, a serial port serves two roles: to receive user input and to send program output. Accordingly, among the events a serial device can receive are one for the reception of a new input string, and one signaling that some past output operation has been completed. Consider a function that first writes to a serial port, then waits until the write operation is completed by repeatedly reading some relevant status register. During one of these reads the user might send new input to the serial port. It would be reasonable for the device to observe the corresponding *Recv* event during one of the register reads, but doing so would make verifying the write function unnecessarily complex; not only would a function need to handle its own logic, but it would also need to handle any other state transition the device could undergo, even if the result were not observable in the current function.

To address this verification challenge, each device keeps a set of local logs $\boldsymbol{\ell} = \{\ell_1, \ldots \ell_k\}$, each of which is a strict prefix of ℓ^{env}.[1] The serial device from the example above could contain two local logs, one for input and one for output. Then when δ^{env} receives an event that does not correspond to the currently processed action, the event can simply be skipped. When a later action observes a part of the device state which is affected by the event, that action will handle the event. In the serial port example, we would defer handling the *Recv* event until some process reads from the serial port.

Every raw device provides two I/O primitives: `read` n and `write` $n\ v$. The `read` primitive first updates the device state based on the environmental device transition δ^{env} with the next relevant external event in ℓ^{env}, then returns a value from the new state, and finally does the transition δ^{CPU} triggered by this read action. The `write` primitive first triggers the transition δ^{env} to update the device state based on the next relevant external event, then performs the transition δ^{CPU} initiated by this write operation.

In the following, the function $\mathsf{next}(\ell^{env}, \ell_i)$ finds the first relevant event e in ℓ^{env} that has not yet been processed with respect to the local log ℓ_i, and returns the event e plus a new local log that is synchronized with ℓ^{env} up to the event e.

[1] We have chosen the prefix form over the subset to allow us determine more easily where the current execution is at on the global event list.

Now, we define the operational semantics of the set of device primitives formally. Let κ be the function retrieving the value of device register addressed by n, then we have:

$$\frac{\begin{array}{c}(e, \ell_i') = \mathsf{next}(\ell^{env}, \ell_i) \\ s' = \delta^{\mathsf{env}}(s, e) \qquad res = \kappa(n, s') \qquad s'' = \delta^{\mathsf{CPU}}(s', \mathsf{input}\ n)\end{array}}{\mathtt{read}(n, s, \ell_i, \ell^{env}) = (res, s'', \ell_i')} \qquad (\mathtt{read})$$

$$\frac{(e, \ell_i') = \mathsf{next}(\ell^{env}, \ell_i) \qquad s' = \delta^{\mathsf{env}}(s, e) \qquad s'' = \delta^{\mathsf{CPU}}(s', (\mathsf{output}\ n\ v))}{\mathtt{write}(n, v, s, \ell_i, \ell^{env}) = (s'', \ell_i')} \qquad (\mathtt{write})$$

Thanks to the local logs, this machine model eliminates much of the nondeterminism that complicates reasoning about asynchronous systems. Nonetheless, it accurately models the observable behaviors of real hardware.

Ideally, the global event list should contain a sequence of small atomic events which get associated with the time when they get triggered. Then the framework should have some notion of real-time clock that allows us to determine what exact set of events should be consumed by the next transition based on the current time. This is out of scope of the current paper. Instead, we allow our transition to consume a single "combined" event (as a list of event). For example, the "Recv" event in the Definition 2 takes a list of characters. In addition, a device may also consume an "empty" event which indicates there was no event triggered in the real world since the last time we checked ℓ^{env}. We parameterize our proof in a way that it hold for all possible combinations of such combined event list.

5 Driver Framework with Interrupts

The processor inherently runs in parallel with devices. In Sect. 4, we have presented a machine model representing this level of concurrency. On top of this machine model, we build certified abstraction layers introducing more and more driver code. At each abstraction layer, our model enforces systematic isolation among the different device objects and the rest of the kernel, so that interaction with one device object does not affect the states of other device objects nor the rest of the kernel. Thus, isolation properties are satisfied by construction. This dramatically simplifies our reasoning by allowing us, at any given time, to focus on only the device objects that are currently interacted with.

In this section, we define the device object more formally; then we show how to incorporate interrupts into our model while still following our isolation policy.

5.1 Device Objects

A *device object* is a logical abstraction containing a hardware device plus its related drivers. Each device object consists of a set of abstract states, abstracting the private states of the device (e.g., device registers, driver private memory); and a set of primitives, abstracting the module interface. The abstract states are private to the device object, and can only be manipulated by explicit calls to the device object's primitives. This is achieved by establishing a contextual refinement relation from the concrete memory and device function implementation to the abstract state and primitives. As shown in Fig. 11, we follow the layer-based methodology introduced in Sect. 2 and utilize the CompCert memory permissions [31] to hide the relevant memory at overlay, which prevents the context code from accessing the object's private

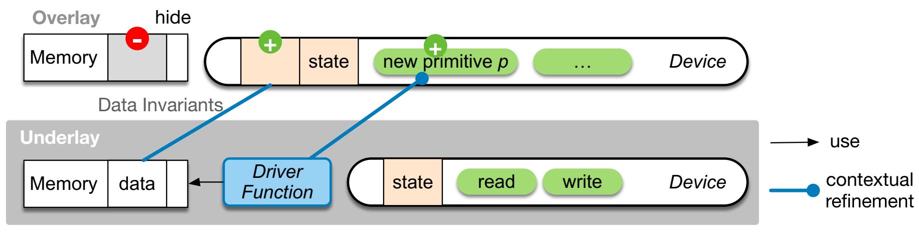

Fig. 11 Layer-based contextual refinement of the device object

data. These logical permissions do not correspond to any physical protection mechanism, but are used to ensure that the abstract machine at overlay gets stuck if any code tries to directly access this portion of data. The safety proof of our entire operating system (the kernel never gets stuck) guarantees that such a situation never happens. The set of driver functions at underlay, which manipulate the memory that will be abstracted away at overlay, are themselves abstracted into the set of device primitives at the overlay (see Fig. 11).

For example, the console buffer is implemented as a circular buffer in our console driver. The concrete implementations of the buffer operators (*cb_read* and *cb_write*) directly manipulate the concrete circular buffer in memory. At a higher layer, in our abstract console device object, the logical buffer is represented as a list, and the primitives are specified directly over this abstract list, i.e., the *cb_read* simply returns the head element in the list, while *cb_write* adds the new element to the end of the list, discarding a single head element if the size of the list exceeds its limit. The contextual refinement relation between the two layers ensures that any code running on top of the more abstract overlay exhibits behavior equivalent to running on top of the underlay.

The primitives at the underlay can be passed through to the overlay, or hidden if they are no longer needed. For example, once the primitive *ahci_transfer* is introduced at the overlay, the underlay primitives *ahci_read* and *ahci_write*, used to implement *ahci_transfer*, are hidden. This facilitates the invariant proofs as stronger invariants can be introduced at higher layers, which could otherwise be violated by the lower-level primitives.

This kind of abstraction does not necessarily have to include any code, and sometimes are achieved already at the raw device level. For example, some part of memory may be designated to the hardware device to set up the direct memory access (DMA) to allow the device directly read from or write to the main memory without going through the main CPU. In this case, the part of memory designated for DMA can also be abstracted into the device's internal abstract states through the contextual refinement.

Combining Device Objects At a certain abstraction layer, some drivers, or more generally, system services, may interact with multiple device objects, by, e.g., transferring data between two devices, or broadcasting messages to multiple devices. At this stage, such devices are no longer totally isolated, but are synchronized through hardware or software mechanisms. This does not fit directly into our model providing systematic isolation among different device objects and the rest of the kernel.

In the above scenario, we introduce at the overlay a single heterogeneous device object, which combines the device objects from the underlay via the newly introduced functions. The abstract machine at overlay thereby provides systematic isolation between the new abstract device object and the rest of the kernel. The internal states and local logs of the combined device object are the disjoint union of the relevant objects at underlay, while the functions that manipulate multiple device objects at underlay become primitives of the new device object,

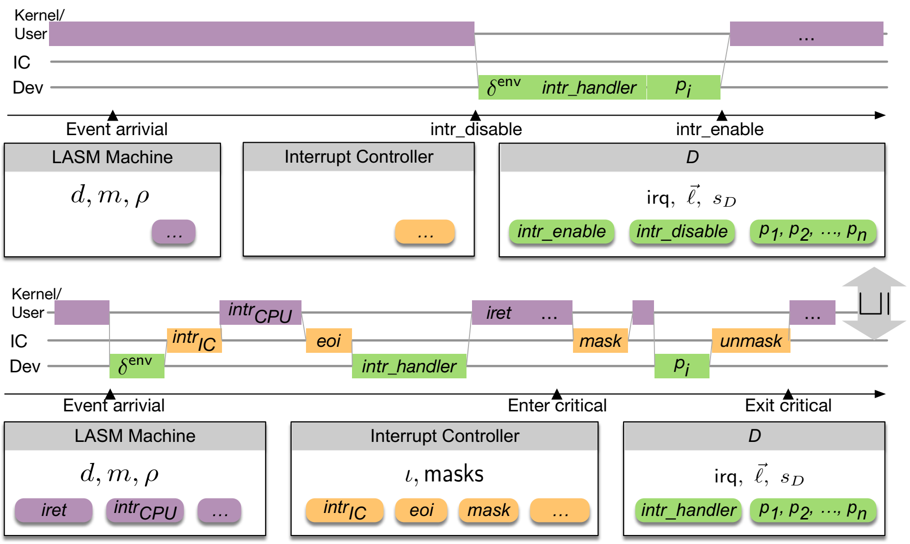

Fig. 12 The hardware interrupt model (bottom), the abstract interrupt model (top), and the contextual refinement between these two models

operating on a wider range of internal states, at overlay. As in all device objects, existing primitives can be either passed through to this new device, or hidden.

5.2 Interrupts

We now show how to adapt the interrupts into our setting. We first present our interrupt model at the hardware level, where the interrupt transitions are separately defined for the CPU, the interrupt controllers (IC), and the devices. At this low level we lack the full behaviors of interrupt handlers, so all the primitives verified at this machine level have the precondition that interrupts are disabled or the corresponding interrupt lines are masked. A special flag critical is defined in the abstract state of each device to make sure that every such low level primitives have precondition of critical being true to enter the critical section for accessing the data shared between the primitives and the interrupt handler of the device. On top of this hardware abstraction layer, we incrementally introduce and verify interrupt handlers for each device through abstraction layers. Above a certain abstraction layer, we have full behaviors of the interrupt handlers, so we introduce a new abstraction layer with an abstract interrupt model, where an interrupt only changes the state of the device object that triggered it. This makes the interrupt completely transparent to the CPU, the IC, and other devices; thus, guaranteeing our desired isolation properties. We prove the strong contextual refinement property between these two abstraction layers to ensure that any context program running on top of the overlay retains behavior equivalent to running atop the underlay. Starting from the abstraction layer with the abstract interrupt model, we support verification of any code with interrupts enabled.

The entities involved in any given interrupts are categorized into three parties (shown in the bottom half of Fig. 12). If a device transition (e.g., from the device *D* in Fig. 12) triggers an interrupt, it gets sent to the IC. The IC multiplexes several interrupt lines onto the CPU (i.e., the *LAsm* machine in Fig. 12), with the ability to mask and unmask each

$$\frac{s_{\mathsf{ic}}.\mathsf{masks}[N_D] = \mathsf{Masked} \qquad s_{\mathsf{ic}}.\mathsf{irqs}[N_D] = n}{\mathtt{intr}_{\mathsf{IC}}(s_{\mathsf{ic}}, N_D) = (s_{\mathsf{ic}}, \varnothing)} \quad (\mathtt{intr}^{m}_{\mathsf{IC}})$$

$$\frac{s_{\mathsf{ic}}.\mathsf{masks}[N_D] = \mathsf{Unmasked} \qquad s_{\mathsf{ic}}.\mathsf{irqs}[N_D] = n \qquad s_{\mathsf{ic}}.\iota = \varnothing}{\mathtt{intr}_{\mathsf{IC}}(s_{\mathsf{ic}}, N_D) = (s_{\mathsf{ic}}[\iota \leftarrow \mathsf{n}], \mathsf{IRQ}\ n)} \quad (\mathtt{intr}^{u}_{\mathsf{IC}})$$

$$\frac{}{\mathtt{eoi}(s_{\mathsf{ic}}) = s_{\mathsf{ic}}[\iota \leftarrow \varnothing]} \quad (\mathtt{eoi})$$

Fig. 13 Interrupt transition for the IC

interrupt line. At each transition, the IC selects the pending unmasked interrupt with the highest priority and forwards it to the CPU. When the CPU receives an interrupt signal, it first checks whether interrupts are enabled on that CPU, and, if so, saves the current context and jumps to the corresponding entry in the interrupt descriptor table (IDT). If interrupts are turned off, the interrupt signal is ignored. Thus, an interrupt involves at most three consecutive transitions: the device, the IC, and the CPU. These three transitions seem inter-related, and isolation among the three entities is non-obvious. In our approach, we first develop a low level hardware interrupt model that separately defines the set of interrupt related operations. Then these three disconnected components are united at some higher level abstract machine model after we have verified all the interrupt handlers.

5.2.1 Interrupt Transition for Devices

As described in Sect. 4, every raw device has its own transition function δ^{env} specifying how it reacts to the external events. When a particular transition triggers an interrupt (e.g., see the event arrival and the green box δ^{env} along the `Dev` line in the bottom half of Fig. 12), the device marks an interrupt request bit (*irq*) in its internal state.

5.2.2 Interrupt Transition for IC

When the IC receives an interrupt signal (e.g., see the orange box $\mathtt{intr}_{\mathsf{IC}}$ along the `IC` line in Fig. 12), it first checks whether the particular interrupt line is masked, and if so, it ignores the interrupt; if not, then the IC marks the corresponding interrupt line. The transition rules are defined in Fig. 13. Here, N_D is the corresponding interrupt line number of the device D which triggered the interrupt; it is fixed by the hardware connection, and is mapped to $\mathsf{IRQ}\ n$ by the configuration of the IC; the ι field of s_{IC} indicates which IRQ number is raised; we use $\varnothing$ to indicate that there is no raised interrupt. After the CPU performs its initial interrupt transition, the IC would receive the End Of Interrupt (EOI) signal (e.g., see the orange box `eoi` along the `IC` line in Fig. 12), it clears the raised mark on the interrupt line. The IC also has two primitives *mask* and *unmask*, which set the $s_{\mathsf{ic}}.\mathsf{masks}[N_D]$ of the interrupt line number N_D to Masked and $\mathsf{Unmasked}$ respectively.

