# Formal Functional Verification of Device Drivers

Eyad Alkassar[1,⋆] and Mark A. Hillebrand[2,⋆]

[1] Saarland University, Saarbrücken, Germany
eyad@wjpserver.cs.uni-sb.de
[2] German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany
mah@dfki.de

**Abstract.** We report on the formal functional verification of a simple device driver for an ATAPI hard disk in Isabelle/HOL. The proof is based on a functional model of the hard disk, which has been integrated into the instruction set architecture of a verified RISC processor as one of several memory-mapped devices. The result is an interleaved computational model, in which the devices and the processor take turns in execution. Even in this concurrent context, the verification can be kept largely sequential and modular with respect to the other devices. This is made possible by sound reordering of computation traces, given that devices do not interfere with each other and the driver monopolizes the hard disk. To the best of our knowledge, this paper presents the first formal functional verification of a device driver against a realistic device and system model.

## 1 Introduction

The Verisoft project deals with formal pervasive verification, attempting the complete verification of several example computer systems [1]. The systems being considered employ I/O devices for non-volatile storage, communication, or user interaction. The devices are integrated at the hardware level as memory-mapped devices and controlled via device drivers running in system or user mode.

When pervasively verifying the correctness of such systems, a stack of formal computational models has to be built, reflecting the structure of the implementation; implementations in one layer must simulate the model of the next higher layer. All models in the stack must include formal models of devices. From one level to the next the representation of a certain device may change due to device drivers abstracting device behavior (consider, e.g., a hard disk versus a file system). The timing behavior of a device is usually not modeled exactly even in its most concrete representation and devices often interact with an external

⋆ Work funded by the German Research Foundation (DFG) within the program 'Performance Guarantees for Computer Systems' and by the German Federal Ministry of Education and Research (BMBF) under grant 01 IS C38.

N. Shankar and J. Woodcock (Eds.): VSTTE 2008, LNCS 5295, pp. 225–239, 2008.

environment like a user or a network. Thus, the computational models in the stack have to be non-deterministic and concurrent.

The correctness statement of a device driver should be formulated with respect to the concurrent formal model it is implemented in. For the complete verification of the driver, not only this statement has to be shown but the correctness of all underlying device drivers as well. For user-level device drivers and for models in which devices are not accessible directly, typically several layers have to be considered, requiring simulation proofs between two or more concurrent computational models.

In this paper we consider the verification of a simple device driver for an ATAPI hard disk. We formulate and prove the correctness statement of the device driver in an assembly semantics with only a single device, the disk, visible. By reordering computation traces, we generalize this result to assembly computations involving other devices and lift it to the next-higher model, a C-like language providing device access only by calls to assembly drivers. Reordering exploits commutativity to sequentialize interleaved executions. To apply the theorems, different devices and drivers must be shown not to interfere with each other.

The theory and proofs above have been formalized in Isabelle/HOL [2].[1] To the best of our knowledge, this paper presents the first formal functional verification of a device driver against a realistic device and system model. The theorems of sound reordering of computation traces, which have been used in this proof, are useful for the verification of other device drivers as well and not specific to the considered verification example.

The remainder of this paper is structured as follows. In Sect. 2 we introduce device models and integrate them into larger computational models, i.e., here mainly an assembly semantics with devices. Assuming that devices are exclusively controlled by their respective drivers, we show in Sect. 3 how to transfer the correctness of a driver with respect to its device to the correctness of the driver in the full, running system. Moreover, we show that exclusive control of a device by its driver usually enables one to abstract from device and driver when going up one level in the stack of computational models. Thus, devices and their drivers have a simpler representation for client code. In Sect. 4 we present the device model of a hard disk, based on a subset of the ATAPI standard [3]. In Sect. 5 we present a simple hard disk driver for the disk, which writes a single page from memory to the hard disk. We sketch the formal correctness proof of the driver in a model with a single device. We then show how to generalize this results to the full system using the theory presented in Sect. 3. In Sects. 6 and 7 we present future work and conclude.

**Related Work.** Two earlier Verisoft publications are relevant for our work. We have reported on paper-and-pencil models and proofs related to a simple disk driver [4]. We extend this work in two important ways: models and proofs are formalized in Isabelle/HOL, and the models are now concurrent instead of lockstep. Thus, they are not restricted to disks, which are 'simple' for lack of external

[1] Theory files are available at `http://www.verisoft.de/VerisoftRepository.html`.

communication. In [5] we have reported on formal models of a serial interface and an architecture with devices, but not treated drivers. Here, we formally prove (disk) driver correctness up to the model of a high-level programming semantics.

So far most other device related verifications have either targeted the correctness of gate-level implementations or safety properties of drivers. In approaches of the former kind, simulation- and test based techniques are used to check for errors in the hardware designs. In particular, [6, 7] deal with serial interfaces in that manner. In approaches of the latter kind the driver code is usually shown to guarantee certain API constraints of the operating system and hence cannot cause system crashes. For example, the SLAM project [8] provides tools for the validation of safety properties of drivers written in C. SLAM's success led to the deployment of the Static Driver Verifier (SDV) as part of the Windows Driver Foundation [9]. SDV automatically checks 65 safety rules concerning the Windows Driver API for device drivers. Hallgren *et al.* [10] modeled device interfaces for a simple operating system written in Haskell. Three memory-mapped I/O calls were specified: read, write, and test for valid region. However, the only correctness property being stated is the disjointness of the device address spaces.

In contrast to all mentioned approaches, we aim at the formalization and *functional* verification of drivers interacting with a device. Thus, it is not sufficient to argue about the device or programming model alone. Even in other ongoing systems verification projects, the L4.verified project [11] and the FLINT project [12], device behavior and driver correctness are not considered. To our knowledge, the only work similar in scope is the challenge proposed by Holzmann [13] dealing with the formal verification of a file system for a Flash device. In response to the challenge, Woodcock reports on the partial specification of the file system (the 'file store') and a refinement proof mapping the store to a Java program [14]. Simultaneously, the Flash hardware is being formalized [15]. Verifying a low-level Flash driver and integrating it into the filesystem proofs are future work. Concurrency is not an issue since only a single device is considered.

