Formal Device and Programming Model for a Serial Interface

Eyad Alkassar[1,⋆], Mark Hillebrand[2,⋆], Steffen Knapp[1,⋆], Rostislav Rusev[1,⋆], and Sergey Tverdyshev[1,⋆]

[1] Saarland University, Dept. of Computer Science, 66123 Saarbrücken, Germany
{eyad, sknapp, rusev, deru}@wjpserver.cs.uni-sb.de
[2] German Research Center for Artificial Intelligence (DFKI GmbH), Stuhlsatzenhausweg 3, 66123 Saarbrücken, Germany
mah@dfki.de

Abstract. The verification of device drivers is essential for the pervasive verification of an operating system. To show the correctness of device drivers, devices have to be formally modeled. In this paper we present the formal model of the serial interface controller UART 16550A. By combining the device model with a formal model of a processor instruction set architecture we obtain an assembler-level programming model for a serial interface. As a programming and verification example we present a simple UART driver implemented in assembler and prove its correctness. All models presented in this paper have been formally specified in the Isabelle/HOL theorem prover.

1 Introduction

The Verisoft project [1] aims at the pervasive modeling, implementation, and verification of complete computer systems, from gate-level hardware to applications running on top of an operating system. The considered systems employ various devices, e.g., a hard disk controller for persistent storage, a time-triggered bus controller for communication in a distributed system, and a serial interface for user interaction via a terminal. The drivers controlling these devices are part of the operating system and proving their correctness is critical to proving the correctness of the system as a whole.

Here we consider a system which the user may control with a terminal connected via a serial interface. To prove the functional correctness of the serial interface device driver it is not sufficient to argue only about the driver code; the serial interface itself and its interaction with the processor have to be formally modeled, too. In this paper we present for the first time a formal model of a serial interface and its programming model at the assembler language level. Furthermore, as an informal example, we present a serial interface driver and sketch its correctness proof with respect to our models.

The remainder of this paper is structured as follows. In Sect. 2 we discuss previous and related work. In Sect. 3 we sketch the instruction set architecture of the VAMP

⋆ Work of the first author was supported by the German Research Foundation (DFG) within the program 'Performance Guarantees for Computer Systems'. Work of the third author was supported by the International Max Planck Research School for Computer Science (IMPRS). Work of all but the fourth author was supported by the German Federal Ministry of Education and Research (BMBF) in the Verisoft project under grant 01 IS C38.

processor [2, 3] and show how memory-mapped devices can be integrated into this architecture. In Sect. 4 we present the formal model of a UART 16550A controller and formalized environmental and software conditions. To informally demonstrate the utility of the framework, in Sect. 5 we present a simple driver written in assembler, which writes several words to the serial interface. We sketch its correctness proof.

2 Previous and Related Work

For the pervasive verification of computer systems, as done in the Verisoft project, devices must be modeled at different system layers. Some results of Verisoft's completed or ongoing work in the context of devices and their drivers have already been published. One subproject of Verisoft deals with the verification of a FlexRay-like shared serial bus interface to be used in distributed automotive systems. To verify such a system we need to argue formally about low-level clock synchronization [4], gate-level implementation, and driver real-time properties of the FlexRay interface. All these arguments will finally be combined into one formal pervasive correctness proof. A paper-and-pencil style description of this ongoing effort can be found in Knapp and Paul [5].

In another Verisoft subproject the formal pervasive verification of a general-purpose computer system is attempted. In this context, Hillebrand *et al.* [6] presented paper-and-pencil formalizations of a system with devices for the gate and the assembler level. For a hard disk as a specific device, correctness arguments justifying these models and a correctness proof for a disk driver were given on paper. Here we *formalize* large portions of [6] for a communication device.

So far almost all other device related verification approaches have either aimed at the correctness of a gate-level implementation or at showing 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, [7–10] deal with UARTs in that manner. Berry *et al.* [8] specified a UART model in a synchronous language and proved a set of safety properties regarding FIFO queues. From that model a hardware description can be generated (either in RTL or software simulation) and run on a FPGA.

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 the system to crash. For example, the SLAM project [11] provides tools for the validation of safety properties of drivers written in C. Lately, the success of the SLAM project led to the deployment of the Static Driver Verifier (SDV) as part of the Windows Driver Foundation [12]. SDV automatically checks a set of 65 safety rules concerning Windows Driver API for given C driver programs. Hallgren *et al.* [13] modeled device interfaces for a simple operating system implemented in Haskell. Three basic memory-mapped I/O primitives were specified: read, write, and a test for valid region.

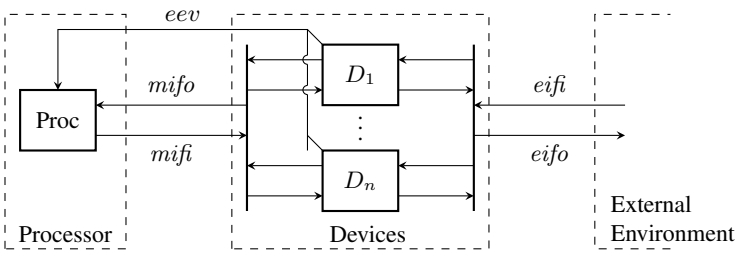

Fig. 1. Overview: A System with Processor and Devices

However, the only correctness property being stated is the disjointness of the device address spaces.

In contrary to all mentioned approaches, we aim at the formalization and *functional* verification a (UART) driver interacting with a device. Thus, it is not sufficient to argue about the device or the programming model alone. Similar in scope is the 'mini challenge' proposed by Holzmann [14], which deals with the formal verification of a file system for a Flash device. Apparently, formalizing the device and its interaction with the driver is also part of the challenge. However, no details have yet been published.

3 Processor and Devices

In this section we define the instruction set architecture (ISA) of a processor with memory-mapped devices as depicted in Fig. 1.

Compared to regular ISA definitions we have the following differences: in addition to the processor state, the state space of the combined architecture also includes devices states. Processor and devices may interact (i) by the processor issuing memory operations to special memory regions (device addresses) and (ii) by the device causing interrupts. Additionally, devices can make computational steps on their own when interacting with an external environment (e.g., a network). Therefore we model the computation of ISA with devices as an interleaved computation.