5.2.3 Interrupt Transition for the CPU

As soon as the IC marks an interrupt line as raised, the CPU will perform its own interrupt transition (e.g., see the purple box $\mathtt{intr}_{\mathsf{CPU}}$ along the `Kernel/User` line in Fig. 12). Let ρ represent the register set, and d be the logical abstract states in the machine model, then the interrupt transition of a CPU is shown in Fig. 14.

$$\frac{\rho[\mathsf{EFLAGS.if}] = \mathsf{Disabled}}{\mathtt{intr}_{\mathsf{CPU}}(d,\rho,\mathsf{IRQ}\ n) = (d,\rho)} \qquad (\mathtt{intr}^{d}_{\mathsf{CPU}})$$

$$\frac{\begin{array}{c}\rho[\mathsf{EFLAGS.if}] = \mathsf{Enabled} \qquad d' = d[\mathsf{isr} \leftarrow \mathsf{true}] \qquad \mathsf{tfs'} = \mathtt{save\_context}(d'[\mathsf{tfs}],\rho) \\ d'' = d'[\mathsf{tfs} \leftarrow \mathsf{tfs'}] \qquad \mathsf{IDT}[n] = p \qquad \rho' = \rho[\mathsf{EIP} \leftarrow p][\mathsf{EFLAGS.if} \leftarrow \mathsf{Disabled}]\end{array}}{\mathtt{intr}_{\mathsf{CPU}}(d,\rho,\mathsf{IRQ}\ n) = (d'',\rho')} \qquad (\mathtt{intr}^{e}_{\mathsf{CPU}})$$

$$\frac{(\mathsf{tfs'},\rho') = \mathtt{restore\_context}(d[\mathsf{tfs}]) \qquad d' = d[\mathsf{isr} \leftarrow \mathsf{false}][\mathsf{tfs} \leftarrow \mathsf{tfs'}]}{\mathtt{iret}(d,\rho) = (d',\rho')} \qquad (\mathtt{iret})$$

Fig. 14 Interrupt transition for the CPU

We use EFLAGS.if to represent the interrupt flag bit in the EFLAGS register. If interrupts are disabled inside the CPU, the $\mathtt{intr}_{\mathsf{CPU}}$ primitive is totally transparent. Otherwise, it first changes the logical `isr` state to `true`, saves the current context into the end of the trap frame list (d'[`tfs`]), and jumps to the corresponding IDT entry. Here `isr` indicates whether the current machine execution is in the interrupt handling mode; the `save_context` function models the hardware behavior of saving the current context into the abstract state (d'[`tfs`]), which corresponds to the concrete stack frames in the memory (abstracted in layers below).

The primitive `iret` is the counterpart of $\mathtt{intr}_{\mathsf{CPU}}$, and models the behavior of CPU when the interrupt handler returns. It restores EFLAGS (including the old interrupt flag bit) from the context and thus also re-enable interrupts. The `restore_context` function models the hardware behavior of restoring the current context from the abstract state (d[`tfs`]).

Lemma 1 *The function* `restore_context` *is a left inverse of the function* `save_context`*.*

$$\frac{\begin{array}{c}\mathit{tfs}' = \mathtt{save\_context}(d[\mathit{tfs}], \rho) \\ (d', m', \rho') = f(d[\mathit{tfs} \leftarrow \mathit{tfs}'], m, \rho) \\ d[\mathit{tfs}] = d'[\mathit{tfs}] \qquad (\mathit{tfs}'', \rho'') = \mathit{restore\_context}(d'[\mathit{tfs}])\end{array}}{\mathit{tfs} = \mathit{tfs}'' \wedge \rho = \rho''}$$

The CPU also has two primitives *sti* and *cli*, which set the EFLAGS.if bit to Enabled and Disabled respectively.

5.2.4 Abstract Interrupt Model

The low-level machine model, we just described, is not suitable for reasoning about interrupts, since each of the three entities has its own disconnected view. For instance, when the CPU jumps to an IDT entry, it is unaware of the behavior of the corresponding interrupt handler, and when the IC sends an interrupt signal to the CPU, it does not know whether the interrupt will be handled or not. We would like to formally connect these three different views to derive a nice machine model that is suitable for reasoning about the end-to-end behavior of interrupts, i.e., an interrupt triggered by a device only modifies the particular device's internal states, and is transparent to the CPU, the IC, and other devices. To achieve this, we need a model of the full behavior of the interrupt handler for each device.

Starting from the above hardware interrupt model, we incrementally extend a raw device by wrapping it with driver code related to the interrupt handler, until we have fully verified the interrupt handler for the device. Each device has exactly one interrupt handler, which, by our isolation policy, only modifies the internal states of its particular device (Lemma 2), and cannot itself be interrupted by the same device.

DISABLENOINTR: Disable with no unhandled interrupt

$$\frac{(e,\ell_i') = \mathsf{next}(\ell^{env},\ell_i) \qquad s_{\mathsf{tmp}} = \delta^{\mathsf{env}}(s,e) \qquad s_{\mathsf{tmp}}.irq = \mathsf{false} \qquad s' = s[\mathsf{critical} \leftarrow \mathsf{true}]}{\mathtt{intr\_disable}(s,\ell_i,\ell^{env}) = (s',\ell_i)}$$

DISABLEINTR: Disable with unhandled interrupts

$$\frac{(e,\ell_i') = \mathsf{next}(\ell^{env},\ell_i) \qquad s' = \delta^{\mathsf{env}}(s,e) \qquad s'.irq = \mathsf{true} \qquad (s'',\ell_i'') = \mathtt{intr\_handler}(s',\ell_i',\ell^{env}) \qquad (s''',\ell_i''') = \mathtt{intr\_disable}(s'',\ell_i'',\ell^{env})}{\mathtt{intr\_disable}(s,\ell_i,\ell^{env}) = (s''',\ell_i''')}$$

ENABLENOINTR: Enable with no raised interrupt

$$\frac{s.irq = \mathsf{false} \qquad s' = s[\mathsf{critical} \leftarrow \mathsf{false}]}{\mathtt{intr\_enable}(s,\ell_i,\ell^{env}) = (s',\ell_i)}$$

ENABLEINTR: Enable with raised interrupts

$$\frac{s.irq = \mathsf{true} \qquad (s',\ell_i') = \mathtt{intr\_handler}(s,\ell_i,\ell^{env}) \qquad (s'',\ell_i'') = \mathtt{intr\_enable}(s',\ell_i',\ell^{env})}{\mathtt{intr\_enable}(s,\ell_i,\ell^{env}) = (s'',\ell_i'')}$$

Fig. 15 Transition rules for *intr_disable* and *intr_enable*

Lemma 2 *The interrupt handler (`intr_handler`) of device D can only observe and modify the abstract states of D.*

At this stage, we have the formal specification of the interrupt handler for a device. Next, through contextual refinement, we encapsulate the behaviors of interrupts into two primitives *intr_enable* and *intr_disable* at overlay for the device, which, as shown in the top half of Fig. 12, render interrupts transparent to the CPU and the IC. The precise transition rules are given in Fig. 15. Here, the next function, as defined at the end of Sect. 4, returns the next relevant event in ℓ^{env} and a new local log synchronized with ℓ^{env} up to the returned event. Before, the states on whether each device's interrupt line is masked or not were part of the IC devices. Following our isolation policy, in the new abstract interrupt model, we introduce a new abstract state critical in the device itself to indicate whether the particular interrupt is masked. When critical is true, it indicates that the interrupt line for the device is masked, thus the execution can enter the critical sections to read and write the device internal states, and *vice versa*. Recall that all the low level primitives (ones introduced before the interrupt handler) of the device have the precondition of critical to be true to enter the critical section.

The *intr_disable* primitive first synchronizes the device state with the previously unhandled interrupts then sets interrupt as disabled. It performs the synchronization by scanning the log from the last place *intr_enable* was called, until we hit the first event that did not trigger any interrupt. This ensures that subsequent observations on the device (in the abstract model) will be consistent with those performed under the hardware interrupt model. Note that *intr_disable* is defined recursively: it performs the environment transition δ^{env} on each event until we hit an event that does not trigger interrupts (i.e., the DISABLENOINTR case); the s_{tmp} state should be discarded since the device transition stops at the point where the last unhandled interrupt is handled.

The *intr_enable* primitive discharges any raised interrupts, then sets interrupt as enabled. This models the physical machine behavior, wherein interrupts (which can occur while interrupts are disabled) get delayed until interrupts are re-enabled. This causes the OS to immediately jump to the interrupt handler after re-enabling interrupts. This repeats until the

$$\frac{\begin{array}{c}(s'_{\mathsf{ic}}, \mathsf{IRQ}\ n) = \mathtt{intr}_{\mathsf{IC}}(s_{\mathsf{ic}}, N_D) \qquad (d', \rho') = \mathtt{intr}_{\mathsf{CPU}}(d, \rho, \mathsf{IRQ}\ n)\\ s''_{\mathsf{ic}} = \mathtt{eoi}(s'_{\mathsf{ic}}) \qquad (s'_D, \ell'_i) = \mathtt{intr\_handler}_{\mathsf{D}}(s_D, \ell_i, \ell^{env}) \qquad (d'', \rho'') = \mathtt{iret}(d', \rho')\end{array}}{\mathtt{intr}(d, m, \rho, s_{\mathsf{ic}}, s_D, \ell_i, \ell^{env}) = (d'', m, \rho'', s''_{\mathsf{ic}}, s'_D, \ell'_i)} \quad (\mathtt{normal})$$

$$\frac{(s'_{\mathsf{ic}}, \varnothing) = \mathtt{intr}_{\mathsf{IC}}(s_{\mathsf{ic}}, N_D)}{\mathtt{intr}(d, m, \rho, s_{\mathsf{ic}}, s_D, \ell_i, \ell^{env}) = (d, m, \rho, s'_{\mathsf{ic}}, s_D, \ell_i)} \quad (\mathtt{masked})$$

$$\frac{(s'_{\mathsf{ic}}, \mathsf{IRQ}\ n) = \mathtt{intr}_{\mathsf{IC}}(s_{\mathsf{ic}}, N_D) \qquad \mathsf{IDT}[n] = \mathsf{None}}{\mathtt{intr}(d, m, \rho, s_{\mathsf{ic}}, s_D, \ell_i, \ell^{env}) = \mathsf{None}} \quad (\mathtt{not\ routed})$$

$$\frac{\begin{array}{c}(s'_{\mathsf{ic}}, \mathsf{IRQ}\ n) = \mathtt{intr}_{\mathsf{IC}}(s_{\mathsf{ic}}, N_D) \qquad \rho[\mathsf{EFLAGS.if}] = \mathsf{Disabled}\\ (d', \rho') = \mathtt{intr}_{\mathsf{CPU}}(d, \rho, \mathsf{IRQ}\ n)\end{array}}{\mathtt{intr}(d, m, \rho, s_{\mathsf{ic}}, s_D, \ell_i, \ell^{env}) = (d', m, \rho', s_{\mathsf{ic}}, s_D, \ell_i)} \quad (\mathtt{disabled})$$

Fig. 16 Interrupt transition for the whole system, in the case when an interrupt is triggered by the device D on interrupt line number N_D

device no longer attempts to trigger an interrupt within the interrupt handler, and normal execution can continue.

With these two new primitives, the CPU transition in the abstract interrupt model can be completely oblivious of the device transitions. For example, in the top half of Fig. 12, the purple box along the `Kernel/User` line can ignore any event arrival from a device; the CPU for the `Kernel/User` line would only force the device transitions when it wants to make observations about a device (e.g., by calling *intr_disable*, then a high-level device primitive p_i, followed by *intr_enable*).

Contextual Refinement Between Two Interrupt Models To show the contextual refinement between the two abstraction layers in Fig. 12, we prove that the behavior of an IRQ can indeed be made transparent to the CPU and the IC.

Lemma 3 *An IRQ is transparent to the CPU and the IC, i.e., the transitions triggered by the IRQ only change the states of the corresponding device that triggered the interrupt.*

Proof When the interrupt is disabled on the CPU or the particular interrupt line is masked in the IC, the proof is obvious. When the interrupt is enabled, i.e., the corresponding interrupt line is routed, not masked, and the $\mathsf{EFLAGS.if}$ register bit is set, the state transition of the whole system is shown in Fig. 16. Here, the transition `intr` takes an abstract state d, the memory m, the register set ρ, the state of interrupt controller s_{ic}, the state of the device s_D, a local log of the device ℓ_i, the event list ℓ^{env}, and returns appropriate new system states after the interrupt transition is fully performed. In this case, we need to show that:

$$\frac{(d', m', \rho', s'_{\mathsf{ic}}, s'_D, \ell'_i) = \mathtt{intr}(d, m, \rho, s_{\mathsf{ic}}, s_D, \ell_i, \ell^{env})}{(s'_D, \ell'_i) = \mathtt{intr\_handler}_{\mathsf{D}}(s_D, \ell_i, \ell^{env}) \wedge d' = d \wedge m' = m \wedge \rho' = \rho \wedge s'_{\mathsf{ic}} = s_{\mathsf{ic}}}$$

This can be proven by composing the interrupt transition rules of the CPU and the IC with Lemma 1.[2]

[2] In our IC model, the middle states in the transition of interrupt delivery are discarded if the interrupt is not successfully handled. In the case when the interrupt is disabled in the CPU but not masked in the IC, the states of IC fallback to their original value. This model is still valid in the sense that we can delay this state change of IC until the next time when the interrupt is raised again for that particular device and gets handled successfully.

$$\frac{\begin{array}{c}(s'_{\mathsf{ic}}, \mathsf{IRQ}\ n) = \mathtt{intr}_{\mathsf{IC}}(s_{\mathsf{ic}}, N_D) \\ (d', \rho') = \mathtt{intr}_{\mathsf{CPU}}(d, \rho, \mathsf{IRQ}\ n) \\ s''_{\mathsf{ic}} = \mathtt{eoi}(s'_{\mathsf{ic}}) \\ s'''_{\mathsf{ic}} = \mathtt{mask}(s''_{\mathsf{ic}}, N_D) \qquad (d'', \rho'') = \mathtt{sti}(d', \rho') \qquad (s'_D, \ell'_i) = \mathtt{intr\_handler}_{\mathsf{D}}(s_D, \ell_i, \ell^{env}) \\ (d''', \rho''') = \mathtt{cli}(d'', \rho'') \qquad s''''_{\mathsf{ic}} = \mathtt{unmask}(s'''_{\mathsf{ic}}, N_D) \qquad (d'''', \rho'''') = \mathtt{iret}(d''', \rho''')\end{array}}{\mathtt{intr}(d, m, \rho, s_{\mathsf{ic}}, s_D, \ell_i, \ell^{env}) = (d'''', m, \rho'''', s''''_{\mathsf{ic}}, s'_D, \ell'_i)}$$

Fig. 17 Interrupt transition for the whole system when nested interrupts are allowed

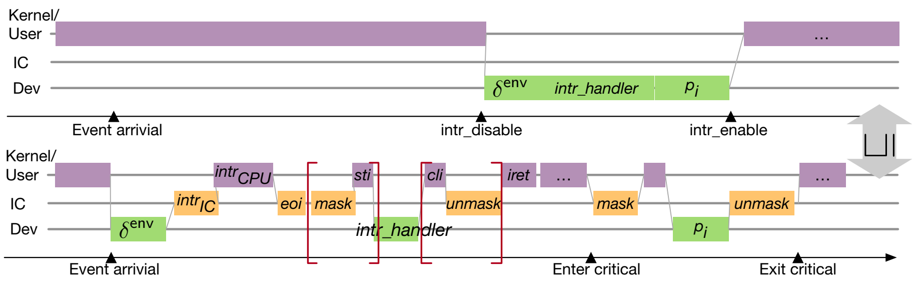

Fig. 18 The contextual refinement between interrupt models with nested interrupts

Corollary 1 *IRQs do not affect the kernel, i.e., they do not change any of the kernel's states*[3].