Reordering execution sequences to obtain atomic specifications have been well studied in literature under the topic of *reduction theorems*. Lipton proved safety properties of pre-/ post-condition style sequentially and propagated these to the implementation [16]. Cohen and Lamport extended this to liveness and a more fine-grained analysis of the reordered parts of the sequence [17, 18]. Most reduction theorems assume that the implementation fulfills some non interference theorems. In contrast we prove this assumption on the atomic specification by exploiting a similar insight as reported in [19]. Justified by the memory mapped I/O architecture, the theory presented here is a specialization, enabling us to formulate even stronger reduction theorems than reported in literature.

## 2 Device Models and Models with Devices

Reasoning about driver correctness requires a detailed programming model. In our case, the driver is executed as an assembly program on a RISC instruction set architecture (ISA). A formal transition system of the ISA is first defined and its interface to devices is described. We proceed with an abstract device

model suitable for the modeling of memory-mapped devices; currently, we do not support device direct memory access (DMA). By combining the previous models we obtain a model where the processor and several devices run concurrently. We model this by introducing an oracle input called *event*, which determines whether some device or the processor takes the next step.

The concurrent model also serves as the specification of a concrete gate-level implementation of a processor with devices, which is an accurate model of the hardware. A simulation proof between these two models is presented in [20].

**Processor Model.** The processor model is the sequential programming model of the hardware as seen by a system software programmer. Machine configurations are 5-tuples $c_{\mathrm{P}} = (pc, dpc, gpr, spr, m)$ with the following components: the normal and the delayed program counters $c_{\mathrm{P}}.pc$ and $c_{\mathrm{P}}.dpc$ used to implement delayed branch, the general purpose register file $c_{\mathrm{P}}.gpr$, the special purpose register file $c_{\mathrm{P}}.spr$, and the byte addressable physical memory $c_{\mathrm{P}}.m$. We denote $d$ consecutive memory cells starting at address $a$ by $m_d[a] = (m[a+d-1], \ldots, m[a])$.

We support up to eight devices identified by natural numbers $i \in \{0, \ldots, 7\}$. These are mapped into the processor's memory at address ranges $DA_i$, which are mutually disjoint. Processor and devices may interact by (i) devices generating interrupts via external event lines $eev[i]$ or (ii) the processor accessing device ports at addresses in $DA_i$ via regular memory instructions.

The interface for the latter operation is defined using two types. The processor requests a device access via the *memory interface input* and receives the device's response via the *memory interface output*; this naming convention is from the point of view of the devices.

Formally, let $DA$ denote the union of all device addresses, let the predicates $lw(c_{\mathrm{P}})$ and $sw(c_{\mathrm{P}})$ indicate load and store word instructions, and let the functions $ea(c_{\mathrm{P}})$ and $RD(c_{\mathrm{P}})$ denote the memory and register operand addresses for such instructions. The memory interface input *mifi* is a quadruple: (i) the read flag $mifi.rd = lw(c_{\mathrm{P}}) \wedge ea(c_{\mathrm{P}}) \in DA$ is set for a load from a device port, (ii) the write flag $mifi.wr = sw(c_{\mathrm{P}}) \wedge ea(c_{\mathrm{P}}) \in DA$ is set for a store to a device port, (iii) the address $mifi.a = ea(c_{\mathrm{P}})$ is set to the effective address, which encodes the accessed device $i$ in bits 12 to 14 and the accessed port in bits 2 to 11 (we support up to 1024 ports of 32 bit width per device), and finally (iv) the data input $mifi.din = c_{\mathrm{P}}.gpr[RD(c_{\mathrm{P}})]$ is set to the store operand.

The memory interface output *mifo* is a 32 bit response to a device port read.

The processor's ISA is formally defined by the output function $\omega_{\mathrm{P}}$ and the transition function $\delta_{\mathrm{P}}$. The former takes a processor state $c_{\mathrm{P}}$ and computes a memory interface input *mifi*, cf. above. The transition function takes a processor state $c_{\mathrm{P}}$, a device output *mifo*, and the devices' external event lines $eev[i]$, which indicate interrupts. It returns the next state of the processor $c'_{\mathrm{P}}$. If all device interrupts are disabled in software and no device is accessed, the external event lines and the memory interface output are ignored. For such steps we use $\delta_{\mathrm{P}}$ in an overloaded, unary variant, which operates on $c_{\mathrm{P}}$ only.

**Devices.** Devices are modeled as finite state transition systems interacting with the processor and with an external environment (e.g., a user or a network). In the

following let $X$ denote a specific kind / type of device. The transition function $\delta_X$ takes a device state, an input from the external environment $eifi_X$, and an input from the processor *mifi*. It returns the next state, an output to the processor *mifo* and an output to the external environment $eifo_X$. Interrupts are signaled by a predicate $\omega_X$ over the device state.

In the models considered here a device either consumes an external or a processor input, never both simultaneously. Hence, in a step either *eifi* or *mifi* is 'empty', denoted with $eifi_\epsilon$ and $mifi_\epsilon$.

**Combined System.** In the overall system we study a model of one processor connected to several devices. A configuration $c_{\mathrm{PD}}$ of the combined system, which we also call global configuration, consists of a processor configuration $c_{\mathrm{PD}}.c_{\mathrm{P}}$ and a mapping $c_{\mathrm{PD}}.c_{\mathrm{D}}$ from device identifiers to device configurations.

The transition function $\delta_{\mathrm{PD}}$ of the combined system has to distinguish whether the processor or a device executes next. Hence, it takes the current global configuration and an oracle input *ev* called event. The event equals $P$ in case of a processor step or is a pair $(i, eifi)$ of device identifier and environment input in case of a device step. The transition function returns the next global configuration and an output *eifo* to the environment.