Note that at the hardware level processor and devices run in parallel and not interleaved. This requires some non-trivial extensions of the formal hardware correctness proof, which we will report on elsewhere.

3.1 Processor

A processor configuration c_{P} is a tuple consisting of (i) two program counters $c_{\mathrm{P}}.pc$ and $c_{\mathrm{P}}.dpc$ implementing delayed branching, (ii) general purpose, floating point, and special purpose register files $c_{\mathrm{P}}.gpr$, $c_{\mathrm{P}}.fpr$, $c_{\mathrm{P}}.spr$, and (iii) a byte addressable memory $c_{\mathrm{P}}.m$.

Devices are mapped into the processor memory. Thus by simple read and write operations the processor can access them. In addition devices can signal an interrupt to the processor via an external event signal (cf. Fig. 1).

Let DA denote the set of memory addresses mapping to devices, which are disjoint from regular physical memory addresses. The processor indicates an access to an address in DA via the memory interface input $mifi$ and receives the device's response on the memory interface output $mifo$; this naming convention is from the point of view of the devices.

Formally, let the predicates $lw(c_{\mathrm{P}})$ and $sw(c_{\mathrm{P}})$ indicate load and store word instructions and let $ea(c_{\mathrm{P}})$ and $RD(c_{\mathrm{P}})$ denote the address and affected processor register for such operations (see Müller and Paul [15] for full definitions).

The memory interface input has the following four components: (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 address, (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 address, (iii) the address $mifi.a = ea(c_{\mathrm{P}})$ is set to the effective address, with $ea[14:12]$ specifying the accessed device and $ea[11:2]$ specifying the accessed device port (we support up to eight devices with up to 1024 ports of width 32 bits), 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 $mifo \in \{0,1\}^{32}$ contains the device's response for a load operation on a device.

The processor model is defined by the output function ω_{P} and the next state function δ_{P}. The former takes a processor state c_{P} and computes a memory interface input $mifi$ to the device as defined above. The latter takes a processor state c_{P}, a device output $mifo$, and an external event (interrupt) vector eev (where $eev[i]$ is set iff device D_i indicates an interrupt). It returns the next state of the processor c'_{P}.

3.2 Devices

The configurations of all devices are combined in a mapping c_{D} from an index D_i to the corresponding device configuration.

Our device model is sequential in the sense that a device may progress either due to a processor access or an input from the external environment. To distinguish both cases we extend the set of device indices by the processor index P and denote this set by PD.

The device transition function δ_{D} specifies the interaction of the devices with the processor and the external environment. It takes a processor-device index $idx \in PD$, an input from the external environment $eifi$, an input from the processor $mifi$, and a combined device configuration c_{D}. It returns a new device configuration c'_{D}, an output to the processor $mifo$, and an external output $eifo$.

Depending on the input index idx and the memory input $mifi$, the transition function δ_{D} is defined according to the following three cases:

- If $idx \neq P$, a step of the device idx is triggered by the external input $eifi$. In this case δ_{D} ignores the given $mifi$.
- If $idx = P \wedge (mifi.wr \vee mifi.rd)$, a device step is triggered by a processor device access. In this case δ_{D} ignores the given $eifi$ and produces an arbitrary $eifo$. The device being accessed as well as the access-type is specified by the given $mifi$.
- Otherwise the processor does not access any device. In this case, δ_{D} does nothing.

The device output function ω_{D} computes the external event vector eev for the processor based on the current device configurations.

3.3 Combined System

By combining the processor and device models we obtain a model for the overall system with devices as depicted in Fig. 1. This model allows interaction with an external environment via $eifi$ and $eifo$ whereas the communication between processor and devices is not visible from the outside anymore.

A configuration c_{PD} of the combined model consists of a processor configuration $c_{\mathrm{PD}}.c_{\mathrm{P}}$ and device configurations $c_{\mathrm{PD}}.c_{\mathrm{D}}$.

Similarly to the previous models, we define a transition function δ_{PD} and an output function ω_{PD}. Both functions take the same three inputs: a processor-device index idx, a configuration c_{PD}, and an external input $eifi$.

We introduce some more notation for the transition and the output function. Let $mifi = \omega_{\mathrm{P}}(c_{\mathrm{PD}}.c_{\mathrm{P}})$ be the memory interface input from the processor to the devices. Let $(c'_{\mathrm{PD}}.c_{\mathrm{D}}, mifo, eifo) = \delta_{\mathrm{D}}(idx, c_{\mathrm{PD}}.c_{\mathrm{D}}, eifi, mifi)$ denote the updated device configuration, the memory output to the processor, and the external output. Let $eev = \omega_{\mathrm{D}}(c'_{\mathrm{PD}}.c_{\mathrm{D}})$ denote the external event vector, which is computed based on the updated device configuration. Finally, if $idx = P$ then $c'_{\mathrm{PD}}.c_{\mathrm{P}}$ denotes the updated processor configuration, i.e., $c'_{\mathrm{PD}}.c_{\mathrm{P}} = \delta_{\mathrm{P}}(c_{\mathrm{PD}}.c_{\mathrm{P}}, eev, mifo)$. Otherwise $c'_{\mathrm{PD}}.c_{\mathrm{P}}$ denotes the unchanged processor configuration, i.e., $c'_{\mathrm{PD}}.c_{\mathrm{P}} = c_{\mathrm{PD}}.c_{\mathrm{P}}$.

The transition function δ_{PD} returns the new configuration, $\delta_{\mathrm{PD}}(idx, eifi, c_{\mathrm{PD}}) = c'_{\mathrm{PD}}$. The output function ω_{PD} simply returns the output to the external environment, $\omega_{\mathrm{PD}}(idx, c_{\mathrm{PD}}, eifi) = eifo$.

3.4 Model Run

A model run is computed by the function run_{PD}, which executes a number of steps in the combined model. It takes as inputs an initial configuration c^0_{PD}, a number of steps i, an external input sequence $eifiseq \in \mathbb{N} \to eifi$, and a computational sequence $seq_{\mathrm{PD}} \in \mathbb{N} \to PD$, which designates the interleaving of the processor and device steps. It returns an updated configuration c'_{PD} and an external output sequence $eifoseq \in \mathbb{N} \to eifo$.