At this abstract interrupt model, every access to the device's abstract states needs to be guarded by a call to *intr_disable*, and this is systematically enforced through explicit preconditions of all device primitives. Note that at this level, an interrupt handler of a device only changes abstract states of that particular device. Thus, the correctness of deferring handling all the interrupts to *intr_disable* and *intr_enable* naturally follows, as none of the "non-critical" steps outside the pair can change the states of the device, nor reading any states of the device.

5.2.5 Nested Interrupts

Note that the $\mathtt{intr}_{\mathsf{CPU}}$ transition in Fig. 14 disables the interrupt. Thus between $\mathtt{intr}_{\mathsf{CPU}}$ and `iret` in Fig. 16, the interrupt is turned off, which means that no nested interrupts are allowed. In many cases, supporting nested interrupts is critical so that some high priority interrupt processing is not delayed by the low priority ones. The interrupt transition for the whole system with nested interrupts is shown in Fig. 17. Here, before the interrupt handler is called, we mask the interrupt line of the particular device (to make sure there is no nested interrupt from the same device) and then turn on the interrupt on the CPU. Accordingly, after the interrupt handling, we disable the CPU interrupt, then unmask the particular interrupt line before the `iret` transition is performed. We have proved that this model also refines the same abstract interrupt model (see Fig. 18).

6 Case Study

In this section, we present case studies of our verified drivers. Fig. 19 shows the overall structure of our verified serial console, while the left side reflects how the implementation of driver

[3] Remember, we consider device drivers a part of the device, not the kernel.

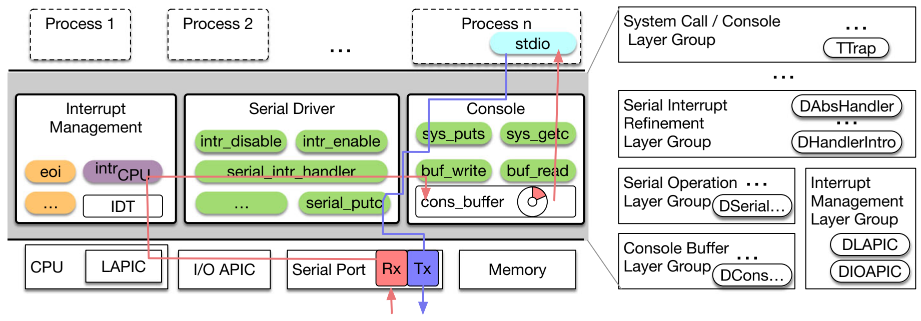

Fig. 19 The data-flow of serial console messages and corresponding layer decomposition

modules organized as well as the data/control flow. The right side gives an overview of how those modules are verified by construction of certified layers. We have used a single controller in Sect. 5 to make the presentation concise. However, mCertiKOS utilizes two physical interrupt controller devices: the I/O Advanced Programmable Interrupt Controller (I/O APIC) and the Local Advanced Programmable Interrupt Controller (Local APIC). Together with the serial device, we present the formal models of these three hardware devices throughout this section.

As shown in Fig. 19, at the system call level, users can invoke relevant system calls (sys_getc, sys_puts, etc.) to read from or write to the serial port. The red lines in the figure represents the flow of receiving data from serial port. The implementation is interrupt driven. Thus, when the serial receiving buffer (Rx) gets new data, the device triggers an interrupt that goes through the Local APIC and I/O APIC devices and the interrupt handler of serial device stores the data into the console circular buffer (`cons_buffer`). Later, when a user process makes the system call sys_getc, the system call handler simply pops and returns the head of `cons_buffer`. On the transmitting side, writing data to the serial port (blue lines in the figure) is completely synchronous. When the system call `sys_puts` is invoked, the system call handler calls the transmission function `serial_putc` of serial driver to write data to the transmission buffer (Tx) of serial device. The critical sections are protected by the functions `serial_intr_disable` and `serial_intr_enable` which masks and unmasks the interrupt signal of the serial device.

The verification of the console circular buffer (`cons_buffer`) is already presented in Sect. 2. In this section, we also present the formal model and verification of their drivers for the serial port, I/O APIC, and Local APIC.

6.1 Serial Port

Figure 20 illustrates a typical serial port with a bounded internal buffer of size 12. It consists of a RS-232 interface and a Universal Asynchronous Receiver/Transmitter (UART) controller. RS-232 delivers electrical signals between the UART controller and the connected cable. The UART controller is responsible for demodulating received data into digital bits and storing them into the internal receiving (Rx) buffer, and also modulating sent data from digital bits and inserting them into the transmission (Tx) buffer.

The hardware UART controller has many features, and the mCertiKOS serial driver only utilizes those parts needed for sending and receiving character strings. When modeling the

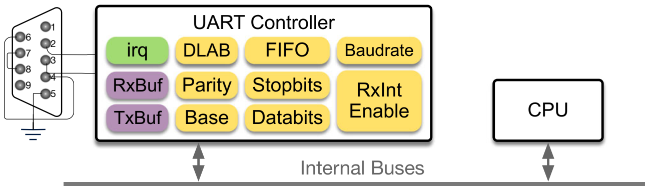

Fig. 20 The hardware connections of a serial port

serial port, we take the minimalistic approach of only modeling the set of features utilized by the existing drivers. The internal state of the serial port device is defined as:

Definition 3 Abstract state of serial device.

$$\begin{aligned}
s = (\ & \mathsf{RxBuf} : \mathsf{list\ char}, && \triangleright\ \textit{Receiving buffer}\\
& \mathsf{TxBuf} : \mathsf{list\ char}, && \triangleright\ \textit{Transmission buffer}\\
& \mathsf{irq} : \mathsf{bool}, && \triangleright\ \textit{Interrupt pending}\\
& \mathsf{Connected} : \mathsf{bool}, && \triangleright\ \textit{Power}\\
& \mathsf{Base} : \mathbb{Z}, && \triangleright\ \textit{Base address}\\
& \triangleright\ \textit{Line and modem configurations:}\\
& \mathsf{RxIntEnable} : \mathsf{bool},\ \mathsf{DLAB} : \mathsf{bool},\ \mathsf{Baudrate} : \mathbb{Z},\\
& \mathsf{Databits} : \mathbb{Z},\ \mathsf{Stopbits} : \mathbb{Z},\ \mathsf{Parity} : \mathsf{ParityType},\\
& \mathsf{FIFO} : \mathbb{Z},\ \mathsf{Modem} : \mathbb{Z}\ ).
\end{aligned}$$

There are three external events for the serial device. The serial event Recv s indicates that a string has been received. The SendingCompAck event implies the device received the acknowledgment that the characters in the transmission buffer have been sent out successfully, while the NoSendingCompAck events indicates that the sending of characters in the transmission buffer is not yet complete.

The serial device is configured to trigger an interrupt when it receives data (a nonempty string), and not to trigger any interrupt when the transmission buffer becomes empty, i.e., when the characters in the transmission buffer are sent out successfully. Thus, before any data is written to the serial port, we have to poll the transmission status until it becomes empty. We have chosen this setup because it covers both interrupt-triggering and polling events.

Note that, the states s.RxBuf and s.irq are disjoint from s.TxBuf under the environment transitions in that the former is for receiving data and the latter is for sending data only. This allows us to use two separate local logs in our device model, ℓ_{tx} (for transmission) and ℓ_{rx} (for receiving), to handle these possibly commutative events.

Next, in Fig. 21, we define the transition functions δ^{env} and δ^{CPU}, where δ^{env} needs to handle all the possible environmental events against the current state, and δ^{CPU} updates the current state based on the input and output addresses and values. Note that function `last` is used to model the action of dropping some elements in the front of the buffer when the length of the new buffer exceeds the hardware buffer size (BufSize). The baud_low in Fig. 21 is an example configuration of the control registers. The value of Baudrate is not used to simulate the timing of the signal, but is checked against the real hardware settings in certain transitions, such as δ^{CPU} for read and write, in order to verify that the driver is free of mis-configuration bugs.

By instantiating the device state and transition functions from our general device model in Sect. 4, we create a concrete model of the serial port with the `read` and `write` primitives.

$$\frac{s.\mathsf{Connected} = \mathsf{true} \qquad s.\mathsf{RxIntEnable} = \mathsf{true} \qquad e = \mathsf{Recv}\ w \qquad w \neq nil \qquad \mathsf{newBuf} = \mathtt{last}(\mathsf{BufSize}, (s.\mathsf{RxBuf}{+}{+}w))}{\delta^{\mathsf{env}}(s,e) = s[\mathsf{RxBuf} \leftarrow \mathsf{newBuf}][\mathsf{irq} \leftarrow \mathsf{true}]} \quad \text{(recvd\_irq)}$$

$$\frac{s.\mathsf{Connected} = \mathsf{true} \qquad s.\mathsf{RxIntEnable} = \mathsf{false} \qquad e = \mathsf{Recv}\ w \qquad w \neq nil \qquad \mathsf{newBuf} = \mathtt{last}(\mathsf{BufSize}, (s.\mathsf{RxBuf}{+}{+}w))}{\delta^{\mathsf{env}}(s,e) = s[\mathsf{RxBuf} \leftarrow \mathsf{newBuf}]} \quad \text{(recvd\_no\_irq)}$$

$$\frac{s.\mathsf{Connected} = \mathsf{true} \qquad e = \mathsf{Recv}\ w \qquad w = nil}{\delta^{\mathsf{env}}(s,e) = s} \quad \text{(norcv)}$$

$$\frac{s.\mathsf{Connected} = \mathsf{true} \qquad e = \mathsf{SendingCompAck}}{\delta^{\mathsf{env}}(s,e) = s[\mathsf{TxBuf} \leftarrow \varnothing]} \quad \text{(sent)}$$

$$\frac{s.\mathsf{Connected} = \mathsf{true} \qquad e = \mathsf{NoSendingCompAck}}{\delta^{\mathsf{env}}(s,e) = s} \quad \text{(noack)}$$

$$\frac{s.\mathsf{Connected} = \mathsf{true} \qquad o = \mathsf{input}\ n \qquad n = s.\mathsf{Base} + 0 \qquad s.\mathsf{DLAB} = \mathsf{false} \qquad s.\mathsf{RxBuf} = w}{\delta^{\mathsf{CPU}}(s,o) = s[\mathsf{RxBuf} \leftarrow \mathtt{tl}\ w][\mathsf{irq} \leftarrow \mathsf{false}]} \quad \text{(read)}$$

$$\frac{s.\mathsf{Connected} = \mathsf{true} \qquad o = \mathsf{output}\ n\ v \qquad n = s.\mathsf{Base} + 0 \qquad s.\mathsf{DLAB} = \mathsf{false} \qquad s.\mathsf{TxBuf} = w}{\delta^{\mathsf{CPU}}(s,o) = s[\mathsf{TxBuf} \leftarrow \mathtt{last}(\mathsf{BufSize}, (w{+}{+}[v]))]} \quad \text{(write)}$$

$$\frac{s.\mathsf{Connected} = \mathsf{true} \qquad o = \mathsf{output}\ n\ v \qquad n = s.\mathsf{Base} + 0 \qquad s.\mathsf{DLAB} = \mathsf{true} \qquad r = (s.\mathsf{Baudrate}\ \&\ \mathsf{0xff00})\ |\ v}{\delta^{\mathsf{CPU}}(s,o) = s[\mathsf{Baudrate} \leftarrow r]} \quad \text{(baudrate\_low)}$$

Fig. 21 The environment and CPU transition functions

```
1  void serial_putc (unsigned int c) {
2   unsigned int lsr = 0, i;
3   if ( get_serial_exist() ){
4    for (i = 0; !lsr && i < 12800; i++)
       {
5     lsr = serial_read(0x3FD) & 0x20;
6     delay();
7    }
8    serial_write (0x3F8, c);
9    ...
10 }
11
```

```
12 unsigned int serial_getc () {
13  unsigned int rv = 0;
14  unsigned int rx;
15  if (get_serial_exist()) {
16   if (serial_read(COM1+COM_LSR, BIT1)
        %2==1) {
17    rx = serial_read(COM1+COM_RX, M_ALL);
18    cb_write(rx);
19    ...
20 }
21
```

Fig. 22 The implementation of `serial_putc` and `serial_getc` in C

Next, we show how the drivers are specified and verified on top of this model. Figure 22 shows code fragments of the function `serial_putc` and `serial_getc`. There, the `serial_read` and `serial_write` are the two primitives in the serial hardware model, while `get_serial_exist` is a new primitive (already verified in some underlay) indicating whether the serial device is already initialized. The if statement (line 3) prevents any misuse of `serial_putc()` before initialization.