Let the function *da* indicate whether the processor wants to perform a local step or access a specific device. Formally, $da(c_{\mathrm{P}}) = i$ if $(sw(c_{\mathrm{P}}) \vee lw(c_{\mathrm{P}})) \wedge ea(c_{\mathrm{P}}) \in DA_i$ and $da(c_{\mathrm{P}}) = P$ otherwise.

In the definition of the transition function we distinguish three cases:

1. A *processor-device transition* is taken if it is the processor's turn, $ev = P$, and the current instruction accesses a device $da(c_{\mathrm{PD}}.c_{\mathrm{P}}) = i$ with type $X$. The device takes a step with $\delta_X$, consuming the output $\omega_{\mathrm{P}}(c_{\mathrm{PD}}.c_{\mathrm{P}})$ of the processor and an empty external input $eifi_\epsilon$. For a read, the device returns an output *mifo* to the processor. The processor configuration is updated by applying $\delta_{\mathrm{P}}$ to the current processor configuration, the memory output *mifo*, and the external event bit vector *eev*, defined as $eev[j] = \omega_X(c'_{\mathrm{PD}}.c_{\mathrm{D}}(j))$ for each device $j$ of type $X$.

2. A *local processor transition* is taken if it is the processor's turn, $ev = P$, and the current instruction does not access a device, $da(c_{\mathrm{P}}) = P$. The processor configuration is updated by applying $\delta_{\mathrm{P}}$ to the current processor configuration $c_{\mathrm{PD}}.c_{\mathrm{P}}$, a (dummy) device input, and the external event vector as defined above.

3. An *external device transition* is taken if there is an external input *eifi* for a device $i$ of type $X$, i.e., $ev = (i, eifi)$. Only the configuration of device $i$ is updated by applying $\delta_X$ with the processor input set to $mifi_\epsilon$.

Devices and the processor are executed in an interleaved way. A model run is defined by the start configuration and an *execution sequence* denoted by *seq*. The latter returns for a given step number $t$ the oracle event input $seq(t) = ev$, i.e., it resolves the non-determinism.

The function $\Delta_{\mathrm{PD}}$ is used to model a computation of the overall system. It takes a global start configuration, an execution sequence and a step number $t$ as inputs. It returns a pair, the global configuration reached after applying the

transition function $\delta_{\mathrm{PD}}$ for $t$ times and the sequence of external output generated during this process.

When proving a property of the combined system, not all execution sequences have to be considered. For example, termination of drivers can typically only be shown if processor and devices are scheduled infinitely often, which must be guaranteed by the hardware implementation [20]. Hence, we define valid execution sequences as:

$$Seq_{\mathrm{V}}(seq) \equiv (\forall t\,.\,\exists k > t\,.\,seq(k) = P) \wedge (\forall t, i\,.\,\exists k > t, eifi\,.\,seq(k) = (i, eifi))$$

Correctness of drivers could depend on further device-specific restrictions of the environment. For example, for the hard disk the environment eventually signals termination of a read or write operation. Such assumptions are also formulated in terms of $Seq_{\mathrm{V}}$ and proven for the gate-level implementation.

## 3 Reordering and Abstraction

Obviously, when proving correctness of a concrete driver for a specific device, an interleaved semantics of all devices is cumbersome. Preferably, for the proof we would like to use a simpler programming model first, e.g., a sequential model or a model with just a single device, and then generalize the result. In this section we develop theory for that purpose.

We assume that we have driver code that exclusively controls a certain device $X$ and only that device. For simplicity, in this section we use $X$ both to identify the kind of the controlled device and its number $X \in \{0, \ldots, 7\}$. We also assume that all interrupts are masked in hardware via the special-purpose status register while the driver runs. In our scenario this restriction is not severe. On the one hand, interrupts of device $X$ are assumed already being delivered to our driver. On the other hand, interrupts for other devices should be handled by different drivers in a manner transparent to the driver under verification. Thus, this problem is orthogonal to the one that we focus on here. Techniques for the verification of concurrent (assembly) programs apply in this case (cf. [21]).

A key observation in our scenario is that some steps in the computation of the combined model can be swapped without changing the outcome. This reordering is sound if devices do not influence each other. Swapping steps repeatedly, an execution of the driver in the combined model can be separated into steps involving only the driver and the controlled device followed by steps involving only other devices. Thus, correctness of the driver can be shown in a model with only the processor and the controlled device. Still, this model is concurrent. Two further simplifications may be applicable for parts of the driver execution. First, for phases not involving device access at all properties can be proven relative to just the isolated processor model. Second, for phases in which the device is in a stable state (by which we mean it does not react to external input) properties can be proven relative to a model without (external) device steps.

A similar technique can be applied for higher-level models with devices. For example, in Verisoft the bulk of all software is implemented in a type-safe fragment of C. Most of this code is verified in a Hoare logic verification environment

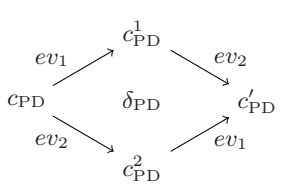



**Fig. 1.** Swapping Two Non-Interfering Steps

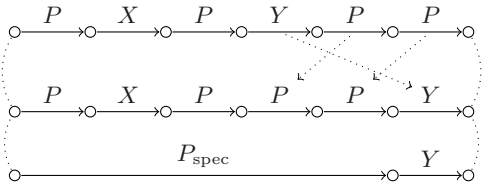



**Fig. 2.** Reordering and Abstraction of an Execution Sequence

for C. Integration of concurrent correctness results into traditional Hoare logic proofs is hardly manageable. It is much more convenient to show the correctness of high-level code against a sequential specification in which the driver calls are executed atomically. Because we use type-safe C we cannot allow direct access to device ports, which do not behave like regular memory. Hence, reordering techniques can be used for a concurrent C semantics with devices, separating C steps from device and driver execution steps.