The run function run_{PD} is defined by recursive application of δ_{PD}. For the base case, i.e., $i = 0$, we set $run_{\mathrm{PD}}(0, seq_{\mathrm{PD}}, eifiseq, c^0_{\mathrm{PD}}) = (c^0_{\mathrm{PD}}, \langle\rangle)$.

For $i+1$, let $(c_{\mathrm{PD}}, \mathit{eifoseq}) = \mathit{run}_{\mathrm{PD}}(i, \mathit{seq}_{\mathrm{PD}}, \mathit{eifiseq}, c^0_{\mathrm{PD}})$ denote configurations and outputs after executing i steps. To execute the $(i+1)$-th step, we apply the transition and output function of the combined model one more time. Let $c'_{\mathrm{PD}} = \delta_{\mathrm{PD}}(\mathit{seq}_{\mathrm{PD}}(i), \mathit{eifiseq}(i), c_{\mathrm{PD}})$ and $\mathit{eifo} = \omega_{\mathrm{PD}}(\mathit{seq}_{\mathrm{PD}}(i), \mathit{eifiseq}(i), c_{\mathrm{PD}})$. We define $\mathit{run}_{\mathrm{PD}}(i+1, \mathit{seq}_{\mathrm{PD}}, \mathit{eifiseq}, c^0_{\mathrm{PD}}) = (c'_{\mathrm{PD}}, \mathit{eifoseq} \circ (\mathit{eifo}, \mathit{seq}_{\mathrm{PD}}(i)))$.

4 Serial Interface (UART 16550A)

The *universal asynchronous receiver / transmitter* (UART) is a chip for communication over a serial interface. In the following we will simply speak of a serial interface.

A serial interface provides a computing device with the capability to send data via a few copper wires to another serial interface. This facility is used for non-network communication, e.g., with terminals, modems, and other computers.

In this section we describe the driver programmer's model of the serial interface chip UART 16550A [16]. Briefly summarized, the processor can send or receive data byte-wise. In case of a send this byte is stored in a FIFO queue, called the transmitter buffer, and later on sent to the external environment. A receiver buffer stores all incoming bytes from the environment. Reads from the programmer are served in FIFO manner, too. The UART provides the programmer with two methods to access status information for both buffers: either by interrupts or by polling special ports.

The transmitter and receiver queue are bounded in size (16 bytes); thus they may overrun. The receiver queue may overrun if the speed of incoming data from the environment exceeds the speed at which the processor can handle it. The transmitter queue may overrun if the processor writes data into the transmitter buffer faster than the serial interface can send out the data to the environment. Another error related to the queue size occurs when an empty receiver queue is read.

Our model handles the three error cases as follows: (i) overruns in the receiver queue are treated according to the UART specification, i.e., new incoming bytes are dropped, (ii) writing to a full transmitter queue and reading the empty receiver buffer are not allowed in our model, they are excluded through explicit software conditions.

Discharging these software conditions for a particular driver is tricky, and obviously it would be desirable to find easier accessible programming rules, e.g., write only if the transmitter buffer signals that it is empty. Proving that a concrete system will never let the receiver queue overrun, is even harder. A programmer would typically rely on run-time estimations, which ensure that the environment does not send data faster than the driver code can handle (when running on a certain machine). However, our model does not provide any real-time bounds on computations, and hence proving correctness of a concrete system would either require further assumptions over the environment (e.g., as part of a communication protocol between two serial interfaces) or correctness criteria which tolerate overruns.

For brevity, in this paper we do not go into detail regarding the *memory control register* and the *memory status register*. These registers are used to address additional

wires connected to some external modem and to configure hardware flow control. The remainder of this section is structured as follows. In Sect. 4.1 the configuration and the ports of the serial interface is detailed. The transition function δ_{u} of the serial interface is split into two logical parts: a processor- and an environment sided transition function. The first part is specified in Sect. 4.2, the second in Sect. 4.3. Finally in Sect. 4.4 all required software and environment restrictions are stated and different ways of discharging them are discussed.

4.1 Configuration

In the definition of the UART we use FIFO queues C_T of maximum size 16 for types T. For example, we use queues of type $C_{\mathbb{B}^8}$ to send and receive data.

We model these queues by cyclic buffers with head and tail pointers hd and tl. The buffer content ct maps indices to elements of type T. The number of queue entries is denoted as len.

Queues with $len = 0$ and $len = 16$ are called empty and full. The head element of a queue is accessed by $head(b) = b.ct(b.hd)$. Queues are manipulated by the operations $push$ and pop. The function $push$ adds a new byte to the queue at the tail pointer. It is only defined for non-full queues. We set $push(bdin, b) = b'$ where $b'.ct[b.tl] = bdin$, $b'.tl = (b.tl + 1) \bmod 16$ and $b'.len = b.len + 1$. The function pop deletes the element pointed to by the head pointer. It is only defined for non-empty queues. We set $pop(b) = b'$ where $b'.hd = (b.hd + 1) \bmod 16$ and $b'.len = b.len - 1$. A configuration of a serial interface c_{u} is a record with the following components:

1. The *transmitter holding buffer* $thb \in C_{\mathbb{B}^8}$ is a FIFO byte queue of size 16. Input bytes from the external environment are stored in chronological order. The transmitter holding buffer can be read byte-wise by the programmer.

2. The *receiver buffer* $rb \in C_{\mathbb{B}^8}$ is a FIFO byte queue of size 16. It can be written byte-wise by the programmer.

3. Interrupt driven mode configuration. The serial interface generates four types of interrupts (mapped to a single interrupt line). For each type two kinds of flags are maintained in the configuration: one indicating whether the interrupt type is enabled or disabled and the other indicating whether a corresponding interrupt is still pending.