Fig. 23 The implementation of `serial_puts` in C

```c
void serial_puts(char * s, int len) {
    int i = 0;
    while (i < len && s[i] != 0) {
        serial_intr_disable();
        serial_putc(s[i]);
        serial_intr_enable();
        i++;
    }
}

```

For `serial_putc`, if the $s.\mathsf{TxBuf}$ buffer is initially empty, or the device receives a SendingCompAck event during the loop (line 4–6), the program sends the character `c` to the serial port (line 8). The function `serial_putc` is specified as follows:

$$\frac{s.\mathsf{TxBuf} = \varnothing \qquad s.\mathsf{get\_serial\_exist} = \mathsf{true} \\ s' = s[\mathsf{TxBuf} \leftarrow [c]] \qquad (e, \ell'_{\mathsf{tx}}) = \mathtt{next}(\ell^{env}, \ell_{\mathsf{tx}}) \qquad (e', \ell''_{\mathsf{tx}}) = \mathtt{next}(\ell^{env}, \ell'_{\mathsf{tx}})}{\mathtt{serial\_putc}(s, c, \ell_{\mathsf{tx}}, \ell^{env}) = (s', \ell''_{\mathsf{tx}})}$$

$$\frac{s.\mathsf{TxBuf} \neq \varnothing \qquad s.\mathsf{get\_serial\_exist} = \mathsf{true} \qquad (e, \ell'_{\mathsf{tx}}) = \mathtt{next}(\ell^{env}, \ell_{\mathsf{tx}}) \\ s' = \delta^{env}(s, e) \qquad (s'', \ell''_{\mathsf{tx}}) = \mathtt{serial\_putc}(s', c, \ell'_{\mathsf{tx}}, \ell^{env})}{\mathtt{serial\_putc}(s, c, \ell_{\mathsf{tx}}, \ell^{env}) = (s'', \ell''_{\mathsf{tx}})}$$

The first rule above shows the case when the transmission buffer is originally empty. Here, lsr immediately becomes 1 in the first loop iteration, and the character is written to the transmission buffer in the device right away. Note that the function next is called twice because the implementation of serial_putc has two serial I/O operations for the base case: one to check whether the transmission buffer is empty, and the other to put the character into the transmission buffer.

The second rule above shows the case when the initial transmission buffer is not empty. Here, the device performs transition based on the received event e, and repeats the same process until it finally receives the SendingCompAck event. Then, by definition of δ^{env} in Fig. 21, the transmission buffer becomes empty and the next recursive call falls into the first case of the specification.

As for `serial_getc`, a check of status register will be first performed to clear the irq state, and confirm that there are pending receiving messages (line 16). If the receiving buffer is not empty, the head of the buffer is fetched (line 17) and inserted into the console buffer.

In Fig. 23, we show the implementation of the driver function serial_puts that writes a string into the serial device by repeatedly calling serial_putc for each character in the input string. Each call to serial_putc is wrapped with calls to serial_intr_disable and serial_intr_enable (both derived from those in Fig. 15) to protect the critical section.

For each driver function, we prove that the concrete implementation satisfies its specification. Our proof is termination-sensitive; we prove total correctness of each function. In the case of `serial_putc`, the maximum iteration counter (12,800) is used solely to enforce termination. We maintain an invariant on ℓ^{env} that the serial port receives a SendingCompAck event within 12,800 times the `delay()` function is called. This assumption is reasonable because a sending operation that does not complete within this time frame implies an underlying hardware failure.

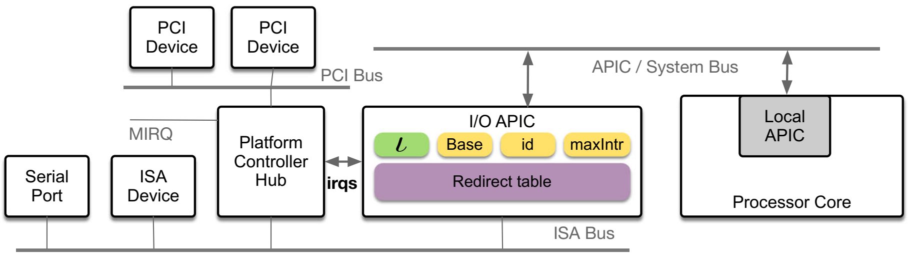

Fig. 24 The hardware connections and registers of APIC

Fig. 25 Internal states of I/O APIC

$$
\begin{array}{lll}
s = (& \iota : \mathsf{option}(\mathbb{Z} \times \mathbb{Z}), & \rhd \textit{Current raised IRQ} \\
& \mathsf{id} : \mathbb{Z}, & \rhd \textit{I/O APIC ID} \\
& \mathsf{maxIntr} : \mathbb{Z}, & \rhd \textit{Max redirection entries} \\
& \mathsf{irqs} : \mathsf{list}\ \mathbb{Z}, & \rhd \textit{Redirected IRQs} \\
& \mathsf{masks} : \mathsf{list}\ \mathsf{TMask}, & \rhd \textit{Interrupt line masks} \\
& \mathsf{dests} : \mathsf{list}\ \mathbb{Z}). & \rhd \textit{Destinations in Local APIC ID}
\end{array}
$$

$$
\frac{s.\iota = \mathsf{None} \qquad w = \mathsf{IRQ}\ n \qquad s.\mathsf{irqs}[n] = q \qquad s.\mathsf{masks}[n] = \mathsf{Unmasked} \qquad s.\mathsf{dests}[n] = d}{\delta^{intr}(s, w) = s[\iota \leftarrow \mathsf{Some}\ (q, d)]} \quad \text{(delivered)}
$$

$$
\frac{s.\iota = \mathsf{None} \qquad w = \mathsf{IRQ}\ n \qquad s.\mathsf{masks}[n] = \mathsf{Masked}}{\delta^{intr}(s, w) = s} \quad \text{(masked)}
$$

$$
\frac{s.\iota = \mathsf{Some}(q, d) \qquad w = \mathsf{EOI} \qquad s.\mathsf{irqs}[n] = q \qquad s.\mathsf{dests}[n] = d}{\delta^{intr}(s, w) = s[\iota \leftarrow \mathsf{None}]} \quad \text{(EOI)}
$$

Fig. 26 I/O APIC transition rules

6.2 I/O APIC

An I/O APIC device collects interrupts from externally connected devices and distributes them to the corresponding Local APIC. It can be programmed to mask one or more of these interrupt lines, if the OS does not wish to receive interrupts from some device(s).

Figure 24 illustrates the registers and connections of an I/O APIC device, which collects the IRQs from devices and route them to the Local APIC controller on CPU. The "redirect table" controls the mapping between interrupt lines and the IRQ number. Following our minimalistic approach, we omit logical destination, remote-IRR configuration, and other features that are not used in our kernel. The internal state of the I/O APIC is defined in Fig. 25, where ι represents the interrupt request currently being processed and its corresponding destination LAPIC ID.

As an interrupt controller, the I/O APIC is treated as a special device. It does not observe any event from the external environment, and thus has neither a local log nor an environmental transition δ^{env}, but instead, it receives interrupt requests from the devices and EOI signals from the Local APIC. We have introduced a special transition function δ^{intr} to specify these interrupt-related behaviors. Accordingly, δ^{intr} takes two kinds of events: $\mathsf{IRQ}\ n$ indicates that an IRQ with number n is triggered by a device; and EOI states that the latest interrupt request has been handled by the OS. The interesting parts of the transition rules for δ^{intr} are shown in Fig. 26.

```
 1 void ioapic_init(void) {
 2  int j = 0;
 3  int maxintr = ioapic_read(1)>>24;
 4  while(j <= maxintr) {
 5   ioapic_write(0x10+2*j,0x10000|gsi+j);
 6   ioapic_write(0x10+2*j+1,0);
 7  }
 8 }
 9
```

```
10 void ioapic_mask (unsigned int n)
11 {
12  unsigned int r;
13  int m = ioapic_read(1)>>24;
14  if (n>=gsi && n<=gsi + m) {
15   r = ioapic_read(0x10+2*(n-gsi));
16   r |= 0x10000;
17   ioapic_write (0x10+2*(n-gsi),r);
18  }
19 }
20
```

Fig. 27 The implementation of `ioapic_init` in C

In addition to δ^{intr}, the I/O APIC also contains the CPU transition function δ^{CPU} used to specify the read/write primitives of I/O APIC, discussed in Sect. 4.

In order to coordinate the IRQs assigned by the kernel with the external interrupt vector, a kernel usually utilizes the Global System Interrupt (GSI) number. Thus, the IC is first extended into a device object with this extra data as part of its internal state. Then this IC object is further extended into more abstract objects by introducing additional driver layers.

At the top level, the I/O APIC device object provides four primitives, two of them are used to setup the IRQ mappings in the I/O APIC. Specifically, `ioapic_init` initializes the device when the kernel boots, and `ioapic_enable` links a given interrupt line to a Local APIC when a new device is plugged in or some device changes its working mode. Another two, namely `ioapic_mask` and `ioapic_unmask`, are used to enable and disable a certain interrupt line.

Function `ioapic_init` in Fig. 27 shows the initialization of the I/O APIC. It first reads the size of the interrupt redirection table (line 2), and for each entry, marks the corresponding interrupt to be edge-triggered, active high, and masked (i.e. not routed to any Local APIC). The behavior of this function can be described using the following rule:

$$\frac{\begin{array}{c} l = s.\mathsf{maxIntr} \qquad s' = s[\mathsf{masks}[1..l] \leftarrow \mathsf{Masked}] \\ s'' = s'[\mathsf{irqs}[1..l] \leftarrow \mathsf{gsi}..(\mathsf{gsi}+l)][\mathsf{dest}[1..l] \leftarrow 0][\iota \leftarrow (\mathsf{None}, \mathsf{None})] \end{array}}{\texttt{ioapic\_init}(s) = s''}$$

Function `ioapic_mask` in Fig. 27 shows the code for masking interrupt line n. It first reads the entry '$n - \mathsf{gsi}$' in the interrupt redirection table (line 13), sets the mask bit (line 14), and then writes it back to the redirection table. The behavior can be described using the following rule:

$$\frac{\mathsf{gsi} \le n \le \mathsf{gsi} + s.\mathsf{maxIntr} \qquad s' = s[\mathsf{masks}[n - \mathsf{gsi}] \leftarrow \mathsf{Masked}]}{\texttt{ioapic\_mask}(s, n) = s'}$$

6.3 Local APIC

Each processor has a Local APIC device which manages and delivers the interrupt requests dedicated to this core. It serves as a bridge between I/O APIC and the processor core and is also programmable which can be used to specify the manner for each type of interrupt. As shown in Fig. 28, the Local APIC in our kernel is mainly served to deliver the external interrupt and generate the End-of-Interrupt (EOI) signals. An EOI signal indicates current interrupt is completely handled so that the I/O APIC can issue the next interrupt.

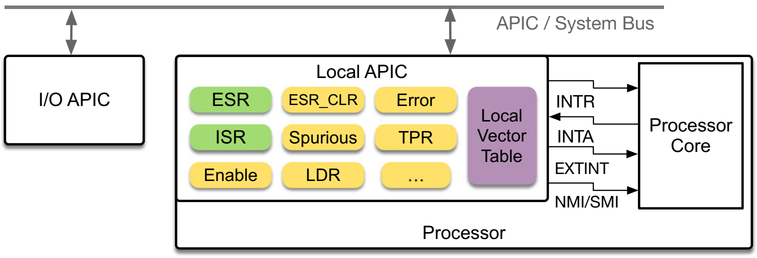

Fig. 28 The hardware connections of a Local APIC

$$
\begin{array}{lll}
s = (& \mathsf{ISR} : \mathsf{option}(\mathsf{IntType} \times \mathbb{Z}), & \rhd \textit{Current in-service IRQ} \\
& \mathsf{ESR} : \mathbb{Z}, & \rhd \textit{Error status register} \\
& \mathsf{ESR\_CLR} : \mathbb{Z}, & \rhd \textit{ESR back-to-back clear ready} \\
& \mathsf{Enable} : \mathbb{B}, & \rhd \textit{Local APIC enable} \\
& \mathsf{Spurious} : \mathbb{Z}, & \rhd \textit{Spurious interrupt vector} \\
& \mathsf{Timer} : \mathsf{LvtEntry}, & \rhd \textit{Interrupt control of timer} \\
& \mathsf{Lint\ 0} : \mathsf{LvtEntry}, & \rhd \textit{Interrupt control of local connected I/O devices on line 0} \\
& \mathsf{Lint\ 1} : \mathsf{LvtEntry}, & \rhd \textit{Interrupt control of local connected I/O devices on line 1} \\
& \mathsf{PCint} : \mathsf{LvtEntry}, & \rhd \textit{Interrupt control of performance counter} \\
& \mathsf{Error} : \mathsf{LvtEntry}, & \rhd \textit{Interrupt control of APIC internal error} \\
& \mathsf{LDR} : \mathbb{Z}, & \rhd \textit{Logical destination register} \\
& \mathsf{TPR} : \mathbb{Z} \times \mathbb{Z}). & \rhd \textit{Task priority register}
\end{array}
$$

Fig. 29 Internal states of Local APIC

$$
\frac{s.\mathsf{ISR} = \mathsf{None} \qquad s.\mathsf{Enable} = \mathsf{true} \qquad s.\mathsf{TPR} = (0,0) \qquad w = (\mathsf{IoAPIC}, n)}{\delta^{intr}(s, w) = s[\mathsf{ISR} \leftarrow (\mathsf{ExtInt}, n)][\mathsf{ESR\_CLR} \leftarrow \mathsf{false}]} \quad \text{(ioapic\_delivered)}
$$

$$
\frac{o = \mathsf{output}\ n, v \qquad n = 44 \qquad v = 0 \qquad s.\mathsf{Enable} = \mathsf{true}}{\delta^{\mathsf{CPU}}(s, o) = s[\mathsf{ISR} \leftarrow \mathsf{None}][\mathsf{ESR\_CLR} \leftarrow \mathsf{false}]} \quad \text{(EOI)}
$$

Fig. 30 Local APIC transition rules

The internal state of the Local APIC is defined in Fig. 29, where ISR represents the IRQ which is being handled. It is set to None if the CPU is available for incoming interrupts. Following the minimalistic approach, we omit the features that are not used in our kernel, such as performance monitoring counters and prioritizer. Other fields in the Local APIC device object have structure and meaning as in the hardware specification manual.

Figure 30 presents some transition rules in Local APIC. The first rule shows a successful delivery of external IRQ from I/O APIC if there is no interrupt in service. The second rule shows the transition of EOI in the Local APIC side, where 44 is the offset to access EOI in the memory mapped registers. Note that the models of I/O APIC and Local APIC can be merged into a heterogeneous interrupt controller with the simplified transition rules that are presented in Sect. 5.