**A Basic Observation.** A basic observation of our overall model is that device and processor steps not interfering with each other can be swapped. Assuming that interrupts are disabled, we say that two steps do not interfere if at least one is not a processor step and if they do not involve the same device; we call a device involved in a step if it is accessed by the processor or makes a step itself.

Recall that for a processor configuration $c_{\mathrm{P}}$ the function $da$ indicates whether the processor makes a local step, $da(c_{\mathrm{P}}) = P$, or accesses a specific device, $da(c_{\mathrm{P}}) \in \{0, \ldots, 7\}$. For an event $ev$, we let $Da(c_{\mathrm{P}}, ev)$ denote the *set* of components involved in a step. We have $P \in Da(c_{\mathrm{P}}, ev)$ iff $ev = P$ and $X \in Da(c_{\mathrm{P}}, ev)$ iff $ev = (X, \ldots)$ or $da(c_{\mathrm{P}}) = X$.

For a global configuration $c_{\mathrm{PD}}$, two events $ev_1$ and $ev_2$ with $Da(c_{\mathrm{PD}}.c_{\mathrm{P}}, ev_1) \cap Da(c_{\mathrm{PD}}.c_{\mathrm{P}}, ev_2) = \emptyset$ can be executed in arbitrary order, as depicted in Fig. 1:

$$\delta_{\mathrm{PD}}(\delta_{\mathrm{PD}}(c_{\mathrm{PD}}, ev_1)), ev_2) = \delta_{\mathrm{PD}}(\delta_{\mathrm{PD}}(c_{\mathrm{PD}}, ev_2), ev_1) \tag{1}$$

Lifting this observation to execution sequences is simple: we only have to ensure that a valid sequence remains valid after swapping. This is true since we restricted validity only to liveness. However, more complex assumptions over the environment could link input and output behavior of different devices or their relative speed. In this case the invariance of validity has to be proven before applying the reordering theorem that we present in the next section.

We define the following simple criterion to determine whether the basic observation holds over a set of execution sequences. Let $\pi(seq, i)$ denote the *projection* of a sequence $seq$ to a given device $i$, i.e., it returns the subsequence of external steps of device $i$. A predicate over execution sequences is called separable if it can be expressed as a conjunction of predicates over projected execution sequences:

$$separable(Q) \equiv \exists p_0 \ldots p_7 \,.\, \forall seq \,.\, Q(seq) = p_0(\pi(seq, 0)) \wedge \cdots \wedge p_7(\pi(seq, 7))$$

**Lemma 1 (Separable Valid Sequences).** *If the valid sequence predicate is separable, then observation (1) holds over this set.*

**Reordering.** We study when sequential proofs over a given assembly code can be generalized to arbitrary computations. We abbreviate the configurations of a processor-local computation by $c_{\mathrm{P}}^t$ with $c_{\mathrm{P}}^{t+1} = \delta_{\mathrm{P}}(c_{\mathrm{P}}^t)$ and the configurations of a computation of the combined system by $c_{\mathrm{PD}}^{seq,t} = \Delta_{\mathrm{PD}}(c_{\mathrm{PD}}, seq, t)$.

In the simplest case the processor does not access any devices. Since interrupts are masked, processor computations yield the same result in the combined model regardless of device steps.

**Lemma 2 (No Device Access)**

$$(\forall t \leq T \,.\, da(c_{\mathrm{P}}^t) = P) \implies \forall seq \,.\, (c_{\mathrm{P}}^0 = c_{\mathrm{PD}}^{seq,0}.c_{\mathrm{P}} \implies \exists T' \,.\, c_{\mathrm{PD}}^{seq,T'}.c_{\mathrm{P}} = c_{\mathrm{P}}^T)$$

The stated lemma is useful in two situations. First, it allows to reason locally about local steps in the execution of a device driver. Second, it is applicable when reasoning about code of a high-level programming language without direct device access. In this case, code correctness proofs of the code in the high-level language can be performed purely sequentially.

In a more general case the processor accesses only a certain device $X$. We call such parts of the computation *pure*. This is our assumption for drivers; it can usually be shown statically or for local processor computations. Furthermore, we define device configurations to be *stable* if they do not change under external transitions. These predicates are defined formally as follows:

$$\begin{aligned} pure(c_{\mathrm{PD}}^0, seq, T, X) &\equiv \forall t < T \,.\, da(c_{\mathrm{PD}}^{seq,t}.c_{\mathrm{P}}) \in \{P, X\} \\ stable(c_X) &\equiv \forall eifi \,.\, \delta_X(c_X, eifi, mifi_\epsilon) = c_X \end{aligned}$$

The *empty sequence* is the schedule where only the processor takes steps. It is defined as $emp(t) = P$ for all $t$. In pure computations where the accessed device is stable, sequential properties proven over the empty sequence can be generalized to properties over arbitrary sequences.

**Lemma 3 (Pure Sequences and Stable Devices)**

$$\begin{aligned} &pure(c_{\mathrm{PD}}^0, emp, T, X) \wedge \forall t < T \,.\, stable(c_{\mathrm{PD}}^{emp,t}.c_{\mathrm{D}}(X)) \implies \\ &\qquad \forall seq \,.\, \exists T' \,.\, c_{\mathrm{PD}}^{emp,T}.c_{\mathrm{P}} = c_{\mathrm{PD}}^{seq,T'}.c_{\mathrm{P}} \wedge c_{\mathrm{PD}}^{emp,T}.c_{\mathrm{D}}(X) = c_{\mathrm{PD}}^{seq,T'}.c_{\mathrm{D}}(X) \end{aligned}$$

Note that stability of a device is a relatively strong assumption, but sufficient for handling a hard disk driver. For other devices, the notion of stability should be refined, requiring stability only for those parts of the device that are accessed by the processor.