- The *received data available interrupt* is generated, when the number of bytes in the receive buffer exceeds its interrupt trigger level. This level is computed as $itl(x) = 7x[1] + 3x[0] \cdot (x[1] + 1) + 1 \in \{1, 4, 8, 14\}$ for $x = c_{\mathrm{u}}.rbitl \in \mathbb{B}^2$. The component $c_{\mathrm{u}}.erdai \in \mathbb{B}$ indicates if the interrupt is enabled or not, while $c_{\mathrm{u}}.rdai \in \mathbb{B}$ indicates if the interrupt is currently pending.
- The *transmitter holding buffer empty interrupt* is generated if the transmitter buffer is empty. The component $c_{\mathrm{u}}.ethrei \in \mathbb{B}$ indicates if the interrupt is enabled or not, while $c_{\mathrm{u}}.threi \in \mathbb{B}$ indicates if it is currently pending.
- The *receiver line status interrupt* is generated if certain transmission errors occur. These are *overrun*, *parity*, *framing*, and *breaking* errors. The components $c_{\mathrm{u}}.oe$

specifies if an overrun in one of the two queues occurred. The *parity*, *framing*, and *breaking* errors are linked to particular bytes in the receiver queue. Their occurrence is saved in the 3-bit FIFO queue $c_{\mathrm{u}}.trerr \in C_{\mathbb{B}^3}$. For example, 110 encodes a parity and a framing error in the corresponding byte of the receiver queue.

– The *timeout interrupt* is generated if for a certain period of time no data was received or read from the receiver queue. The UART sets this timeout to the time needed to receive four bytes. This interrupt type can be used by the programmer to ensure that no data is forgotten in the receive queue after the input stream ended. The component $c_{\mathrm{u}}.toi \in \mathbb{B}$ indicates if the interrupt is currently pending. This interrupt is enabled iff the received data available interrupt is.

4. Polling mode configuration. The serial interface can also be operated in polling mode, in which the driver can check the status of the buffers by reading special ports. These ports map to three boolean configuration components: the data ready flag $c_{\mathrm{u}}.dr \in \mathbb{B}$, the empty transmitter holding buffer flag $c_{\mathrm{u}}.ethb$, and the empty data holding registers flag $c_{\mathrm{u}}.edhr$.
It is possible to mix interrupt and polling modes, e.g., the programmer could be informed about incoming data by an interrupt and then read the receiver buffer as long as it is non-empty.

5. Word length configuration. The following components can be set by the programmer, but do not affect the modeled behavior of our device. Nevertheless we need to model them: when connecting two serial interfaces the word length must be configured equally on both sides.
The serial interface uses a timer with a 115.2 kHz frequency; the *baud rate* is computed as $115200/c_{\mathrm{u}}.div$. Due to port overloading the programmer has to set a so-called *Divisor Latch Access Bit* $c_{\mathrm{u}}.dlab \in \mathbb{B}$ before accessing the $c_{\mathrm{u}}.div$ field.
The low-level encoding of the transmitted data (including error protection) is configured via the *word length* $c_{\mathrm{u}}.wl \in \mathbb{B}^2$, the stop bit length $c_{\mathrm{u}}.sbl \in \mathbb{B}$, and the parity select $c_{\mathrm{u}}.ps \in \mathbb{B}^3$. We omit details here.

The UART has eleven different registers, which are mapped to eight different addresses. Hence, some addresses are used in different contexts. They map to different registers either depending on the access type (read / write operation) or depending on the value of the divisor latch access bit $c_{\mathrm{u}}.dlab$ (see Table 1).

4.2 Processor-Side Transitions

As already mentioned, the transition function δ_{u} of the serial interface is split into two logical parts: a processor-side and an environment-side transition function.

The processor-side transition function $\delta_{\mathrm{u}}^{\mathrm{mem}}$ defines the behavior of the serial interface when communicating with the processor. Given a current configuration of the serial interface and an input from the processor, it computes an updated serial interface configuration and an output to the processor, i.e., $\delta_{\mathrm{u}}^{\mathrm{mem}}(c_{\mathrm{u}}, mifi) = (mifo, c'_{\mathrm{u}})$.

Table 1. Ports of the serial interface

UART Register / Buffer	Port	Abbreviation	Access Type
Transmitter Holding Buffer	0	THB_{p}	Write, $dlab = 0$
Receiver Buffer	0	RB_{p}	Read, $dlab = 0$
Divisor Latch Low Byte	0	$DLLB_{\mathrm{p}}$	Read / Write, $dlab = 1$
Interrupt Enable Register	1	IER_{p}	Read / Write
Divisor Latch High Byte	1	$DLHB_{\mathrm{p}}$	Read / Write, $dlab = 1$
Interrupt Identification Register	2	IIR_{p}	Read
FIFO Control Register	2	FCR_{p}	Write
Line Control Register	3	LCR_{p}	Read / Write
Line Status Register	5	LSR_{p}	Read

Note that the transition function $\delta_{\mathrm{u}}^{\mathrm{mem}}$ is only partially defined because some processor accesses to the device are considered illegal and lead to an undefined device configuration. Later on we will formulate software conditions excluding all these cases.

In the following we abbreviate a read access to port x by $rd(mifi, x) = mifi.rd \wedge mifi.a = x$ and a write access to port x by $wr(mifi, x) = mifi.wr \wedge mifi.a = x$. Although in general we allow devices to have ports of width 32 bit, the serial interface only has ports of width 8 bit. Hence, only the lower 8 bits of $mifi.din$ and $mifo$ are significant. In the following we assume that $mifo$ will be zero-padded by the device and omit these extra bits here.