7 Coq Implementation of Certified Abstraction Layers

In this section, we present in detail the actual implementation of the certified abstraction layers. To make it more easy to reproduce our work, various examples in this sections are

Table 1 Common Coq terms, keywords and definitions

Statement	Description
`x: T`	The type of `x` is `T`
`A -> B`	Arrow type. Non-dependent product
`option _`	Option type, either ‘Some v’ or ‘None’
`hd :: tl`	List construction. Create a new list with an element ‘hd’ and an old list ‘tl’
`l1 ++ l2`	List concatenation. Create a new list with `l1` followed by `l2`
`s {f: v}`	Field update. Replace the value of field ‘f’ in ‘s’ with ‘v’
`Definition x: X := t.`	Constant definition
`Function x` $(a_1{:}T_1)(a_2{:}T_2)$	Function definition with arguments $a_1, a_2, \cdots$, return value rv and
`... : (rv: T) :=` t.	function body t
`Fixpoint x` $(a_1{:}T_1)(a_2{:}T_2)$	As for function definition but t can make recursive calls to x.
`... : (rv: T) :=` t.	
`match t with` p_1 `=>` t_1 | ... `end`	Pattern matching, select t_1 if t matches with p_1
`if b then t else u`	Binary selection, b can be either `true` or `false`
`let x :=` t `in` u	Local binding

presented directly as they are implemented in the Coq proof assistant. Various Coq constructions commonly used throughout this section is presented in Table 1. We also explain them as they show up in the examples.

Starting from a bottom-most layer interface abstracting the CPU, device models and external interfaces, we gradually introduce driver code modules to develop certified abstraction layers by introducing higher level layer interfaces abstracting the concrete driver behaviors, and showing contextual refinement between each overlay interface and the code module running on the underlay interface, for each of two adjacent layer interfaces. More formally, for each layer interface L_{low}, we introduce a device driver module M, and a new overlay interface L_{high} abstracting the behaviors of M, and show that $[\![M]\!]L_{low}$, is a *contextual refinement* of the overlay interface L_{high}. By repeating this strategy, we have developed a stack of layer interfaces, with the top-most layer interface containing the full abstract behaviors of the verified device drivers. Next, we introduce these layer interfaces and the verified driver modules one by one. The implementations of various concepts in Coq is shown in Fig. 31, with the concrete examples used in the rest of sections listed on the right hand side. We explain the concepts and implementations in more detail as we get to the corresponding layer implementation.

7.1 The Lowest Level Layer Interface: MBoot

The interface MBoot is the bottom-most layer interface used to model the behaviors of the CPU and devices. In addition to the abstract states and primitives in the original verified mCertiKOS, we have extended MBoot with the new abstract states and read/write primitives for the serial port, I/O APIC, and Local APIC devices as described in Sect. 6.

This layer interface introduces a set of layer invariants to enforce the validity of various hardware states, and all the read/write primitives of the devices are required to preserve the invariants. In the rest of the invariant definitions, nth is a list operation defined in Coq, which

Event	$E \in$	Recv $(s :$ list char$) \mid \cdots$	`Recv str / SendingCompAck / ···`
Log	$\ell \in$	list *Event*	`s.ltx / s.lrx / ...`
EnvLog	ℓ^{env} :	All possible list of events.	Build into the 'next' function.
ident	$id \in$	*Nat*	`Let serial_init := 5950.`
val	$v :=$	$Vint\ v \mid \ldots \mid Vundef$	`Vint v / Vundef / ...`
Abs	$s \in$	$Type \times L^k$	`serial / ioapic / ...`
Primitive	$p \in$	list $val \to Abs \to L \to val \to Abs \to Prop$	`Function serial_putc_spec ...`
Invariant	INV $\in$	$Abs \to$ Prop	`valid_cons_buf_length: ...`
Impl	κ :	LAsm instruction list or ClightX function.	
LayerIf	$L \in$	$(ident \to Primitive)^*$	`Definition dserial :=` `serial_init` $\mapsto$ `serial_init_spec` $\oplus \ldots$
Module	$M \in$	$(ident \to Impl)^*$	`Definition DSerial_module :=` `serial_init` $\mapsto$ `f_serial_init` $:: \ldots$

Fig. 31 The formal model and Coq presentation of device abstract states and layer interfaces

takes a natural number n, a list l, a default value v, and returns the n'th value in the list l. In the case when the index is invalid, it returns the default value v. nth_error is a similar operation but returns either a value or none in the case of invalid index, instead of returning the default value. Zlength is another list operation that returns the length of provided list as an integer.

Invariant 1 (valid serial port) *The base address of the I/O port to access COM1 is always* `0x3F8` *(1016 in decimal) and cannot be changed by any primitive.*

$$serial.\mathsf{Base} = 1016$$

The primitives serial_read and serial_write use $serial.$Base to map the I/O address to the registers. Requiring this value to be constant ensures that there is no misuse of I/O addresses.

Invariant 2 (valid I/O APIC maxIntr) *The number of entries in the redirection table of an I/O APIC is less than 239.*

$$0 \leq ioapic.\mathsf{maxIntr} < 239$$

The length of the redirection table in an I/O APIC varies depending on the hardware implementation. The actual value for a certain I/O APIC is static and stored in the I/O APIC Version Register[16:23], which corresponds to $ioapic.$maxIntr in our device model. This invariant guarantees that the actual value of $ioapic.$maxIntr is within a range.

Invariant 3 (valid I/O APIC masks length) *The length of the mask array in an I/O APIC should be exactly* $\mathsf{maxIntr} + 1$.

$$\mathsf{Zlength}(ioapic.\mathsf{masks}) = ioapic.\mathsf{maxIntr} + 1$$

For the purpose of saving space, the hardware implementation of an I/O APIC makes each entry in the redirection table to be a 64-bit register containing all the configuration items of a certain interrupt line, such as rtbl[i][0:7] for interrupt-vector, rtbl[i][8:10] for delivery mode, etc. In our abstract machine model, we extract each configuration item from all the entries and combine them into a list. This invariant requires the number of masks to be always equal to $\mathsf{maxIntr} + 1$.

Invariant 4 (valid I/O APIC irq length) *The length of the interrupt-vector in an I/O APIC should be exactly* $\mathsf{maxIntr} + 1$.

$$\mathsf{Zlength}(ioapic.\mathsf{irqs}) = ioapic.\mathsf{maxIntr} + 1$$

Invariant 5 (valid I/O APIC irq) *The value of each interrupt-vector in an I/O APIC ranges from* `0x10` *to* `0xFE`*.*

$$\forall n,\ 0 \le n \le ioapic.maxIntr \rightarrow 16 \le \mathsf{nth}\, n\ (ioapic.\mathsf{irqs})\ 0 \le 254$$

The value in the interrupt-vector is the IRQ number raised on the connected CPU. This value can be configured during the runtime, and should be within the specified range.

Invariant 6 (valid interrupt states) *If a device is not in the interrupt service mode (the execution is not in the interrupt handler), the interrupt controller observed by the device should not be in the interrupt service mode either. The state* in_intr *indicates whether the system is currently handling an interrupt. Note that it will be abstracted into corresponding logical states in the extended device objects during the interrupt refinement process to enforce our isolation policy.*

$$\mathsf{in\_intr} = \mathsf{false} \rightarrow (lapic.\mathsf{ISR} = \mathsf{None} \wedge ioapic.\iota = None)$$

7.2 Layer Interface DConsoleBufferIntro

On top of the underlay interface MBoot, we introduce and verify the circular console buffer (for the serial driver) using the strategy illustrated in Sect. 2.6. The layer interface DConsole-BufferIntro corresponds to the intermediate layer interface L_{mid} in Sect. 2.6. At this layer, we also introduce the following invariants on the intermediate console buffer (see Fig. 5) at this layer interface. Recall that, by the design of a layer interface, the invariants are required to hold at every moment of the system execution. Thus, these invariants can be used as facts in any part of the verification. Invariants are stronger notions than the preconditions in the specifications, in the sense that the preconditions need to be validated during the primitive call while the invariants are guaranteed to hold before and after calling the primitive.

Invariant 7 (valid console buffer positions)

$$0 \le d.rpos < \mathsf{CB\_SIZE} \wedge 0 \le d.wpos < \mathsf{CB\_SIZE}$$

As explained in Sect. 2.6, the contextual refinement from MBoot to DConsoleBufferIntro is achieved relatively easily as the abstraction in the current layer interface is extremely similar to the actual implementation. The concrete methodology is illustrated in Sect. 7.4, with a simpler example.

7.3 Layer Interface DAbsConsoleBufferIntro

This layer interface provides the higher level of abstraction of the console buffer as a Coq list and corresponds to the layer interface L_{high} in Sect. 2.6. Detailed definitions and the refinement proofs are omitted since they are already presented in Sect. 2.6. This layer interface introduces a new layer invariant on the abstract console buffer.

Invariant 8 (valid console buffer length)

$$0 \le \mathsf{Zlength}(serial.cons\_buf) \le \mathsf{CB\_SIZE}$$

```
(* Specification *)
Function set_serial_exist_spec (v: Z)
 (serial: Abs): option Abs :=
 if zeq 0 v then
   Some (serial {serial_exist: false})
 else if zlt_le 0 v Int.max_unsigned
     then
   Some (serial {serial_exist: true})
 else None

```

```
// Implementation
void set_serial_exist(unsigned int ex)
{
 if (ex > 0) {
  serial_exist = true;
 } else {
  serial_exist = false;
 }
}

```

Fig. 32 The specification and implementation of C function `set_serial_exist`

```
Inductive set_serial_exist_low_level_spec : genv -> list val -> mem -> option val ->
     mem :=
    | set_serial_exist_intro_true ge (m m': mem) (b: block) (v: int):
      find_symbol ge serial_exist = Some b ->
      0 < Int.unsigned v ->
      Mem.store Mint32 m b 0 (Vint Int.one) = Some m' ->
      set_serial_exist_low_level_spec ge (Vint v :: nil) m Vundef m'
    | set_serial_exist_intro_false ge (m m': mem) (b: block) (v: int):
      find_symbol ge serial_exist = Some b ->
      0 = Int.unsigned v ->
      Mem.store Mint32 m b 0 (Vint Int.zero) = Some m' ->
      set_serial_exist_low_level_spec ge (Vint v :: nil) m Vundef m'.
```

Fig. 33 The low level specification of C function `set_serial_exist`

7.4 Layer Interface DSerialIntro

The next three layer interfaces are designated to the verification of serial driver. On top of the underlay interface DAbsConsoleBufferIntro, we introduce a global variable serial_exist of type bool to indicate whether the serial device is initialized or not, and provide a getter and setter function called get_serial_exist, set_serial_exist, respectively. Accordingly, in DSerialIntro, we introduce a new abstract state serial_exist : bool and two new getter and setter primitives under the same names.

As an example, the source code (in C) and the specification (in Coq) of set_serial_exist() are shown in Fig. 32. Here, Abs is the type of abstract state in this layer interface, v is the argument of the primitive, and the boolean function zlt_le $a\ b\ c$ returns true if and only if $a < b \leq c$, as its name suggests. Furthermore, $d\{attr : val\}$ is our own notation defined in the Coq, which indicates a new abstract states derived from the original abstract states d, where the attribute $attr$ is replaced by the new value val, but otherwise the same as d. Note that the return type of the specification function is an option type, and the specification gets stuck (returns None) when the argument is not within the bound of 32-bit integer. This is the requirement that any future caller of this primitive needs to satisfy.

The function set_serial_exist runs on a machine with the underlay interface, and our goal is to prove the contextual refinement between the code running on the underlay and the abstract overlay interface DSerialIntro. The contextual refinement is proven in three steps.

First, we write a separate low level specification for the C code set_serial_exist with the low level memory load and store operations on the underlay interface, as shown in Fig. 33. Here, "Inductive" is the Coq keyword used to define a logical predicate inductively, with the vertical bar "|" used to separate different cases that can cause the relation to hold. Then the inductively defined predicate set_serial_exist_low_level_spec $ge\ args\ m\ rval\ m'$ indicates that under the global environment (ge) mapping global variable identifiers to their locations in the (CompCert) memory, given the argument list $args$, the function changes the memory from m to m' with the return value $rval$ (Vundef if no return value). Mem.store is an operation in the CompCert memory model; it takes the memory writing type, initial memory to write to, the memory block, block offset, and a value, and returns the new memory after writing the value on the location represented by the memory block and offset on the initial memory. As shown in the figure, the specification has two cases as in the source code, and in fact, it is very close to the source code. Thus, it is relatively easy to show that the source code satisfies the low level specification shown in Fig. 33 and the proof can be achieved nearly automatically by our proof tactics.

Second, we prove a simulation from the low level specification at underlay shown in Fig. 33 and the overlay specification shown in Fig. 32, with a refinement relation that trivially maps the value of the global variable in the memory at underlay to the new abstract state at overlay. This part of proof, which we call *data abstraction*, can vary depending on the kind of concrete data structures in memory and the abstracted states in the overlay interface. In the case of set_serial_exist, the proof is very simple given that it is just a single variable, but in other cases, the proof could be quite complex and we may need to further split the proof into multiple layers as shown in the example of the circular console buffer. The benefit is that after the data abstraction, the modules and layer interfaces built on top of that could benefit greatly from the simplicity in the abstracted form.

Last, we apply the correctness theorem of our CompCertX compiler to compile the C source code and its simulation proof from the first step into appropriate form in LAsm and link it with the simulation proof obtained in the second step, to derive the desired final contextual refinement theorem.

7.5 Layer Interface DSerial

On top of the underlay layer interface DSerialIntro, we introduce a new layer interface DSerial to abstract three driver functions for the serial device. They are serial_init that initializes the serial device, serial_getc and serial_putc that reads a character from and writes a character to the serial device, respectively. The implementation of serial_getc and serial_putc were shown in Fig. 22, and the C source code for serial_init is shown in Fig. 34.

A new layer invariant is introduced in DSerial to protect the configuration data.

Invariant 9 (valid serial state) *After initialization, the control registers in the serial device should be properly configured.*

$$\begin{aligned} serial.\mathsf{serial\_exist} = \mathit{true} \ \rightarrow \ & serial.\mathsf{Baudrate} = 115200 \ \wedge \ serial.\mathsf{Databits} = 8 \ \wedge \\ & serial.\mathsf{Stopbits} = 1 \ \wedge \ serial.\mathsf{Parity} = \mathit{NoParity} \ \wedge \\ & serial.\mathsf{RxIntEnable} = \mathit{true} \ \wedge \ serial.\mathsf{FIFO} = 1 \ \wedge \\ & serial.\mathsf{Modem} = \mathit{DTR} + \mathit{RTS} + \mathit{OUT2} \end{aligned}$$

Both serial_getc and serial_init do not contain any loops and are relatively easy to verify. Note that serial_init detects the existence of serial device, sets the proper configuration based on our hardware connection parameters, and uses set_serial_exist to set the abstract state serial_exist when the initialization succeeds.