In general, of course, driver correctness can not be shown solely using Lemmas 2 and 3. In the situations not covered by these lemmas, we may still assume

that only the processor or the device $X$ are being scheduled. We call such fragments of an execution sequence *reduced*. Formally, we define

$$\mathit{reduced}(\mathit{seq}, T_1, T_2, X) \equiv \forall T_1 \leq t < T_2 \,.\, \mathit{seq}(t) = P \vee \mathit{seq}(t) = (X, \dots) \ .$$

Complementary, a fragment is free of steps of a device $X$ or the processor iff

$$\mathit{free}(\mathit{seq}, T_1, T_2, X) = \forall T_1 \leq t < T_2 \,.\, \mathit{seq}(t) \neq P \wedge \mathit{seq}(t) \neq (X, \dots) \ .$$

If we have a separable valid sequence, the theorem below states that a pure computation can always be reordered into a reduced part and followed by a free part. The resulting overall state of both computations are equal.

**Theorem 1 (Reordering of Sequences)**

$$\begin{aligned} &\mathit{pure}(c^0_{\mathrm{PD}}, \mathit{seq}, T, X) \implies \\ &\exists \mathit{seq}', T_1, T_2 \,.\, \mathit{reduced}(\mathit{seq}', 0, T_1, X) \wedge \mathit{free}(\mathit{seq}', T_1, T_2, X) \wedge c^{\mathit{seq},T}_{\mathrm{PD}} = c^{\mathit{seq}',T_2}_{\mathrm{PD}} \end{aligned}$$

This theorem can be proven by repeatedly applying the basic observation above. Generalizing this result, the execution of drivers controlling different devices can also be separated, enabling modular verification of device drivers.

In Fig. 2 on page 231 we show an example of a complete execution of a driver for some device $X$. By applying Theorem 1 we soundly reorder the execution of any device $Y$ after the termination of the driver (top line to middle line). The interaction between the driver and the corresponding device can now be specified by a single atomic state update (middle line to bottom line).

**Abstraction.** We examine the scenario that a program implemented in a high-level language wants to access devices by calling assembly drivers for these devices. We assume that the language does not provide direct device access, which is true for Verisoft.

To reason about correctness in this scenario we have to consider the compiled high-level program linked with the assembly driver in the model of the combined system (cf. Sect. 2). The compiler guarantees that the compiled code does not access devices by placing code and data region in regular memory (i.e., not on device ports). Whenever we execute a certain fragment of compiled code we can thus apply Lemma 2 and get to a processor configuration in the combined model that is equal to the processor configuration in the sequential processor model. Therefore, compiler correctness is preserved in a model with devices. In other words, device steps do not interfere with compiler correctness. Moreover, whenever we execute a certain fragment of the compiled code, we can shift the device steps beyond this fragment using Theorem 1 for the special case that there is no device access. In other words, high-level language and compiled code do not interfere with devices (and drivers).

Suppose a driver for device $X$ is called by the high-level program. We apply Theorem 1 on the driver execution, obtaining a reduced fragment for the driver and its controlled device. This division not only eases the driver verification, but

also enables an atomic specification of the driver grouping involved processor and device steps to one semantical call (see the bottom line in Fig. 2). At the end of this fragment the postcondition of the driver ranging only over the device and the processor state can be established. All interleaved and non-interfering device steps are moved beyond the fragment and hence a (partially) sequential programming model is obtained. As a consequence, traditional program logics becomes also applicable to high-level programs including calls to that driver.

We have formally instantiated the described abstraction technique for the Verisoft C compiler in Isabelle/HOL and applied it to the hard disk driver correctness (cf. Sect. 5). Interrupts can be handled similarly if driver execution is transparent to / separated from high-level execution.

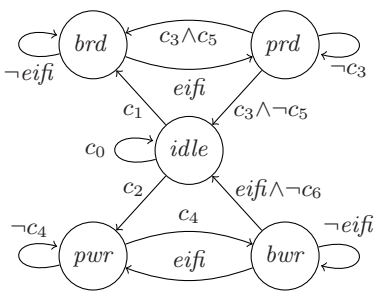
## 4 Hard Disk Model

Our formal hard disk controller model is based on a subset of the ATAPI standard [3]. We restrict ourselves to a few ATAPI commands and assume that only a single disk is hooked up to the controller, the master disk. Hence, we use the terms disk and controller interchangeably. Below, we sketch the definitions of hard disk state and operation, omitting many details due to space restrictions.

Hard disk state is modeled as a record $c_{\mathrm{hd}}$. Hard disk operation is defined via a transition function $\delta_{\mathrm{hd}}$. It takes an external input *eifi*, a memory interface input *mifi*, and a current configuration $c_{\mathrm{hd}}$. It returns an updated configuration $c'_{\mathrm{hd}}$, a memory interface output *mifo*, and an external output *eifo*. In Sect. 2 we have already defined the signature of the memory interface. The external interface for the disk is quite simple. The disk does not produce an external output *eifo*; we will omit it from now on. The external input $\mathit{eifi} \in \{0,1\}$, also known as trigger, indicates when the disk completes certain operations; we abstract from exact timing. For liveness reasons, the trigger must be active infinitely often. This is a (separable) environment restriction we need to make, cf. Sect. 3.