Configuration Updates. If the bit $dlab$ is cleared and the processor reads the port receiver buffer having the address RB_{p} and the receiver queue of the serial interface is not empty then its first byte is popped. Furthermore the queue maintaining transmission errors for received bytes is updated, too:

$$rd(mifi, RB_{\mathrm{p}}) \wedge c_{\mathrm{u}}.dlab = 0 \wedge c_{\mathrm{u}}.rb.len > 0 \implies (c'_{\mathrm{u}}.rb = pop(c_{\mathrm{u}}.rb)) \wedge (c'_{\mathrm{u}}.trerr = pop(c_{\mathrm{u}}.trerr))$$

The processor writes the byte to be transmitted into the port transmitter holding buffer. If the corresponding queue is not full, the written byte is pushed into it:

$$wr(mifi, THB_{\mathrm{p}}) \wedge c_{\mathrm{u}}.thb.len < 16 \implies c'_{\mathrm{u}}.thb = push(c_{\mathrm{u}}.thb, mifi.din[7:0])$$

Pending interrupt flags, raised by the device, are cleared if the processor reads the corresponding ports. Reading the receiver buffer clears the received data available- and the time-out interrupt. Similarly reading the interrupt identification register or reading the transmitter holding buffer clears the transmitter holding buffer empty interrupt. Finally the receiver line status interrupt is cleared by reading the line status register:

$$\begin{aligned} rd(mifi, RB_{\mathrm{p}}) &\implies c'_{\mathrm{u}}.rdai = 0 \wedge c'_{\mathrm{u}}.toi = 0 \\ rd(mifi, THB_{\mathrm{p}}) \vee rd(mifi, IIR_{\mathrm{p}}) &\implies c'_{\mathrm{u}}.threi = 0 \\ rd(mifi, LSR_{\mathrm{p}}) &\implies c'_{\mathrm{u}}.rlsi = 0 \end{aligned}$$

By writing the port interrupt enable register, the programmer can specify which interrupt types are enabled, i.e., which can be raised by the device. Since the timeout interrupt is enabled when the received data available interrupt is, only three bits are relevant. The other five bits are ignored:

$$wr(mifi, IER_{\mathrm{p}}) \implies$$
$$(c'_{\mathrm{u}}.erdai = mifi.din[0]) \wedge (c'_{\mathrm{u}}.ethrei = mifi.din[1]) \wedge (c'_{\mathrm{u}}.erlsi = mifi.din[2])$$

The transmit or receive FIFO can be cleared manually by the programmer through writing the FIFO control register FCR_{p}. The bit zero of the FCR_{p} indicates if FIFOs should be used at all. If it is cleared no buffers will be used.

Setting bits one and two will clear the receiver and transmitter buffers, resp.:

$$wr(mifi, FCR_{\mathrm{p}}) \wedge mifi.din[1] = 1 \implies c'_{\mathrm{u}}.rb.len = 0 \wedge c'_{\mathrm{u}}.rb.hd = c_{\mathrm{u}}.rb.tl$$
$$wr(mifi, FCR_{\mathrm{p}}) \wedge mifi.din[2] = 1 \implies c'_{\mathrm{u}}.thb.len = 0 \wedge c'_{\mathrm{u}}.thb.hd = c_{\mathrm{u}}.thb.tl$$

Bit three indicates if DMA is supported. In this paper we do not deal with DMA. Bit four and five of the FCR_{p} are reserved.

If the received data available interrupt is enabled, the last two bits of the FCR_{p} encode at what length of the receive queue an interrupt is generated. Hence, the two bits map to the $rbitl$ component of the serial interface:

$$wr(mifi, FCR_{\mathrm{p}}) \implies c'_{\mathrm{u}}.rbitl = mifi.din[7:6]$$

The first two bits of the line control register are mapped to the transmission word length wl, bit two relates to the stop bit length sbl, bits three to five map to the parity select ps, bit seven is set to access the two divisor bytes, and bit six is reserved:

$$wr(mifi, LCR_{\mathrm{p}}) \implies (c'_{\mathrm{u}}.wl = mifi.din[1:0]) \wedge$$
$$(c'_{\mathrm{u}}.sbl = mifi.din[2]) \wedge (c'_{\mathrm{u}}.ps = mifi.din[5:3]) \wedge (c'_{\mathrm{u}}.dlab = mifi.din[7])$$

By writing the divisor latch high byte register $DLHB_{\mathrm{p}}$ and the divisor latch low byte register $DLLB_{\mathrm{p}}$ the divisor div is set:

$$wr(mifi, DLLB_{\mathrm{p}}) \wedge c_{\mathrm{u}}.dlab = 1 \implies c'_{\mathrm{u}}.div[7:0] = mifi.din[7:0]$$
$$wr(mifi, DLHB_{\mathrm{p}}) \wedge c_{\mathrm{u}}.dlab = 1 \implies c'_{\mathrm{u}}.div[15:8] = mifi.din[7:0]$$

Generated Output. The predicate is_int indicates if for a given configuration of the serial interface c_{u} at least one of the four interrupts types is pending:

$$is\_int(c_{\mathrm{u}}) = c_{\mathrm{u}}.threi \vee c_{\mathrm{u}}.rdai \vee c_{\mathrm{u}}.rlsi \vee c_{\mathrm{u}}.toi$$

In case of a write operation the 32-bit wide data output $mifo$ is irrelevant and therefore set to zero. In case of a read operation the first 24 bits of the output are filled with zeros since the serial interface operates byte-wise.

If the processor reads from the receiver buffer and the receive queue is not empty, the first byte is taken from the queue and returned to the processor:

$$rd(mifi, RB_{\mathrm{p}}) \wedge c_{\mathrm{u}}.dlab = 0 \wedge c_{\mathrm{u}}.rb.len > 0 \implies \\ mifo = head(c_{\mathrm{u}}.rb) \wedge c'_{\mathrm{u}}.rb = pop(c_{\mathrm{u}}.rb)$$

If the processor reads port $DLLB_{\mathrm{p}}$, the lower eight bits of the divisor component div are returned. If it reads port $DLHB_{\mathrm{p}}$, the upper eight bits of the divisor component div are returned:

$$rd(mifi, DLLB_{\mathrm{p}}) \wedge (c_{\mathrm{u}}.dlab = 1) \implies mifo = c_{\mathrm{u}}.div[7:0] \\ rd(mifi, DLHB_{\mathrm{p}}) \wedge (c_{\mathrm{u}}.dlab = 1) \implies mifo = c_{\mathrm{u}}.div[15:8]$$

When reading the interrupt enable register the output encodes the four flags indicating which interrupt types are enabled:

$$rd(mifi, IER_{\mathrm{p}}) \wedge (c_{\mathrm{u}}.dlab = 0) \implies mifo = 0^5 \circ c_{\mathrm{u}}.erlsi \circ c_{\mathrm{u}}.ethrei \circ c_{\mathrm{u}}.erdai$$