In the rest of this subsection, we present details on the function serial_putc. As shown in Fig. 22, the implementation of serial_putc involves a loop to poll the status of TxBuf (through the serial_read primitive of serial device) until it is empty or reaches the maximum waiting steps. We assume the correctness of hardware. Thus, we have a separate assumption

```
void serial_init() {
    // Turn off interrupt.
    serial_write(COM1 + COM_LCR, 0); // set DLAB to zero
    serial_write(COM1 + COM_IER, 0);

    // Set DLAB.
    serial_write(COM1 + COM_LCR, COM_LCR_DLAB);

    // Set baud rate.
    serial_write(COM1 + COM_DLL, 0x0001ul);
    serial_write(COM1 + COM_DLM, 0x0000ul);

    // Set the line status.
    serial_write(COM1 + COM_LCR, COM_LCR_WLEN8_AND_DLAB);

    // Enable the FIFO.
    serial_write(COM1 + COM_FCR, 0xc7);

    // Turn on DTR, RTS, and OUT2.
    serial_write(COM1 + COM_MCR, 0x0b);

    // Serial COM1 doesn't exist if COM_LSR returns 0xFF.
    set_serial_exist (serial_read(COM1+COM_LSR, M_NIL) != 0xFF);

    // Clear any pre-existing overrun indications and interrupts.
    serial_read(COM1+COM_IIR, M_ALL);
    serial_read(COM1+COM_RX, M_ALL);
}
```

Fig. 34 The implementation of `serial_init` in C

on the event log stating the TxBuf always become empty before the maximum iteration is reached.

The specification of the abstract serial_putc primitive at the new DSerial layer interface puts the sending character into the serial device's TxBuf and updates the local log accordingly, as shown in Fig. 35. It first performs some validation checks, including whether the current execution status is in the critical mode, whether the serial device is initialized (line 4) and whether the device is properly configured (line 7). Then, it cases on whether the transmission buffer (TxBuf) is empty (line 8). Here, we use the notation $d.attr$ to indicate the attribute value with name $attr$ in the abstract state d, and $Z.eq\_dec$ is the boolean function for integer equality.

Empty buffer indicates that the device is ready to send more characters, and in this case, the primitive writes the proper contents to the abstract hardware buffer. Our serial communication protocol requires a new line ('`\n`') to be a sequence of LF and CR, to be compatible with most serial applications. The 'if' expression at line 10 and 14 serves for this purpose. The local log ℓ_{tx} only records the transmitting events, which separates from the receiving log ℓ_{rx}. Since we only consumed a single SendingCompAck event in this case, we simply fetch the next transmitting event from the environment log (ℓ^{env}) and update the transmitting log. Recall that function next is defined in the Sect. 4. It takes a local log, finds the first relevant event with respect to this log and returns a pair of event and the updated log.

When the buffer is non-empty, the device is still in the middle of transmitting messages and the execution needs to wait until the transmission is complete. In the implementation, there is a loop to poll the status of the transmitter, but the specification simply writes the TxBuf with the sending message and updates the log ℓ_{tx} to find the first SendingCompAck

```
Function serial_putc_spec (c: Z) (serial: Abs): option Abs :=
 match (serial.critical, serial.serial_exist) with
 | (true, true) =>
   match serial with
   | mkDevData
     (mkSerialState TxBuf _ _ true base _ false baud db sb par fifo modem) _ ltx _ =>
     if config_ok baud db sb par fifo modem then
     match TxBuf with
     | nil =>
       if Z.eq_dec c CHAR_LF
       then Some (serial {TxBuf: CHAR_LF::CHAR_CR::nil} {ltx: snd (next (serial.ltx))
      }})
       else Some (serial {TxBuf: c::nil} {ltx: snd (next (serial.ltx))})
     | _ =>
       if Z.eq_dec c CHAR_LF
       then Some (serial {TxBuf: CHAR_LF::CHAR_CR::nil}
                      {ltx: (next_sendcomplete (serial.ltx))})
       else Some (serial {TxBuf: c :: nil} {ltx: (next_sendcomplete (serial.ltx))})
     end
     else None
     ...

```

Fig. 35 The specification of `serial_putc` in Coq

```
Function putc_scan_log (t: log)(bound:
     nat): option log :=
 match bound with
 | 0 => None
 | S bound' =>
   match next (t, SerialEnv) with
   | (SendingCompAck, t') => Some t'
   | (_, t') => putc_scan_log t' bound'
   end
 end.

Definition next_sendcomplete (t: log) :=
 match putc_scan_log t 12800%nat with
 | None => nextk (t, 12800)
 | Some i => next i
 end.

```

Fig. 36 The specification of `next_sendcomplete` in Coq

event in ℓ^{env}. This is achieved through the next_sendcomplete fix-point function shown in Fig. 36. Note that nextk ℓ k is a function to call next k times and only returns the updated log.

Next, we need to prove the contextual refinement between the specification and the C code. To do that, we need to design a loop invariant that allows us to prove that once the loop terminates, the transmission buffer (TxBuf) is empty so that sending new messages to the serial will not overwrite the previous messages. Recall that because of our assumption on the correctness of the hardware, there is always an index t between 0 and 12800, at which iteration the loop terminates and serial device gets ready for the next write. Thus, we have designed the following loop invariant:

$$(0 \le i \le t \ \wedge \ serial_i = serial_0[\ell_{tx} \leftarrow \mathsf{nextk}\ (serial_0.\ell_{tx},\ \ i)] \ \wedge \ \mathsf{lsr} = 0) \ \vee$$
$$(i = t + 1 \ \wedge \ serial_i = serial_0[\ell_{tx} \leftarrow \mathsf{nextk}\ (serial_0.\ell_{tx},\ \ t+1)][\mathsf{TxBuf} \leftarrow nil] \ \wedge$$
$$\mathsf{lsr} = 1).$$

Here, $serial_i$ indicates the value of abstract state $serial$ after the i^{th} iteration of the loop, where $serial_0$ indicates the initial state before entering the loop. With this loop invariant, the proof can be achieved with the help from our automation tactic libraries.

```
void serial_intr_handler () {
  unsigned int hasMore, t = 0;
  hasMore = serial_getc ();
  while (hasMore == 1u && t < CB_SIZE) {
    hasMore = serial_getc ();
    t++;
  }
}
```

```
void cons_init() {
  cons_buf_init();
  serial_init();
}

```

Fig. 37 The implementation of `serial_intr_handler` and `cons_init` in C

```
Function serial_intr_handler_spec (serial: Abs): option Abs :=
  match (serial.serial_exist, serial.irq) with
  | (true, true)  =>
    let rxbuf = serial.RxBuf in
    let lrx = serial.lrx in
    match last 1 lrx with
    | Recv str =>
      if list_eq_dec Z.eq_dec str rxbuf) then
        if zle (Zlength (serial.cons_buf ++ rxbuf) CB_MAX_CHARS
        then Some serial
                  {cons_buf: serial.cons_buf ++ rxbuf}
                  {lrx: nextk (lrx, Zlength rxbuf * 2 + 1)}
                  {RxBuf: nil}
                  {irq: false}
        else Some serial
                  {cons_buf: skipn (Z.to_nat (Zlength (serial.cons_buf ++ rxbuf)
                                   - CB_MAX_CHARS)) (serial.cons_buf ++ rxbuf)}
                  {lrx: nextk (lrx, Zlength rxbuf * 2 + 1)}
                  {RxBuf: nil}
                  {irq: false}
        ...

```

Fig. 38 The specification of `serial_intr_handler` in Coq

7.6 Layer Interface DConsole

The layer interface DConsole introduces two new primitives. One is cons_init which initializes the serial and the console buffer. The other is the interrupt handler of the serial device serial_intr_handler. Their C implementations are shown in Fig. 37.

Once an interrupt signal is produced by the serial device indicating we have newly received characters in the receiving buffer (RxBuf), the interrupt handler repeatedly reads the characters one by one from RxBuf and writes them to the console buffer until either all characters are received or the console buffer is full of the newly received messages. It is achieved by repeatedly calling the serial_getc primitive (introduced at the DSerial) within a loop.

The specification of the serial_intr_handler primitive is shown in Fig. 38. In contrast to the C implementation in Fig. 37, the specification in Fig. 38 does not involve any fix points, but simply concatenates the whole receiving buffer (RxBuf) into the abstract console buffer list, skipping extra characters in the head of the list if necessary. This discrepancy between the implementation and the clean loop-free specification introduces extra complexity in proving the simulation between those two.

To prove the simulation between them, we first introduce a specification for the loop and prove the loop body refines this specification. This is used later to prove the whole function body containing the loop. The specification for the loop body is a predicate on the abstract state $serial$ and the local environment for storing the temporary variables in Clight, before $(serial, le)$ and after $(serial', le')$ the loop:

```
Definition cons_buf_mid (str: list Z) (cb: list Z) (n: Z) :=
 if (zle_le 0 n (Zlength str - 1))
 then if (zle (n + 1 + Zlength cb) CB_MAX_CHARS)
      then cb ++ firstn (Z.to_nat (n + 1)) str
      else skipn (Z.to_nat (n + 1 + Zlength cb - CB_MAX_CHARS))
                 (cb ++ firstn (Z.to_nat (n + 1)) str)
 else if (zle (Zlength str + Zlength cb) (CB_MAX_CHARS - 1))
      then cb ++ str
      else skipn (Z.to_nat (Zlength str + Zlength cb - CB_MAX_CHARS))
                 (cb ++ str).

```

Fig. 39 The specification of `cons_buf_mid` in Coq

$$
\begin{aligned}
&serial.\text{serial\_exist} = \text{true} \rightarrow\\
&serial.\text{RxBuf} = str \rightarrow\\
&\text{Forall isChar } str \rightarrow\\
&serial.(\text{cons\_buf}) = \text{cons\_buf\_mid } str\ serial.(\text{cons\_buf})\ 0 \rightarrow\\
&le[\text{hasMore}] = \text{true} \rightarrow\\
&le[\text{t}] = 1 \rightarrow\\
&(serial'.lrx = \text{nextk } (serial.lrx,\ \text{Zlength}(tl\ str) * 2 + 1) \wedge\\
&\ serial'.\text{cons\_buf} = \text{cons\_buf\_mid } str\ serial.\text{cons\_buf } (\text{Zlength } str)),
\end{aligned}
$$

where, tl is the standard list operation that retrieves the tail of a list, Forall isChar is an inductive predicate to describe the property that all the characters in str is a valid character (fun $x \rightarrow 0 \leq x \leq 255$), and cons_buf_mid $str\ cb\ n$ is a function shown in Fig. 39. This function calculates the console buffer when the first $n + 1$ characters in str is moved to cb.

The loop invariant is constructed as:

$$
\begin{aligned}
&(\ 0 \leq i \leq \text{Zlength}(tl\ str)\ \wedge\ le[\text{hasMore}] = 1\ \wedge\\
&\ \ serial_i.lrx = \text{nextk } (serial.lrx,\ \ i \times 2)\ \wedge serial_i.\text{RxBuf} = \text{skipn } (i+1)\ str\ \wedge\\
&\ \ serial_i.\text{cons\_buf} = \text{cons\_buf\_mid}\ \ str\ \ serial.\text{cons\_buf}\ \ i\ \ )\\
&\vee\\
&(\ \ i = \text{Zlength } str\ \ \wedge\ le[\text{hasMore}] = 0\ \wedge\\
&\ \ serial_i.lrx = \text{nextk } (serial.lrx,\ \ (\text{Zlength } str) \times 2 + 1)\ \wedge serial_i.\text{RxBuf} = nil\ \wedge\\
&\ \ serial_i.\text{cons\_buf} = \text{cons\_buf\_mid}\ \ str\ \ serial_i.\text{cons\_buf}\ \ (\text{Zlength } str)\ \ )
\end{aligned}
$$

This loop invariant has two parts: the first part shows the states during the loop; and the second part shows the states when the loop terminates.

7.7 Layer Interface DIOApic

The layer interface DIOApic is built on top of I/O APIC machine interface. Four primitives are introduced in DIOApic: ioapic_init, ioapic_enable, ioapic_mask, and ioapic_unmask.

The function ioapic_init initializes the I/O APIC device. Figure 27 shows the C implementation of this primitive. It includes a loop of calling `ioapic_write()` to set the interrupt vectors and masks. The specification of this primitive is shown in Fig. 40, which contains a fix-point to set related states. In the figure, the abstract state init indicates whether the device has been properly initialized. Every primitive other than the initialization primitive has the precondition on the init being true.

To prove the simulation between these two loops, we first prove the refinement between the loop body and the abstract `disable_irq`. Then we prove the simulation from the loop to the fix-point ioapic_init_aux with following loop invariant:

```
Function disable_irq (ioapic: Abs)
  (n: nat): Abs :=
  (ioapic {irqs[n]: T_IRQ0 + Z.of_nat n}
          {masks[n]: true}).

Fixpoint ioapic_init_aux (ioapic: Abs)
 (n: nat): Abs :=
 match n with
 | O => disable_irq ioapic 0
 | S n' =>
   let ioapic' := (ioapic_init_aux
     ioapic n') in
   disable_irq ioapic' n
 end.
```

```
Function ioapic_init_spec (ioapic: Abs)
 : option Abs :=
 match ioapic.init with
 | false =>
  let n := ioapic.MaxIntr + 1 in
  if zeq n (Zlength ioapic.irqs)
  then
   if zeq n (Zlength ioapic.masks) then
    let o := ioapic_init_aux ioapic (Z.
     to_nat (n - 1)) in
     Some (o {init: true})
   else
    None
   ...
```

Fig. 40 The specification of `ioapic_init` in Coq

$$0 \le i \le ioapic_0.\mathsf{maxIntr} \wedge ioapic_i = \mathsf{ioapic\_init\_aux}\ \ ioapic_0\ \ (Z.\mathsf{to\_nat}\ i)$$

From this loop invariant, we can show that when the loop completes, the final state is consistent with the one in the specification.

We also introduce a new invariant in this layer interface.

Invariant 10 (valid I/O APIC state) *After initialization, the value of interrupt vector corresponding to interrupt line n should be equal to n + IRQ0 and this interrupt line should be either masked or unmasked.*

$$\begin{aligned}
&init = \mathit{true} \rightarrow\\
&\forall n \in \mathbb{N},\ v \in \mathbb{Z},\ \ \mathit{nth\_error}\,(ioapic.\mathit{irqs})\ n = value\ v \rightarrow\\
&(v = Z.of\_nat\ n + \mathit{IRQ0}\ \wedge\ \exists\, b \in \mathbb{B},\ \ \mathit{nth\_error}(ioapic.\mathit{masks})\ n = value\ b)
\end{aligned}$$

In the mCertiKOS kernel, we allocate sequential entries from the interrupt descriptor table (IDT) for the IRQs. The range is from IRQ0 to IRQ0 + $ioapic$.maxIntr. IRQ0 is the first IRQ number which is also known as the global system interrupt (GSI). The initialization sets the correct values of interrupt vectors which match the IDT entries so that the correct interrupt handler can be called when an IRQ occurs. The mapping from interrupt lines to IRQ numbers is static in mCertiKOS, so this invariant protects this mapping on the I/O APIC.