We now describe the hard disk state in more detail. Hard disks are parameterized over the number of sectors $0 < c_{\mathrm{hd}}.S \leq 2^{28}$ they store. Each sector stores 128 words, making up for a maximum content of $c_{\mathrm{hd}}.S \cdot 2^9 \leq 128\,\mathrm{GB}$,

The disk is accessed by issuing commands to it. We only model three commands: *reset* initializes the disk state; *read* and *write* load and store a range of sectors. This range is processed sector by sector. During command execution, the sector range that remains to be processed is identified by a start sector $c_{\mathrm{hd}}.\mathit{lba} \in \mathbb{N}_{<c_{\mathrm{hd}}.S}$ and a sector counter $c_{\mathrm{hd}}.\mathit{scnt} \in \mathbb{N}_{<257}$. Each sector is first being transferred into an internal (volatile) buffer of the disk and then to the processor resp. the disk. The internal buffer is represented as a mapping $c_{\mathrm{hd}}.\mathit{buf} : \mathbb{N}_{<128} \to \mathbb{N}_{<2^{32}}$. The processor accesses this buffer sequentially by reading or writing the *data port* of the disk; each such access increments the buffer pointer $c_{\mathrm{hd}}.\mathit{bp} \in \mathbb{N}_{<128}$ that serves as an index into the internal buffer. Read and write commands can be executed in two modes. In *polling mode*, the processor queries the disk for the completion of a sector transfer; in *interrupt mode*, the disk causes an interrupt for each sector transfer. The interrupt enable




$c_0 = \neg wr(mifi, cmd_{\mathrm{p}})$
$\hat{c} = c_{\mathrm{hd}}.scnt \neq 0 \wedge (c_{\mathrm{hd}}.S \geq c_{\mathrm{hd}}.lba + c_{\mathrm{hd}}.scnt)$
$c_1 = cmd(mifi, rd_{\mathrm{c}}) \wedge \hat{c}$
$c_2 = cmd(mifi, wr_{\mathrm{c}}) \wedge \hat{c}$
$\tilde{c} = (c_{\mathrm{hd}}.bp = 127)$
$c_3 = rd(mifi, data_{\mathrm{p}}) \wedge \tilde{c}$
$c_4 = wr(mifi, data_{\mathrm{p}}) \wedge \tilde{c}$
$c_5 = c_{\mathrm{hd}}.scnt > 0$
$c_6 = c_{\mathrm{hd}}.scnt > 1$

**Fig. 3.** Regular State Transitions

flag $c_{\mathrm{hd}}.ien \in \{0,1\}$ indicates the current mode. It is zero for polling mode. In interrupt mode, the pending interrupt flag $c_{\mathrm{hd}}.pint \in \{0,1\}$ indicates whether a hard disk interrupt waits to be serviced by the processor. The interrupt predicate thus simply equals the pending interrupt flag, i.e., $\omega_{\mathrm{hd}}(c_{\mathrm{hd}}) = c_{\mathrm{hd}}.pint$.

All of the processor-device interaction we sketched above (other than the disk generating interrupts) takes place by the processor accessing the hard disk's eight ports. The sector count port $scnt_{\mathrm{p}}$ defines the number of sectors to process. The sector number, cylinder low, cylinder high, and drive head ports ($snumb_{\mathrm{p}}$, $cyll_{\mathrm{p}}$, $cylh_{\mathrm{p}}$, and $drvhd_{\mathrm{p}}$) define the 28-bit start sector represented as a quadruple of $3 \cdot 8 + 4$ bits. The device control port $devcntrl_{\mathrm{p}}$ selects polling or interrupt mode. The command port $cmd_{\mathrm{p}}$ is used to issue commands when written and check polling status when read. Finally, the internal buffer is accessed via the data port $data_{\mathrm{p}}$. The data port has width 32 bits, all other ports have width 8 bits.

Hard disk operation is governed by a small control automaton with state $c_{\mathrm{hd}}.cs \in \{idle, brd, bwr, prd, pwr, err\}$. In *idle* state, the disk awaits new commands. Read commands will loop over states *brd* and *prd*. In the former the disk fills its buffer, which is then read out by the processor. Likewise, write commands loop over states *pwr* and *bwr*. In the former the processor fills the disk's buffer, which is then written by the disk. The state *err* is an error state.

We call non-reset, non-error state transitions of the control automaton *regular*. These transitions are shown in Fig. 3. Edges are labeled with transition constraints, where $rd(mifi, x) = mifi.rd \wedge (mifi.a = x)$ and $wr(mifi, x) = mifi.wr \wedge (mifi.a = x)$ indicate read and write accesses to port $x$ and $cmd(mifi, c) = wr(mifi, cmd_{\mathrm{p}}) \wedge (mifi.din = c)$ indicates the issuing of command $c$.

Let us outline the major distinctions in the transition function (in addition to the control state transitions). We assume $\delta_{\mathrm{hd}}(eifi, mifi, c_{\mathrm{hd}}) = (c'_{\mathrm{hd}}, mifo)$.

Issuing the reset command, $cmd(mifi, rst_{\mathrm{c}})$, has top priority: the disk control enters the idle state, $c'_{\mathrm{hd}}.cs = idle$, and the buffer pointer, the interrupt enable flag, and the pending interrupt flag are set to zero. The other components do not change; the value of *mifo* is irrelevant (for any processor write, in fact).

If no reset command is issued an error transition may be taken, $c'_{\mathrm{hd}}.cs = err$. Absence of error transitions should guarantee that the processor handles the device correctly. In addition to obvious error transitions, e.g., write to a read-only

port, we also use error transitions for modeling shortcomings, e.g., an attempt to issue an unmodeled command. We do not define the error conditions here.

Finally, we distinguish two types of regular transitions, which do not occur simultaneously. Regular, processor-initiated transitions are used to set up command parameters, start commands, access the internal buffer, or query the disk status. The definition of the transition function for these cases is easy. As side effects, the buffer pointer is incremented for a data port access and the pending interrupt flag is cleared for a disk status check.

