Reliable Device Drivers Require Well-Defined Protocols[*]

Leonid Ryzhyk Timothy Bourke Ihor Kuz
NICTA[†] and the University of New South Wales
Sydney, Australia
{leonidr,tbourke,ikuz}@cse.unsw.edu.au

Abstract

Current operating systems lack well-defined protocols for interaction with device drivers. We argue that this hinders the development of reliable drivers and thereby undermines overall system stability. We present an approach to specify driver protocols using a formalism based on state machines. We show that it can simplify device programming, facilitate static analysis of drivers against protocol specifications, and enable detection of incorrect behaviours at runtime.

1. Introduction

Device drivers constitute a major source of instability in modern computer systems. While accounting for about 70% of the operating system code, drivers typically contain several times more errors per line of code than other system components [3]. As a result, the majority of operating system failures can be traced to software faults in device drivers.

A number of projects have explored techniques to improve device driver reliability and decrease the reliance of systems on driver correctness. L4 [10, 11], Xen [6], Minix [8], and Nooks [15] experiment with hardware-enforced isolation of device drivers. Singularity [5] and SafeDrive [16] focus on language-based isolation techniques. Vault [4], Singularity [5], and SLAM [1] explore static approaches to detect the incorrect use of resource management protocols and other operating system APIs. Finally, the Devil [12] project provides a hardware interface definition language aimed at simplifying programming of the device interaction layer of device drivers.

Despite these improvements, driver development remains a tedious error-prone task. The source of the complexity is that device drivers must simultaneously comply with two sophisticated interfaces: the hardware interface of the device and the software interface of the operating system. Both interfaces are large, highly concurrent and often poorly defined.

In this paper we focus on improving the interface between a driver and the operating system. Contemporary operating systems are required to manage devices that provide advanced capabilities, such as request queuing, power management, and hot plugging. In order to access these capabilities, operating system designers define complex protocols for interaction with device drivers. These protocols allow a high degree of concurrency, with multiple user I/O requests and hardware IRQ notifications being processed simultaneously. Additionally, the protocols require that driver programmers manage a large amount of state, including device status information, the current protocol phase (e.g., active, switching to low power mode, device disconnected, etc.), and the status of individual I/O requests.

Typically, operating system documentation describes protocols between the operating system and device drivers in terms of a set of functions that the driver must implement and a set of callbacks that the driver can invoke. This leaves the constraints on the ordering, timing and arguments of invocations implicit in the operating system implementation. As a result, the driver developer is forced to construct and maintain a mental model of the protocol between a driver and its environment. If this model diverges from the one used by the operating system, driver implementation becomes susceptible to subtle errors. Furthermore, constraints of the protocol can change between operating system revisions, turning correct drivers into faulty ones [14].

Besides being a major source of bugs, poorly defined protocols make it impossible to verify the correctness of a driver either statically or at runtime. In the absence of a precise specification of permitted behaviours it is meaningless to ask whether a driver is correct.

To overcome these issues, we propose specifying driver

[*] This research was supported in part by a grant of computer software from Telelogic.

[†] NICTA is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs

protocols—including the ordering, timing and content of messages exchanged across a driver interface—using a formalism based on finite state machines (FSMs). This formalism makes automatic reasoning about driver behaviour possible, but at the same time is easily readable by programmers and can be used as a core item of documentation. We argue that this approach can reduce the number of bugs in drivers. In addition, it facilitates static analysis of driver implementations against protocol specifications and the detection of incorrect behaviours at runtime.

We want to stress that we are not aiming to fix or improve an existing driver framework (although the proposed approach could be incorporated into an existing system). Rather, we want to answer the question: how should a driver framework be designed from the ground-up for improved reliability? In order to validate the state-machine based approach, we are integrating it into our new user-level driver framework for the L4/Iguana [9] embedded operating system called Dingo.

2. The Dingo driver framework

The definition of driver protocol state machines has been developed in tandem with the Dingo driver framework. A driver in Dingo is represented as an object whose functionality is accessed through ports. A port is a collection of required and provided interfaces that comprise a distinct interaction point between the driver and its environment. Ports are typed entities. For instance, a driver for the PCI-based RTL8139D Ethernet controller has three ports: the `csr` port of type `PCIIORegion` for access to the PCI I/O region containing control and status registers of the controller, the `irq` port of type `IRQSource` for interrupt delivery, and the `ethernet` port of type `EthernetController` that exports the Ethernet controller functionality to the operating system (Figure 1).

Dingo drivers are single-threaded and non-blocking[1]. Operations that involve waiting for an external event, e.g., a hardware interrupt, are split into two non-blocking phases: (i) request and (ii) completion. Furthermore, the execution model is non-reentrant, that is, when the driver calls the operating system, the system may not call back into the driver. As a result, calls to the operating system are atomic from the point of view of a driver.