The type of the interrupt that caused the eev flag to be set can be checked by reading the interrupt identification register. For $rd(mifi, IIR_{\mathrm{p}})$ we define $mifo = 1100 \circ is \circ \neg is\_int(c_{\mathrm{u}})$ where the three interrupt status bits $is \in \mathbb{B}^3$ are defined as follows:

$$is = \begin{cases} 011 & \text{if } c_{\mathrm{u}}.rlsi \\ 010 & \text{if } \neg c_{\mathrm{u}}.rlsi \wedge (c_{\mathrm{u}}.rdai \vee c_{\mathrm{u}}.toi) \\ 110 & \text{if } \neg c_{\mathrm{u}}.rlsi \wedge \neg(c_{\mathrm{u}}.rdai \vee uart.toi) \\ 001 & \text{if } \neg uart.rlsi \wedge \neg(uart.rdai \vee uart.toi) \wedge uart.threi \end{cases}$$

The line status register is a read-only register which encodes the polling mode information of the transmitter and the receiver queues. Furthermore, in case of a transmission error (i.e., in case of *line status interrupt*), this register provides the error type. Remember that the component $c_{\mathrm{u}}.trerr$ stores parity, framing and break errors of all bytes in the receiver queue. When reading LSR_{p}, the errors occurred in the head of the queue are reported in bits two, three and four. Let $errQ$ denote whether at least one error occurred in any of the bytes in the queue. Reading the port LSR_{p} results in:

$$rd(mifi, LSR_{\mathrm{p}}) \implies \\ mifo = errQ \circ c_{\mathrm{u}}.edhr \circ c_{\mathrm{u}}.ethb \circ head(c_{\mathrm{u}}.trerr)[2:0] \circ c_{\mathrm{u}}.oe \circ c_{\mathrm{u}}.dr$$

The line control register can also be read out. As mentioned before it contains the parity select, word length, stop bit length, set break interrupt enable flag and the divisor latch bit components of the serial interface:

$$rd(mifi, LCR_{\mathrm{p}}) \implies mifo = c_{\mathrm{u}}.dlab \circ c_{\mathrm{u}}.ebi \circ c_{\mathrm{u}}.ps \circ c_{\mathrm{u}}.sbl \circ c_{\mathrm{u}}.wl$$

4.3 Environment-Side Transitions

We describe the interaction of the serial interface with the environment, which is given by the environment-sided transition function $\delta_{\mathrm{u}}^{\mathrm{env}}$. This function takes an input from the environment and a serial interface configuration and it returns an updated serial interface configuration and an output to the environment, i.e., $\delta_{\mathrm{u}}^{\mathrm{env}}(c_{\mathrm{u}}, eifi) = (eifo, c'_{\mathrm{u}})$.

The input from that environment is given by (i) a bit $eifi.tshrready$ indicating if the transmitter shift register is empty and hence the next byte of the transmitter queue can be sent, (ii) a bit $eifi.serdinvalid$ indicating if new and valid data was received, (iii) the serial input data $eifi.serdin$, (iv) three bits indicating parity, framing, and break error, $eifi.pe$, $eifi.fe$ and $eifi.be$, (v) and a bit $eifi.to$ indicating a time-out interrupt. Since no time is modeled, a non-deterministic input from the environment signals time-out. The only output of the serial interface to the environment is the byte being sent.

If the transmission shift register is empty $eifi.tshrready$ and the transmitter queue has data in it, $c_{\mathrm{u}}.thb.len > 0$, the first byte of the queue is sent, $head(c_{\mathrm{u}}.thb)$ to the external environment. Otherwise, a special empty output is transmitted, i.e., 0^8.

The byte written to the external environment is taken from the transmitter queue:

$$eifi.tshrready \wedge c_{\mathrm{u}}.thb.len > 0 \implies c'_{\mathrm{u}}.thb = pop(c_{\mathrm{u}}.thb)$$

If the receive queue is not full, received bytes are added to it. Furthermore the queue maintaining the parity, framing and break errors is updated for the received byte:

$$eifi.serdinvalid \wedge c_{\mathrm{u}}.rb.len < 16 \implies c'_{\mathrm{u}}.rb = push(c_{\mathrm{u}}.rb, eifi.serdin) \wedge c'_{\mathrm{u}}.trerr = push(c_{\mathrm{u}}.trerr, eifi.pe \circ eifi.fe \circ eifi.be)$$

If a new byte is received although the receive queue is full, then the new byte is dropped and an error is indicated by raising the overrun flag:

$$eifi.serdinvalid \wedge c_{\mathrm{u}}.rb.len = 16 \implies c'_{\mathrm{u}}.oe = 1$$

The interrupt pending signals are raised if (i) the transmitter queue is empty and the environment signals through $eifi.tshrready$ that the next byte can be sent, $c'_{\mathrm{u}}.threi = c'_{\mathrm{u}}.thbp.len > 0$, (ii) the length of the receiver queue reaches the specified trigger level (receive data available interrupt), $c'_{\mathrm{u}}.rdai = c_{\mathrm{u}}.rb.len \geq itl(c_{\mathrm{u}}.rbitl)$, (iii) or a framing, parity, break or an overrun error occurs (line status interrupt), $c'_{\mathrm{u}}.rlsi = eifi.fe \vee eifi.pe \vee c'_{\mathrm{u}}.oe$, (iv) or the receiver queue is non-empty and the external environment signals the occurrence of a time-out, $c'_{\mathrm{u}}.toi = eifi.to \wedge c'_{\mathrm{u}}.rb.len > 0$.

In the polling driven mode the configuration is updated similarly: (i) $c'_{\mathrm{u}}.ethb$ is set if the transmitter queue is not empty, $(c'_{\mathrm{u}}.thbp.len = 0)$, (ii) $c'_{\mathrm{u}}.dr$ is set if the receiver queue is non-empty, $(c'_{\mathrm{u}}.rb.len > 0)$, (iii) $c'_{\mathrm{u}}.edhr$ is set if both the transmitter queue and the shift register are empty, $(c'_{\mathrm{u}}.thbp.len = 0) \wedge eifi.tshrready$.