Regular, external transitions are initiated by the external trigger flag *eifi*, which only has an effect in states *bwr* or *brd*, waiting for a sector to be written to or read from the disk. Such a transition has side effects. The start sector is incremented and the sector count is decremented, $c'_{\mathrm{hd}}.lba = c_{\mathrm{hd}}.lba + 1$ and $c'_{\mathrm{hd}}.scnt = c_{\mathrm{hd}}.scnt - 1$. For buffer write, the buffer is copied to the disk, $c'_{\mathrm{hd}}.sm_{128}(c_{\mathrm{hd}}.lba \circ 0^9) = c_{\mathrm{hd}}.buf$. For buffer read, $c'_{\mathrm{hd}}.buf = c_{\mathrm{hd}}.sm_{128}(c_{\mathrm{hd}}.lba \circ 0^9)$. The pending interrupt flag is turned on in interrupt mode, $c'_{\mathrm{hd}}.pint = c_{\mathrm{hd}}.ien$.

## 5 Hard Disk Driver and Correctness

We present a simple assembly device driver for which we have formally proven correctness in Isabelle/HOL [2] based on the combined system from Sect. 2 and the theory developed in Sect. 3. The driver writes a 4 K page (8 sectors) from the processor's memory, starting at address $a$, to the disk, starting at sector $b$. Its code is shown in Fig. 4. We use MIPS-like syntax; GPRs are written as `r`$k$, memory operands as $imm$(*RS1*). Arrows indicate jump targets; according to the delayed PC, instructions in delay slots are always executed.

The code can be structured into five main parts. In part 0, we set up all parameters for the disk write command in the registers. For example, the start sector index $b$ is decomposed into the sector number, cylinder low, cylinder high, and drive index. In part 1, command parameters are written to the disk's configuration ports. Interrupt mode is disabled in step (1.2) and the write command is issued in step (1.8). Each iteration of the outer loop in steps (2.1) to (5.3) copies one sector from the main memory of the processor to the sector memory of the disk. One sector consists of 128 words. The first inner loop copies word

```
andi r16,r15,#255    (0.1)
srli r17,r15,#8      (0.2)
andi r17,r17,#255    (0.3)
srli r18,r15,#16     (0.4)
andi r18,r18,#255    (0.5)
srli r19,r15,#24     (0.6)
andi r19,r19,#15     (0.7)
addi r19,r19,#224    (0.8)
addi r3,r0,#wr_c     (0.9)
addi r4,r0,#2       (0.10)
addi r12,r0,#8      (0.11)

xori r1,r0,#Da(X_hd) (1.1)
sw devctrl_p(r1),r4  (1.2)
sw scnt_p(r1),r12    (1.3)
sw snumb_p(r1),r16   (1.4)
sw cyll_p(r1),r17    (1.5)
sw cylh_p(r1),r18    (1.6)
sw drvhd_p(r1),r19   (1.7)
sw cmd_p(r1),r3      (1.8)

→addi r10,r0,#128    (2.1)
 →lw r3,0(r2)        (3.1)
  sw data_p(r1),r3   (3.2)
  subi r10,r10,#1    (3.3)
  bnez r10,#-16      (3.4)
 addi r2,r2,#4       (3.5)
 →lw r3,stat_p(r1)   (4.1)
  sgei r14,r3,#128   (4.2)
  bnez r14,#-12      (4.3)
 nop                 (4.4)
  subi r12,r12,#1    (5.1)
  bnez r12,#-48      (5.2)
 nop                 (5.3)
```

**Fig. 4.** Device Driver

after word from the processor's memory to the internal disk buffer. When the buffer is full (i.e., after one complete sector) the driver enters the second inner loop, which polls until the hard disk has written its buffer.

In the following we will abbreviate component access to the hard disk and the processor with $c_{\mathrm{hd}}^{seq,t} = c_{\mathrm{PD}}^{seq,t}.c_{\mathrm{D}}(X_{\mathrm{hd}})$ and $c_{\mathrm{P}}^{seq,t} = c_{\mathrm{PD}}^{seq,t}.c_{\mathrm{P}}$ respectively.

The correctness of the driver is proven for all valid execution sequences. It states that when driver execution terminates, the page located in the processor memory is finally copied to the sector memory of the disk. Furthermore the physical memory of the processor stays unchanged.

**Theorem 2 (Hard Disk Driver Correctness).** *Assume memory address and start sector are in the first two registers,* $c_{\mathrm{P}}^{0}.gpr[1] = a$ *and* $c_{\mathrm{P}}^{0}.gpr[2] = b$*. The hard disk is assumed to be idle,* $c_{\mathrm{hd}}^{0}.cs = idle$*. Then it holds:*

$$\forall seq\,.\,Seq_{\mathrm{V}}(seq) \implies \exists t\,.\,c_{\mathrm{hd}}^{seq,t}.sm_{8\cdot 128}[b\cdot 128] = c_{\mathrm{P}}^{0}.m_{4\cdot 8\cdot 128}[a] \wedge c_{\mathrm{P}}^{seq,t}.m = c_{\mathrm{P}}^{0}.m$$

In the following we apply the theory developed in Sect. 3 for hard disk driver correctness. Since the first part of the code does not access any device at all, with Lemma 2 we can prove its correctness resorting only to ISA semantics. Next we establish that during part 1 only the hard disk is accessed and it remains idle, i.e., the *pure* and *stable* conditions instantiated for the disk are fulfilled. Using Lemma 3 we can now prove correctness of part 1, by only analyzing the global computations without external device steps. Note, that it suffices to validate stability and purity only for the empty sequence.

The hard disk remains stable in buffer write state *bwr*, and purity still holds for the first inner loop. Hence, again by Lemma 3, it suffices to establish the invariant only for the empty sequence.

Things get more involved in the second inner loop, due to absence of stability: at an arbitrary time the hard disk may transfer the buffer content to the persistent sector memory. Hence, termination of the polling loop depends on the external environment, which finally indicates the operation to be completed. A full-blown interleaved analysis is still not necessary. Applying Theorem 1 device steps other than the hard disk can be ignored, and we establish the proof only over sequences reduced to disk steps. However, we first have to discharge the separability condition for the trigger restriction imposed by the disk on the environment. This follows from a simple application of Lemma 1.