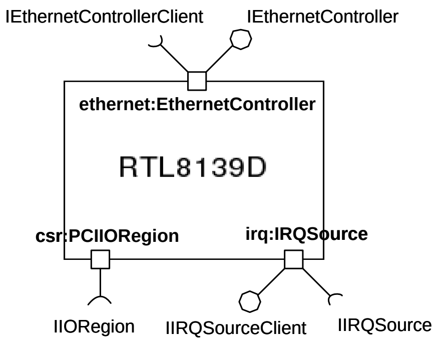


Figure 1. RTL8139D Ethernet controller driver. Squares at the object boundary represent driver ports. Lollipop and socket symbols denote, respectively, provided and required interfaces.

3. Driver protocol state machines: the basic model

A *protocol state machine* defines which sequences of incoming and outgoing calls are permitted through a port of a certain type. A driver must comply with the protocol state machines of all its ports. Protocol state machines are specified in a simplified form [2] of UML Statecharts [13]. The language allows refinement and concurrency, with semantics given in terms of classical automata that define sets of finite and infinite traces. It is sufficiently expressive to capture many of the constraints on driver behaviour, has a tractable formal semantics and a graphical depiction that is readily understood by software engineers.

We present details of the protocol state machines formalism using the example of the `EthernetController` port protocol that must be implemented by all Ethernet controller drivers in the Dingo framework. Figure 2 summarises the protocol interfaces and depicts the associated statechart. States of the statechart represent conceptual states and activities of the driver that are visible at the protocol level. State transitions are triggered by method calls to and from the driver. The syntax of state transition labels is `<trigger>[<guard>]/<action>`, where `<guard>` and `<action>` elements are optional. Question marks (`?`) in trigger names denote incoming calls from the operating system to the driver, i.e. events where control passes into the driver. Exclamation marks (`!`) denote outgoing calls. The points where control returns from the driver are not marked explicitly.

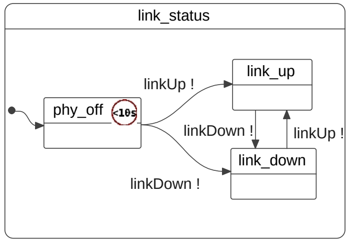
Several standard Statecharts features [7] make modelling more convenient. The compact representation of complex protocols is achieved by organising states into a hierarchy. Several states comprising a single higher-level activity can

[1] Multithreaded driver framework APIs constitute a major source of driver bugs [3]. As the performance of most drivers is bounded by device rather than CPU speed, the benefits of multithreaded drivers are minimal, if any. One notable exception are high-speed network controllers with independent receive and transmit engines. In a single-threaded framework such devices can be managed by two separate drivers in order to exploit intrinsic parallelism. A detailed discussion of threading issues is beyond the scope of this paper.

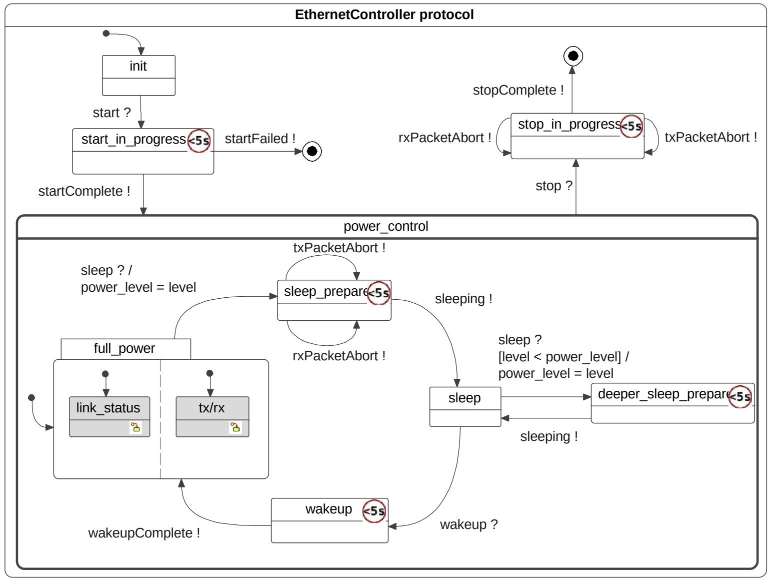
IEthernetController interface	
void start()	initialise the driver
void stop()	shutdown the driver, release all resources
void enable()	enable receiver and transmitter
void disable()	disable receiver and transmitter
void txContinue()	more packets are available in the output queue
void rxAllocContinue()	more buffers for incoming packets are available
void sleep(power_level_t level, wakeup_pattern_t pattern)	put the device in a lower power mode
void wakeup()	put the device in full-power mode