4.4 Software Conditions and Environment Restrictions

The transition function δ_{u} is not total. Undefined cases are related to over- and underruns of the queues and illegal accesses to unmodeled or write-only ports. Formally, we characterize these cases by predicates over memory input and UART configurations:

- The line-status register must not be written, $\neg wr(mifi, LSR_{\mathrm{p}})$, and the unmodeled ports MCR_{p} and MSR_{p} must not be accessed, $mifi.a \notin \{MSR_{\mathrm{p}}, MCR_{\mathrm{p}}\}$.
- The receiver buffer must not be read when empty and the transmitter buffer must not be written to when full. Formally, if $c_{\mathrm{u}}.dlab = 0$ then $rd(mifi, RB_{\mathrm{p}}) \implies c_{\mathrm{u}}.rb.len > 0$ and $wr(mifi, THB_{\mathrm{p}}) \implies c_{\mathrm{u}}.thb.len < 16$.

Only if these software conditions are met, we can assume the model to be accurate. The driver programmer is responsible for discharging them. For example, a driver which writes no more than 16 byte chunks between each two transmitter holding buffer empty interrupts, obviously fulfills the second condition.

For proving correctness of a driver implementation we need to impose further restrictions on the behavior of the environment.

Liveness. We need to assume liveness of the sending part: data in the transmitter buffer must eventually be sent, $\forall i\, \exists j > i\, .\, seq_{\mathrm{PD}}(j) = D_{\mathrm{uart}} \wedge eifiseq(j).tshrready = 1$.

Also the processor is assumed to be live, $\forall i\, \exists j > i\, .\, seq_{\mathrm{PD}}(j) = P$. While liveness can be assumed by the programmer, it has to be shown in the hardware correctness proof.

Overrunning Receiver Queue. The speed of the environment sending packets to the serial interface is not related in any sense to the speed of the processor; packets arrive completely non-deterministically. The question is: how can a driver programmer under these circumstances assure that no packets are lost due to overrunning queues?

This is a tricky task. In a first approach we might impose timing restriction on the environment. Hardware implementation details like caches, pipelining, etc. are invisible in the ISA. Thus, the numbers of instructions executed cannot be related to real time and a relation between transmission speed and ISA execution cannot be established.

Note that the problem of overrunning queues is not inherent to our way of modeling. It is a problem that a device programmer must expect and deal with in non-real-time operating systems, too. This situation leads to serious difficulties in the formalization of the correctness statements for serial interface drivers. For example, it is impossible to prove that all key presses sent from a keyboard to a serial interface are finally processed by the driver because the model contains runs in which the environment is too fast leading to a queue overrun. There are three approaches to deal with the problem:

```
 0: addi r3,r0,#Da(D_uart)  (1.1)
 4: addi r4,r0,#3           (1.2)
 8: sw LCR_p·4(r3),r4       (1.3)
12: sw IER_p·4(r3),r0       (1.4)
16: addi r0,#14, r4         (1.5)
20: sw FCR_p·4(r3),r4       (1.6)
24: lw r6, 0(r1)            (2.1)
28: sw THB_p·4(r3), r6      (2.2)
32: slri r6, r6, #8         (2.3)
36: sw THB_p·4(r3), r6      (2.4)
40: slri r6, r6, #8         (2.5)
44: sw THB_p·4(r3), r6      (2.6)
48: slri r6, r6, #8         (2.7)
52: sw THB_p·4(r3), r6      (2.8)
56: addi r1, r1, #4         (2.9)
60: lw r4, LSR_p·4(r3)      (3.1)
64: andi r4, r4, #32        (3.2)
68: beqz r4, #-12           (3.3)
72: nop                     (3.4)
76: subi r5, r5, #1         (4.1)
80: bnez r5, #-60           (4.2)
84: nop                     (4.3)
```

Fig. 2. UART driver. We assume that registers `r1` and `r2` are preset to a and n.

1. *Model overruns in specification and use software synchronization.* A widely used mechanism is called software flow control: the receiver signals the sender when ready / unable to accept new data via the special characters Xon / Xoff.

2. *Hardware synchronization.* Synchronization can also be implemented directly in hardware (called *autoflow control*), as was done for the UART 16750. Using such hardware an assumption can be introduced stating that the environment is not sending new data while the receiver buffer is still full.

3. *Worst case execution time (WCET) analysis.* Good run-time estimates require a cycle-accurate model for the target processor. Indeed, there are tools for several architectures to precisely estimate the WCET of given programs, e.g., [17]. By analyzing the serial interface driver and parts of the kernel, such as the interrupt handlers, we can compute the latency of processing data received at the serial interface. This yields a maximum baud rate under which the driver may be run safely without overruns.

5 Example: A Simple UART Driver

We construct a simple device driver and sketch its correctness proof with respect to the ISA of Sect. 3. The driver writes n words from the processor's memory, starting at address a, to the serial interface with index D_{uart} and base address $Da(D_{\mathrm{uart}})$. Its code is shown in Fig. 2; its size is approximately an order of magnitude smaller than the code of a realistic driver for the UART 16550A. We use a MIPS-like syntax. GPRs, immediates, and register-indexed memory operands are denoted as $\mathtt{r}k$, $\#l$, and $m(\mathtt{r}n)$. Lines are prefixed with an offset to a certain code base address cba. Arrows indicate jump targets; all jumps are executed with one delay slot.

To state the driver correctness, we use the auxiliary function $purge$. For a device index idx and an external output sequence $eifoseq$ it returns the sub sequence of external outputs for device idx.

Let seq_{PD} and $eifiseq$ denote a computational sequence and an external input sequence fulfilling the liveness assumption. Let c^0_{PD} denote an initial configuration which starts with the execution of the driver, $c^0_{\mathrm{P}}.dpc = cba$, and where the word

count and start address are stored in the first two registers, i.e., $c^0_{\mathrm{P}}.gpr[1] = a$ and $c^0_{\mathrm{P}}.gpr[2] = n$.

Furthermore let c^i_{PD} and $eifoseq^i$ denote the reached state and generated output after the execution $run_{\mathrm{PD}}(i, seq_{\mathrm{PD}}, eifiseq, c^0_{\mathrm{PD}})$ of some i steps of the combined system.