Summarizing, except for the polling loop in part 4, correctness could be shown completely sequentially.

The driver presented here is used in Verisoft kernel code to perform page swap-out [22]. This is achieved by embedding it into a C function declared as `void write_to_disk(int a, int b)`. Two more instructions are needed to load the parameters from the program stack. These instructions do not access devices.

With the abstraction technique from Sect. 3, we formally established correctness of the driver call against an atomic specification. Using it the correctness of client code can be shown in Hoare logic.

## 6 Future Work

There are several directions for future work. The disk driver presented in Sect. 5 is used in Verisoft microkernel code for page swap-out [22]. The driver for page swap-in remains to be verified. Also, the driver given is only a polling one. For the file system implementation in Verisoft's simple operating system interrupts are also used. This code is being verified.

For the verification of code for device other than hard disk, it might be interesting to refine the concept of stability, which was introduced in Sect. 3. Typically, a communicating device (e.g., network interface card) is never stable on the complete configuration because it always asynchronously transfers data. However, by defining stability only for parts of the state, it can still usually be preserved. For example, communication is often channeled through buffers for transmission and reception with processor and environment accessing these buffers at different ends. Concurrency can be reduced in such a scenario.

## 7 Conclusion

We have presented the formal functional correctness proof of an assembly disk driver against a formal architecture model integrating devices, which is concurrent. The proof could be decomposed into two parts. First, we have proven the driver correct in a model with just the hard disk present. By abstracting from the other devices, the set of model runs has been reduced significantly. Second, we have generalized this result to computations in the full model by proving a general reordering theorem. This theorem is applicable if devices do not interfere with each other and a device is controlled exclusively by a single driver. The same reordering can also be applied to the high-level language model with devices, allowing to separate high-level computational from device steps.

Not classical problems such as finding correct invariants turned out to be hard during the verification process. Most notably, an appropriate program logic for assembly and better support for arithmetics in the prover were sorely missed. With the help of reordering, interleaved reasoning was only required for two lines of code, amounting to one third of the overall verification effort.

Combining our result with hardware and compiler correctness [20, 23], allows to transfer properties of a high-level program calling our driver down to the gate-level implementation of the complete system.

## References

1. Hillebrand, M.A., Paul, W.: On the architecture of system verification environments. In: Yorav, K. (ed.) HVC 2007. LNCS, vol. 4899, pp. 153–168. Springer, Heidelberg (2008)
2. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
3. American National Standards Institute: ANSI NCITS 340-2000: AT Attachment-5 with Packet Interface (2000)

4. Hillebrand, M., In der Rieden, T., Paul, W.: Dealing with I/O devices in the context of pervasive system verification. In: ICCD 2005, pp. 309–316. IEEE Computer Society, Los Alamitos (2005)
5. Alkassar, E., et al.: Formal device and programming model for a serial interface. In: Beckert, B. (ed.) VERIFY 2007, pp. 4–20 (2007)
6. Berry, G., Kishinevsky, M., Singh, S.: System level design and verification using a synchronous language. In: ICCAD, pp. 433–440. IEEE Computer Society / ACM, Los Alamitos (2003)
7. Rashinkar, P., Paterson, P., Singh, L.: System-on-a-Chip Verification: Methodology and Techniques. Kluwer Academic Publishers, Norwell (2001)
8. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 103–122. Springer, Heidelberg (2001)
9. Microsoft Corporation: SDV: Static driver verifier (2004), `http://www.microsoft.com/whdc/devtools/tools/sdv.mspx`
10. Hallgren, T., et al.: A principled approach to operating system construction in Haskell. In: Danvy, O., Pierce, B.C. (eds.) ICFP. ACM, New York (2005)
11. Heiser, G., et al.: Towards trustworthy computing systems: Taking microkernels to the next level. SIGOPS Oper. Syst. Rev. 41(4), 3–11 (2007)
12. The FLINT Project, `http://flint.cs.yale.edu/flint/`
13. Holzmann, G.J.: New challenges in model checking. In: Symposium on 25 years of Model Checking, Seattle, USA, LNCS, vol. 4925. Springer, Heidelberg (August 2006)
14. Freitas, L., Fu, Z., Woodcock, J.: POSIX file store in Z/Eves: An experiment in the verified software repository. In: ICECCS, pp. 3–14. IEEE Computer Society, Los Alamitos (2007)
15. Butterfield, A., Woodcock, J.: Formalising Flash memory: First steps. In: ICECCS, pp. 251–260. IEEE Computer Society, Los Alamitos (2007)
16. Lipton, R.J.: Reduction: A method of proving properties of parallel programs. Commun. ACM 18(12), 717–721 (1975)
17. Cohen, E., Lamport, L.: Reduction in TLA. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 317–331. Springer, Heidelberg (1998)
18. Cohen, E.: Separation and reduction. In: Backhouse, R., Oliveira, J.N. (eds.) MPC 2000. LNCS, vol. 1837, pp. 45–59. Springer, Heidelberg (2000)
19. Stoller, S.D., Cohen, E.: Optimistic synchronization-based state-space reduction. Form. Methods Syst. Des. 28(3), 263–289 (2006)
20. Tverdyshev, S., Shadrin, A.: Formal verification of gate-level computer systems. In: Rozier, K.Y. (ed.) LFM 2008. NASA STI, NASA, pp. 56–58 (2008)
21. Yu, D., Shao, Z.: Verification of safety properties for concurrent assembly code. In: ICFP 2004 (September 2004)
22. Alkassar, E., Schirmer, N., Starostin, A.: Formal pervasive verification of a paging mechanism. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 109–123. Springer, Heidelberg (2008)
23. Leinenbach, D., Petrova, E.: Pervasive compiler verification – From verified programs to verified systems. In: SSV 2008. ENTCS, vol. 217C, pp. 23–40. Elsevier Science B.V., Amsterdam (2008)