Primitive ioapic_enable is used to set the routing configuration for an interrupt line in order to make it ready for serving the incoming interrupts. The implementation and specification of ioapic_enable are shown in Fig. 41. Because mCertiKOS only uses one mode of the interrupt delivery, we only check the validity of given parameters of $lapicid$, $trigger$, and $polarity$, but do not model the functionality of these parameters in our device. The ioapic_write primitive also requires these parameters to contain exactly the same values.

The primitives ioapic_mask and ioapic_unmask are used to mask and unmask a particular interrupt line designated for a device. It is not allowed to designate a single interrupt line for multiple devices. Thus, the masking status of each interrupt line could be later abstracted into the internal state of corresponding abstract device object to enforce our isolation policy. The verification of ioapic_mask and ioapic_unmask is similar to that of ioapic_enable.

The verification of the four functions introduced in DIOApic is reasonably simple, and can be automated using our tactics.

```c
void ioapic_enable (uint32_t irq,
uint32_t lapicid, uint32_t trigger_mode,
uint32_t polarity) {
 int maxintr = (ioapic_read (IOAPIC_VER)
                             >> 16)&(0xff);
 if (irq >= gsi && irq <= gsi + maxintr)
 {
  uint32_t i = IOAPIC_TBL + 2*(irq-gsi);
  ioapic_write (i, ((trigger_mode<<15) +
      (polarity<<13) + (T_IRQ0 + irq)));
  ioapic_write (i + 1, lapicid << 24);
 }
}
```

```coq
Function ioapic_enable_spec (irq: Z)
(lapicid: Z) (trigger: Z) (polarity: Z)
(ioapic: Abs): option Abs :=
 match ioapic.init with
 | true =>
   if config_ok irq lapicid trigger
     polarity then
    let idx := Z.to_nat irq in
    Some (ioapic
          {irqs[idx]: irq + IRQ0}
          {IoApicEnables[idx]: true})
   else None
   ...

```

Fig. 41 The implementation (in C) and specification (in Coq) of `ioapic_enable`

7.8 Layer Interface DLApic

Two primitives are introduced under the layer interface DLApic: lapic_init and lapic_eoi. The verification of lapic_init is similar to ioapic_init. Recall that during the later contextual refinement of interrupt models, the operations of two interrupt controllers will be merged and canceled to derive the abstract interrupt model, where an interrupt triggered by the serial device is only related to the serial device object, and is independent from either of the interrupt controllers or the CPU.

7.9 Above DLApic

Finally, we introduce a new layer interface to introduce the abstract interrupt model (as shown in Sect. 5.2.4), and more layer interfaces for the driver-related system calls.

8 Evaluation and Lessons Learned

What We Have Proved The final theorem we proved for our kernel is the contextual refinement relation between our lowest level hardware machine model `MBoot` (which defines the x86 instructions, the serial device, and the I/O APIC and Local APIC devices, etc.), and the top level machine `mCertiKOS` (which defines the abstract system call interface). Let $[\![\cdot]\!]_{\mathrm{x86}}$ and $[\![\cdot]\!]_{\mathrm{mCertiKOS}}$ denote the whole-machine semantics of each machine model, and K denote the (assembly) source code of mCertiKOS, then the theorem is formalized as:

Theorem 1 $\forall P,\ [\![K \Join P]\!]_{\mathrm{x86}} \sqsubseteq [\![P]\!]_{\mathrm{mCertiKOS}}$.

The theorem states that for any kernel/user/guest/host context program P, there is a simulation between program P running on top of the top level abstract machine `mCertiKOS`, and the program P linked with the mCertiKOS source code K, running atop the bottom-most machine `x86`.

The abstraction layers also define the data invariants that are proved to hold at any moment of the whole program execution. Some example invariants are: the console's circular buffer is always wellformed, and the interrupt controller states are always consistent, etc.

Besides this, our framework automatically derives that all the system calls always run safely and terminate; there are no code injection attacks, no buffer overflows, no null pointer access, no integer overflows, etc.

Isolation We take the existing implementation of the CertiKOS infrastructure [20], and extend it with our device and interrupt models. On top of the extended machine model, we have verified a subset of the device drivers in mCertiKOS with 10 abstraction layers. Some layers are introduced to verify concrete driver implementation, while others are introduced purely for logical abstraction (e.g., from a circular console buffer implementation in memory to an abstract list, from the hardware interrupt model to the abstract interrupt model enforcing isolation, etc). These abstraction layers are inserted into the existing layers of mCertiKOS as a certified plugin. Thanks to our isolation policy, this does not invalidate most of the existing proofs of mCertiKOS, and the integration only required minimal effort, despite the existing mCertiKOS proofs being unaware of interrupts.

Execution Model and Completeness The majority of our device drivers are specified and verified at C level, then compiled by our CompCertX compiler. The entire kernel (both C and assembly) source code, together with the source code for the verified compiler, are extracted into an OCaml program through Coq's extraction mechanism. When this program gets executed, it compiles the extracted C source code into the assembly, and merges it with the existing assembly kernel source code, to produce a piece of assembly code corresponding to our verified kernel. Thus, our deliverable comes with a piece of assembly code for the entire verified kernel, a high level deep specification of various kernel behaviors, and a machine checkable proof object stating the assembly code running on the actual hardware satisfies the high level specification.

The verified assembly code is then linked with the rest of kernel code (the boot loader and remaining unverified drivers) to produce the actual binary image of the OS. The resulting kernel is practical: it runs on stock x86 hardware and the hypervisor version with the Intel vt-x support can successfully boot an unmodified version of Linux as guest.

Verification Effort Using our general device interface, we have modeled a serial device and two interrupt controller devices. On top of these device models, we have verified the related drivers and interrupt handlers. The entire verification effort consists of roughly 20k lines of Coq code added to the existing mCertiKOS verification code base. Regarding the specification, there are 510 lines of code used to specify the machine model including the device hardware, and 126 lines of code for specification of the additional system call interfaces. There are additional 9829 lines of Coq code that were used to define auxiliary definitions, lemmas, theorems, invariants, *etc*. Note that these 9829 lines of definitions are outside our TCB, thus does not need to be trusted. In terms of proof size, there are 3671 lines of Coq code for the layer refinement proofs, 3589 lines for code verification, 1802 lines for proving invariants, and 307 lines for linking different modules together.

The entire verification effort took roughly 7 person months, the majority of which went into the design and development of the framework itself, including the extended machine model, general device framework, the interrupt refinement, and the tactic libraries for automating most of the non-intellectual parts of verification task. We anticipate the cost of verification for future drivers would be dramatically reduced.

Bugs Found An extended version of the mCertiKOS kernel has been deployed in a practical system that is used in the context of a large DARPA-funded research project [20]. Yet, through the verification of the console driver, we found a critical bug which may lead to the loss of many characters received from the serial device. The bug was in the implementation of the circular console buffer, where, in some rare cases, the read and write positions to the buffer array overlap, causing the entire contents in the buffer to be lost. The bug was caught when we tried to establish the contextual refinement between the concrete implementation of the circular buffer and its abstract list representation.

Another bug was found in the code for initializing the serial device, where the interrupt was not configured correctly by accidentally setting the Interrupt Enable Register (IER) before the DLAB was unset. This was caught when we tried to prove the initialization code against its specification.

9 Limitations and Future Work

Our verified kernel assumes correctness of the hardware. In our device model, we enforce a set of invariants on the list of external events, which specifies correct hardware behaviors, e.g., all the 8 bit characters are 8 bits, serial port eventually transmits its contents, etc. Every function that tries to write to the serial device first busy-waits reading the device's transmission buffer status until it becomes empty. We rely on the above assumption to prove that the loop eventually terminates and when it does terminate, the transmission buffer is empty so we can write to the device again. In the future, we plan to extend our device drivers to handle the hardware errors, e.g., when the serial device does not acknowledge the previous output was successful in the time period specified in the hardware documentation. In this case, we can add states to the device state machine to represent those erroneous cases, and add appropriate error handling code. The process is the same as a non-faulty device. For example, when the serial port does not transmit its contents in a certain amount of time, we can reset the serial port and try again.

Furthermore, as with any verified system, the specification of hardware devices and the top level system call primitives have to be trusted. For the hardware specification, we only model the set of features utilized by the kernel, instead of modeling the entire hardware manual. Our system calls are specified at the top abstraction layer, where all implementation details are hidden. These lead to specifications of a fairly small size (636 lines of Coq code), limiting the possible room for errors, and easing the review process.

Sometimes, the compiler may unsoundly optimize away some memory accesses to the memory mapped registers, e.g., a dead read of a memory mapped device register. In this case, we can use the CompCert built-in calls like `volatile_load`, which are not supposed to be optimized away by CompCert. On the other hand, those operations can also be directly implemented in assembly in our framework.

Some parts of the TCB from the original mCertiKOS still remain, including the bootloader, the Coq proof checker, and the pretty-printing phase of the CompCert compiler.

Verification of Other Drivers Some device drivers (i.e., those with underlined names in Fig. 7) in mCertiKOS still remain unverified. With the new compositional framework and automation libraries we have developed, we anticipate that the rest of the drivers can be verified with a reasonable amount of proof engineering effort.

Among those drivers shown in Fig. 7, the text-mode VGA driver can be verified easily since it is not much more complex than the serial driver. The timer and TSC drivers can also be verified, but mCertiKOS's assembly machine must first be parametrized with a good cost model for x86 instructions.

The disk driver (including the PCI and AHCI drivers) is the largest driver in our kernel. The mCertiKOS kernel communicates with the hard disk through the AHCI controllers using memory mapped registers (mCertiKOS also communicates with the APIC using memory mapped registers through the verified drivers). We believe that our device model is general enough to model required features for these devices used by the disk driver. We have already started applying our approach to verify the mCertiKOS disk driver which will also serve as a basis for building a certified file system.

Concurrency Reasoning Our current certified kernel assumes a runtime environment consisting of a single processor, and user processes do not preempt each other. Therefore, our work so far does not support preemptive nor multicore concurrency. With general concurrency, different user/kernel threads may share memory and use a wide variety of synchronization mechanisms that must also be verified. The techniques presented in this paper do not provide such support (since the logical CPUs for devices and the main kernel/user CPU do not share any state).

This would mean that we sometimes need to poll the device states if not doing so would cause us to handle shared-memory concurrency. For example, a user process tries to read a character from the serial device needs to poll the device status within a loop until the device's receiving buffer becomes non-empty. There have been recent efforts on adding general concurrency support to mCertiKOS [21] and we believe that our device framework can be faithfully merged into the event-based model introduced in [21]. With concurrency support, each logical CPU will have its own (logical) scheduler, (logical) memory, and collection of kernel or user threads that may share memory.

Device drivers often do need to modify kernel memory, as in Linux bottom halves (implemented as low-priority threads) or deferred procedure calls in Windows. With support of these "concurrency-aware" logical CPUs, we believe that our technique can be extended to support low-priority kernel threads dedicated to serve Linux bottom-halves. The idea is to treat these device-serving kernel threads (and memory) as part of the logical CPU dedicated for each device. Since we are already treating device driver code as if it runs on its "device" CPU, it is quite natural to place those device-serving kernel threads on the logical (device) CPUs as well.

10 Related Work

Gu et al. [20] pioneered the compositional proof machinery that builds certified OS kernels using deep specifications and certified abstraction layers. We built our certified interruptible OS kernel and device drivers using the same methodology. Our new compositional proof framework, however, adds two novelties. First, we show how to handle device objects, which are different from regular mCertiKOS kernel objects. The states in these new device objects can be updated either by the kernel (via device methods) or by an external environment, whereas regular mCertiKOS objects can only be mutated synchronously by the CPU; device objects can also be asynchronously mutated by the environment; we introduce a new abstraction of per-device event logs to handle this asynchronicity. Second, we support formal reasoning about kernel code and device drivers running on multiple logical CPUs (see Fig. 10) while under Gu et al. [20], all verified code at each layer must run on a single CPU (see Fig. 8); we treat the driver stack for each device as if it were running on the logical CPU dedicated to that device.

Klein et al. [27] were the first to verify the correctness and security properties of a high-performance L4-family microkernel in a modern mechanized proof assistant [35]. To make verification easier, they introduced an intermediate executable specification to hide

C specifics. Gu et al. [20] built their certified mCertiKOS kernel (in Coq) by decomposing it into many abstraction layers; such fine-grained layer decomposition led to significantly lower proof and development effort and also better extensibility. Both kernels, however, lack a realistic interrupt model, so reasoning about interruptible code is not supported. The device drivers are not verified in either kernel.

Hawblitzel et al. [22] has recently developed a set of new tools based on the Dafny verifier [29] and Z3 SMT solver [14], and applied them to build their Ironclad system which includes a verified kernel (based on Verve [42]), verified drivers, verified system and crypto libraries, and several applications. This is another impressive effort that advances the frontier of system software verification. However, the abstract device model in Ironclad is too high level to model many hardware details.

The Verisoft team [34] has done a large body of work aiming to verify an OS kernel with device drivers in a proof assistant [1,2,4]. Alkassar and Hillebrand [2] reported their work on verification of device driver, which investigated the relationship between external events, device and processor execution, and proved several non-trivial lemmas that can be viewed as a foundation of this work. In their work, the execution of CPU and devices are modeled as a combined transition system with an oracle to select the next one to make a step. The transition of each step is categorized into three cases: (1) processor-device transition, (2) local processor transition, and (3) external device transition. The paper shows that: (a) the steps of local processor and external device transition can be swapped because they do not interfere with each other; (b) the steps of external device transition by other devices can be reordered to the end of the execution because they do not change the state of the current device with the assumption that all the drivers only access to their own devices. Therefore, the device transitions triggered by external events can be moved from the code of high-level language which do not access the devices into the driver code which is written in assembly. In addition, the specification of the driver code which involves processor and device steps can be called atomically. They further proved the correctness of a simple ATAPI disk driver. In our paper, the "logical" CPUs are systematically built in an isolated way, so that the assumption that drivers only access to their own devices are guaranteed by construction. Furthermore, viewing drivers as more abstract devices enables us to add device accessing primitives into the device "ISA", and encapsulate the transitions of devices layer by layer into richer yet atomic specifications. Last, interrupt is allowed when the driver is running. We use intr_disable/intr_enable to mark the specific regions as critical sections, so that the interrupt being disabled is not an assumption of our driver code.