Theorem 1 (Functional correctness). *There exists some step number e, after which the driver finished execution and the n words from the processor's memory are output to the external environment:* $purge(eifoseq^e, D_{\mathrm{uart}}) = c^0_{\mathrm{P}}.m_{4\cdot n}(a)$

Proof. The main part of the code is the *outer loop* in parts (2) to (4). It is traversed n times, sending a word over the serial interface in each iteration. Before iteration $j < n$ and after iteration $j = n$ of the loop after a certain number $s(j)$ of steps the following invariants have to hold: (i) j words have been written to the environment, $purge(eifoseq^{s(j)}, D_{\mathrm{uart}}) = c^0_{\mathrm{P}}.m_{4\cdot j}(a)$, (ii) the first address not yet copied and the number of remaining words are stored in $gpr[1]$ and $gpr[4]$, and (iii) the device has an empty transmitter holding buffer, interrupts disabled, and a cleared $dlab$ flag. The existence of $s(j)$ and the invariants are shown by induction over j.

Initially, the device invariant is established by code part (1) writing the ports LCR_{p}, IER_{p}, and FCR_{p}. For $j > 0$, correctness of the code that copies a word from memory to the transmitter holding buffer, part (2), and the polling loop, part (3) have to be shown. After part (2), $4 - c_{\mathrm{u}}.thb.len$ bytes have been transmitted; the other bytes will have been transmitted after the polling loop exits. To show termination of these parts, the liveness condition over the computational and external sequence has to be applied.

6 Conclusion and Future Work

We have presented the detailed formal model of a serial interface controller, the UART 16550A [16]. By combining this model with the formal model of a processor ISA, we obtained a formal model of a processor in which the UART may be accessed as a memory-mapped device. All presented models have been specified in the theorem prover Isabelle/HOL [18]. The formalized ISA resembles the DLX instruction set architecture that was taken as a specification for the VAMP processor [2, 3].

Our Isabelle/HOL formalization defines a precise programming model for device drivers and may be used as the basis of an integrated, self-contained formal driver verification environment. Thus, it is relevant for both device programmers and verification engineers.

For the programmer, the model is a succinct description of the visible state of the device and its interaction with the external environment and the processor. Moreover, *environmental conditions*, which the programmer may assume, and *software conditions*, which the programmer must satisfy, precisely define the rules for implementing

a functionally correct device driver. An example of such a driver, transmitting data via a serial interface, was given in Sect. 5.

In addition, the model may be used by the verification engineer to develop mathematical software correctness proofs and to check them with a computer-aided verification system. A sketch of such a proof was given in Sect. 5. In contrast to related work, the high level of detail in our device models even allows the verification of complex properties like functional correctness rather than just control or protocol properties.

Our further work in this area can be split into two parts. First, we plan to formalize and extend the implementation and correctness proofs from Sect. 5 to cover data reception and successful communication between two serial interfaces. Second, in the broader context of the attempted system verifications in Verisoft, the scope of our modelling and verification effort needs to be extended to cover all system layers from the gate-level hardware of the VAMP processor [2, 3] with devices up to user-level device drivers for a variety of standard devices (e.g., serial interface, hard disk [6], FlexRay-like bus controller). The final result of this effort is a stack of computational models with device support; adjacent layers in this model stack will be related to each other by simulation theorems.

References

1. The Verisoft Consortium: The Verisoft Project. http://www.verisoft.de/ (2003)
2. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.: Instantiating uninterpreted functional units and memory system: Functional verification of the VAMP. In Geist, D., Tronci, E., eds.: CHARME'03. Volume 2860 of LNCS. Springer (2003) 51–65
3. Dalinger, I., Hillebrand, M., Paul, W.: On the verification of memory management mechanisms. In Borrione, D., Paul, W., eds.: CHARME'05. Volume 3725 of LNCS. Springer (2005) 301–316
4. Schmaltz, J.: A formal model of lower system layer. In: FMCAD'06, IEEE/ACM Press (2006) 191–192
5. Knapp, S., Paul, W.: Pervasive verification of distributed real-time systems. In Broy, M., Grünbauer, J., Hoare, T., eds.: Software System Reliability and Security. Volume 9 of IOS Press, NATO Security Through Science Series. (2007) To appear.
6. Hillebrand, M., In der Rieden, T., Paul, W.: Dealing with I/O devices in the context of pervasive system verification. In: ICCD '05, IEEE Computer Society (2005) 309–316
7. Cohen, B.: Component design by example: A step-by-step process using VHDL with UART as vehicle. VhdlCohen (2000)
8. Berry, G., Kishinevsky, M., Singh, S.: System level design and verification using a synchronous language. In: ICCAD, IEEE Computer Society / ACM (2003) 433–440
9. ALDEC – The Design Verification Company: UART nVS. http://www.aldec.com/products/ipcores/_datasheets/nSys/UART_nVS.pdf (2006)
10. Rashinkar, P., Paterson, P., Singh, L.: System-on-a-Chip Verification: Methodology and Techniques. Kluwer Academic Publishers, Norwell, MA, USA (2001)
11. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In Dwyer, M.B., ed.: SPIN. Volume 2057 of LNCS. Springer (2001) 103–122
12. Microsoft Corporation: SDV: Static driver verifier. http://www.microsoft.com/whdc/devtools/tools/sdv.mspx (2004)
13. Hallgren, T., Jones, M.P., Leslie, R., Tolmach, A.P.: A principled approach to operating system construction in Haskell. In Danvy, O., Pierce, B.C., eds.: ICFP, ACM (2005)
14. Holzmann, G.J.: New challenges in model checking. http://www.easychair.org/FLoC-06/holzmann_25mc_floc06.pdf (2006) Symposium on 25 years of Model Checking, Seattle, USA. Invited talk.

15. Müller, S., Paul, W.: Computer Architecture: Complexity and Correctness. Springer (2000)
16. National Semiconductor: PC16550D – universal asynchronous receiver / transmitter with FIFO's. `http://www.national.com/ds.cgi/PC/PC16550D.pdf` (2005)
17. Ferdinand, C., Heckmann, R.: Verifying timing behavior by abstract interpretation of executable code. In Borrione, D., Paul, W., eds.: CHARME'05. Volume 3725 of LNCS. Springer (2005) 336–339
18. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic. Volume 2283 of LNCS. Springer (2002)