Based on the ideas of concurrent separation logic (CSL), Alkassar et al. [3] presented a modular and polymorphic method to specify IPC algorithms in VCC, which relies on transferring ownership of the ghost objects between IPC entities and attached invariants to guarantee the correctness of the algorithm. They extended their specification pattern to specify and verify an inter-processor interrupt (IPI) protocol in multiprocessor systems, in which IPC mailboxes are used to model the APIC bus between processors, and the IPI sending and receiving (NMI interrupt handler) code are modeled as while- loops. They focused on the interaction between the IPI participants locally, but not between the triggered interrupt handler and the previous execution on its CPU. In our device driver verification, we also address the interleavings between normal execution and interrupt handler on the same processor. In addition, the verification in [3] does not prove strong contextual refinement property as in our paper.

The idea of shuffling the execution of interrupt handlers to a certain point so that the verification can be done at higher level languages is discussed in Pentchev's PhD thesis [36]. Their concurrent machine $MIPS_P$ was modeled as an automaton with a sequence of stepindicators as external inputs. At each step, the transition function makes a case distinction based on an external input pair (component i.e. core/ipi/guest/vmexit, processor) and let one or multiple components to make a step. In order to propagate properties from the C Intermediate Language (C-IL) program to the $MIPS_P$ execution, they applied order reduction to prove the sequential compiler consistency (a simulation relation to achieve that the execution of interrupt handlers only happens between the C statements) by having interleaving occur at consistent state, namely interleaving points. In contrast to our paper, because the IPIs are non-maskable, the interrupt handler has to be carefully designed to avoid the race condition, and they only provided the semantics of concurrent C-IL encapsulated with IPI to specify the interrupt handler. In our paper, we push it further to the boundary of critical sections. We do not consider the interleaving between the kernel and device execution at such early stage. We view them as separated logical machine and design their own transition functions. In addition, the states among logical CPUs are isolated and are not visible to each other. The consistency of the states between a driver and its interrupt handler is guaranteed by examining the same local log to construct all shared states. We enforce a principle of a common programming pattern that drivers have to disable the interrupt before entering into the critical section, so that the "interleaving points" in our paper is only the intr_disable and intr_enable. Thus, above a certain layer, the code verification can not only be done at Clight level, but also be free of considering the interleaving between the code and interrupt handlers, because they are all encapsulated in the primitive of those functions accessing shared states.

Feng et al. [17,18] developed a formal Hoare-logic-like framework for certifying low-level system programs involving both hardware interrupts and preemptive threads. Using ideas from concurrent separation logic [33], they showed how to use ownership-transfer semantics to model enabling and disabling interrupts and reason about the interaction among interrupt handlers, context switching, and synchronization libraries. They successfully certified a preemptive thread implementation (as libraries) and a set of common synchronization primitives in the Coq proof assistant. Their work, however, did not model any hardware device or interrupt controller, and their interrupt model is much simpler than ours. They also only proved the partial correctness property (for their certified library functions), not the strong contextual refinement property which we proved for our kernel. Of course, since our current certified kernel does not support preemptive concurrency, we believe there are good opportunities for combining their techniques (for reasoning about preemptive concurrency) with our refinement-based approach.

Ryzhyk et al. [37,38] have done much work on the synthesis of device drivers from the specifications. In their approach, both the device and the interface of the corresponding driver are modeled as state machines, which communicate via messages. The generated driver code requires some unverified run-time support. Furthermore, the correctness of the drivers is limited to the synthesized C programs, not the compiled assembly code running on the actual hardware.

In the work of Duan and Regehr [16], a UART driver in the ARM architecture with interrupt is verified. They have created an abstract device model which gets plugged into the instruction set of the ARM6 architecture. In their model, the device state is mixed into the machine state. Thus, they have to carefully consider the interleavings between the execution of the device and the CPU. Albeit a realistic UART model, the driver only consists of 20 lines of the assembly code. The framework is later ported to the Cambridge model of the ARMv7 architecture [15]. Schwarz et al. [39] proposed a device model where all the devices are executed nondeterministically in parallel with a single core processor. Based on the model, they have proved several noninterference properties among the processor and devices which potentially use DMA or interrupts. Monniaux et al. [13] have verified a driver with a USB

OHCI controller model written in C with a static analyzer. They have showed the verified driver exhibits no undefined behavior.

Andronick et al. [6,7] presented a scalable framework for formally reasoning an embedded, real-time operation system: eChronos. In eChronos, OS functions are treated as interrupts, which are wrapped with supervisor calls (software triggered interrupts) even in the same address space. An extended Owicki-Gries approach is used to model and verify the system as all the concurrency components, such as tasks, interrupt handlers, supervisor calls, and asynchronous hardware behaviors, which are composed in parallel. Furthermore, in order to show the controlled interleaving allowed by the hardware, a ghost variable "AT" is introduced to represent the current active task. The correctness of eChronos is proven by properties (expressed by invariants) held through all executions and at every reachable step. Because of the non-determinism introduced by parallel composition, it is challenging to write the specification in an atomic way so that refinement can be applied to prove the functional correctness.

There are many lines of work in verifying device drivers based on model checking. Amani et al. [5] proposed an approach to automatically verify the protocols between drivers and the operating system. Thomas Witkowski [41] and Alexey Khoroshilov [25] have verified specific protocols of some Linux drivers using the model checker SATABS and DDVERIFY. Kim et al. [26] have verified a driver for a flash memory in NuSMV, Spin, and CBMC. Ball et al. [9] have developed the static analysis tool SLAM, which is included in the Microsoft Windows Driver Developer Kit.

11 Conclusions

We have presented a novel compositional framework for reasoning about the end-to-end functional correctness of device drivers in a certified interruptible kernel. Our formalization of interrupts follows the abstraction-layer-based approach and includes a realistic hardware interrupt model and an abstract model of interrupts (which is suitable for reasoning about interruptible code). We have proved that the two interrupt models are contextually equivalent. We have successfully extended an existing verified non-interruptible kernel with our framework and turned it into an interruptible kernel with verified device drivers. The implementation, specification, and proofs are all done in a unified framework (realized in the Coq proof assistant), yet the mechanized proofs verify the correctness of the assembly code that can run on the actual hardware. To the best of our knowledge, this is the first verified interruptible operating system with device drivers.

Acknowledgements We thank Quentin Carbonneaux, Hernán Vanzetto, Mengqi Liu, Jérémie Koenig, other members of the CertiKOS team at Yale, and anonymous referees for helpful comments and suggestions that improved this paper and the implemented tools. This research is based on work supported in part by NSF Grants 1065451, 1319671, and 1521523 and DARPA Grants FA8750-12-2-0293 and FA8750-15-C-0082. Hao Chen's work is also supported in part by China Scholarship Council. Any opinions, findings, and conclusions contained in this document are those of the authors and do not reflect the views of these agencies.

References

1. Alkassar, E.: OS verication extended: on the formal verication of device drivers and the correctness of client/server software. PhD thesis, Saarland University, Computer Science Department (2009)

2. Alkassar, E., Hillebrand, M.A.: Formal functional verification of device drivers. In: Proceedings of the Verified Software: Theories, Tools, Experiments Second International Conference (VSTTE), Toronto, Canada, pp. 225–239 (2008)
3. Alkassar, E., Cohen, E., Hillebrand, M., Pentchev, H.: Modular specification and verification of interprocess communication. In: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, FMCAD Inc, Austin, TX, FMCAD '10, pp. 167–174 (2010a)
4. Alkassar, E., Paul, W., Starostin, A., Tsyban, A.: Pervasive verification of an OS microkernel: inline assembly, memory consumption, concurrent devices. In: Verified Software: Theories, Tools, Experiments (VSTTE 2010), Edinburgh, UK, pp. 71–85 (2010b)
5. Amani, S., Chubb, P., Donaldson, A., Legg, A., Ryzhyk, L., Zhu, Y.: Automatic verification of message-based device drivers. In: Systems Software Verification, Sydney, Australia, pp. 1–14 (2012)
6. Andronick, J., Lewis, C., Morgan, C.: Controlled Owicki-Gries concurrency: reasoning about the preemptible eChronos embedded operating system. In: van Glabbeek RJ, Groote JF, Höfner P (eds) Workshop on models for formal analysis of real systems (MARS 2015), Suva, Fiji, pp. 10–24 (2015)
7. Andronick, J., Lewis, C., Matichuk, D., Morgan, C., Rizkallah, C.: Proof of OS Scheduling Behavior in the Presence of Interrupt-Induced Concurrency, pp. 52–68. Springer, Berlin (2016)
8. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006, ACM, New York, NY, USA, EuroSys '06, pp. 73–85 (2006)
9. Ball, T., Bounimova, E., Kumar, R., Levin, V.: SLAM2: Static driver verification with under 4% false alarms. In: Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design, FMCAD Inc, Austin, TX, FMCAD '10, pp. 35–42 (2010)
10. Blazy, S., Leroy, X.: Mechanized semantics for the Clight subset of the C language. J. Autom. Reason. **43**(3), 263–288 (2009)
11. Chen, H., Wu, X.N., Shao, Z., Lockerman, J., Gu, R.: Toward compositional verification of interruptible OS kernels and device drivers. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, New York, NY, USA, PLDI '16, pp. 431–447 (2016)
12. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.: An empirical study of operating systems errors. In: Proceedings of the 18th ACM Symposium on Operating Systems Principles, ACM, New York, NY, USA, SOSP '01, pp. 73–88 (2001)
13. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08), pp. 337–340 (2008)
14. Duan, J.: Formal verification of device drivers in embedded systems. PhD thesis, University of Utah (2013)
15. Duan, J., Regehr, J.: Correctness proofs for device drivers in embedded systems. In: Proceedings of the 5th International Conference on Systems Software Verification, USENIX Association, Berkeley, CA, USA, SSV'10, p. 5 (2010)
16. Feng, X., Shao, Z., Dong, Y., Guo, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. In: Proceedings of the ACM Conference on Programming Language Design and Implementation, pp. 170–182 (2008)
17. Feng, X., Shao, Z., Guo, Y., Dong, Y.: Certifying low-level programs with hardware interrupts and preemptive threads. J. Autom. Reason. **42**(2–4), 301–347 (2009)
18. Ganapathi, A., Ganapathi, V., Patterson, D.: Windows XP kernel crash analysis. In: Proceedings of the 20th Conference on Large Installation System Administration, USENIX Association, Berkeley, CA, USA, LISA '06, pp. 12–12 (2006)
19. Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X., Weng, S.C., Zhang, H., Guo, Y.: Deep specifications and certified abstraction layers. In: Proceedings of the 42nd ACM Symposium on Principles of Programming Languages, pp. 595–608 (2015)
20. Gu, R., Shao, Z., Chen, H., Wu, X.N., Kim, J., Sjöberg, V., Costanzo, D.: Certikos: An extensible architecture for building certified concurrent os kernels. In: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, USENIX Association, Berkeley, CA, USA, OSDI'16, pp. 653–669 (2016)
21. Hawblitzel, C., Howell, J., Lorch, J.R., Narayan, A., Parno, B., Zhang, D., Zill, B.: Ironclad apps: end-to-end security via automated full-system verification. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (2014)
22. Intel: 82093AA I/O advanced programmable interrupt controller (I/O APIC) datasheet. Specification (1996)
23. Intel: Multiprocessor specification, version 1.4. Specification (1997)

24. Khoroshilov, A., Mutilin, V., Petrenko, A., Zakharov, V.: Establishing Linux driver verification process. In: Pnueli, A., Virbitskaite, I., Voronkov, A. (eds.) Perspectives of Systems Informatics. Lecture Notes in Computer Science, vol. 5947, pp. 165–176. Springer, Berlin (2010)
25. Kim, M., Choi, Y., Kim, Y., Kim, H.: Formal verification of a flash memory device driver - an experience report. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) Model Checking Software. Lecture Notes in Computer Science, vol. 5156, pp. 144–159. Springer, Berlin (2008)
26. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP), Big Sky, MT, US, pp. 207–220 (2009)
27. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an OS microkernel. ACM Trans. Comput. Syst. **32**(1), 2 (2014)
28. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Proceedings of the Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2010), pp. 348–370 (2010)
29. Leroy, X.: The CompCert verified compiler. http://compcert.inria.fr/ (2005–2013)
30. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformation. J. Autom. Reason. **41**(1), 1–31 (2008)
31. Lynch, N.A., Vaandrager, F.W.: Forward and backward simulations: I. Untimed systems. Inf. Comput. **121**(2), 214–233 (1995)
32. Monniaux, D.: Verification of device drivers and intelligent controllers: a case study. In: Kirsch C, Wilhelm, R. (eds.) Proceedings of the 7th ACM International Conference On Embedded Software, EMSOFT 2007, pp. 30–36. ACM & IEEE (2007)
33. O'Hearn, P.W.: Resources, concurrency and local reasoning. In: Proceedings of the 15th International Conference on Concurrency Theory (CONCUR'04), pp. 49–67 (2004)
34. Paul, W., Broy, M., In der Rieden, T.: The Verisoft XT Project. http://www.verisoft.de (2007)
35. Paulson, L.C.: Isabelle: A Generic Theorem Prover, Lecture Notes in Computer Science, vol. 828. Springer (1994)
36. Pentchev, H.: Sound semantics of a high-level language with interprocessor interrupts. PhD thesis, Saarland University, Computer Science Department (2016)
37. Ryzhyk, L., Chubb, P., Kuz, I., Le Sueur, E., Heiser, G.: Automatic device driver synthesis with Termite. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP), Big Sky, MT, US, pp. 73–86 (2009)
38. Ryzhyk, L., Walker, A.C., Keys, J., Legg, A., Raghunath, A., Stumm, M., Vij, M.: User-guided device driver synthesis. In: USENIX Symposium on Operating Systems Design and Implementation, Broomfield, CO, USA, pp. 661–676 (2014)
39. Schwarz, O., Dam, M.: Formal verification of secure user mode device execution with DMA. In: Yahav, E. (ed.) Hardware and Software: Verification and Testing, Lecture Notes in Computer Science, vol. 8855, pp. 236–251. Springer (2014)
40. The Coq development team: The Coq proof assistant. http://coq.inria.fr (1999–2016)
41. Witkowski, T.: Formal verification of Linux device drivers. Master's thesis, Dresden University of Technology (2007)
42. Yang, J., Hawblitzel, C.: Safe to the last instruction: automated verification of a type-safe operating system. In: Proceedings of the 2010 ACM Conference on Programming Language Design and Implementation, pp. 99–110 (2010)