IEthernetControllerClient interface	
void startComplete()	driver initialisation completed successfully
void startFailed()	driver initialisation failed
void stopComplete()	driver shutdown completion notification
void linkUp()	link established notification
void linkDown()	no valid link established notification
void enableComplete()	enable completion notification
void disableComplete()	disable completion notification
bool txPacketPull(IMemDescriptor**)	dequeue a packet from the output queue
bool rxPacketAlloc(IMemDescriptor**, size_t)	allocate a buffer for an incoming packet
void txPacketDone(IMemDescriptor*)	packet transmission notification
void rxPacketInput(IMemDescriptor*)	packet reception notification
void rxPacketAbort(IMemDescriptor*)	return an unused rx buffer during driver shutdown or deinitialisation
void txPacketAbort(IMemDescriptor*)	return an unused tx buffer during driver shutdown or deinitialisation
void sleeping()	transition to the lower power mode is complete
void wakeupComplete()	wakeup completion notification

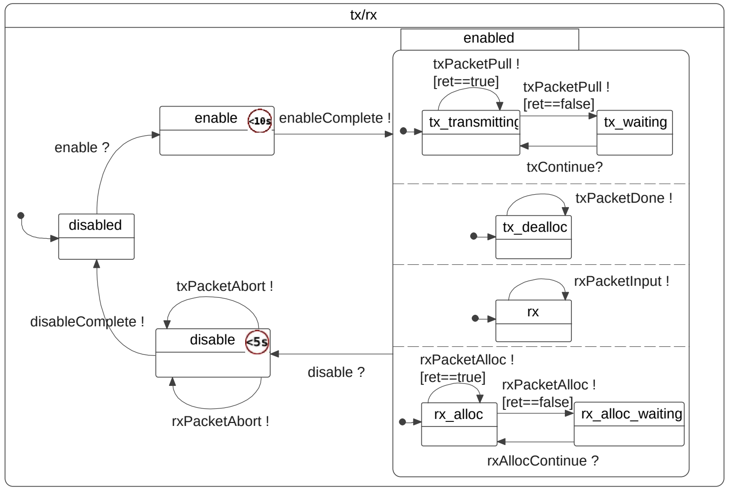
(a) Summary of interfaces involved in the protocol.



(c) `link_status` sub-statechart.



(b) The main statechart of the protocol.



(d) `tx/rx` sub-statechart

Figure 2. `EthernetController` **protocol interfaces and state machine.**

be clustered into a *super-state* (e.g., power_control in Figure 2b). A single transition originating from a super-state preempts any of its internal states (e.g., the stop? transition in Figure 2b). When the statechart is too large to fit in a single diagram, a super-state can be collapsed into a simple state and its content can be moved to a separate diagram. For example, shadowed states in Figure 2(b) represent super-states expanded in Figures 2(c) and (d).

A device driver may be simultaneously involved in several loosely related activities. Often, these activities correspond to services provided by different functional units of the controlled device. They are conveniently modelled using *orthogonal regions* which make the logical concurrency apparent and reduce the number of explicit states (though not the size of the underlying transition system). When the protocol state machine is in a state with several orthogonal regions, each of the regions simultaneously constrains the systems behaviour. On state diagrams, orthogonal regions are separated with dashed lines. For instance, the full_power super-state in Figure 2b contains two orthogonal regions: the left region describes the link status reporting functionality (implemented in the PHY functional unit of the Ethernet controller), while the right region describes the process of packet transmission and reception (implemented in the MAC unit).

Device driver protocols often involve timing constraints. To capture these constraints, the protocol state machine formalism allows states to be annotated with upper timing bounds to be monitored by watchdog timers. A protocol is violated if, after entry into such a state, the given amount of time passes without the triggering of a transition leading to a different state. For example, the start_in_progress state in Figure 2b has a label indicating that the driver should complete initialisation within 5 seconds after receiving a start? request.

Another important feature of protocol state machines are *protocol variables*. Some elements of a protocol are inconvenient to model with explict states and are more naturally described using variables. Variables can be used in transition guards and actions. For example, the Ethernet controller protocol contains numeric variable power_level describing the current power level at which the device is operating (Figure 2b). We require that variables are finitely bounded; they are a convenience rather than an increase in modelling power.

The semantics of the protocol state machine are as follows: any protocol event (i.e., a call to a provided or a required interface of the port) that triggers a valid transition in the state machine complies with the protocol specification. An event that does not trigger any valid transitions violates the protocol specification. A formal semantics of protocol state machines is presented in [2].

We briefly overview the EthernetController protocol statechart. The EthernetController protocol diagram (Figure 2b) specifies initialisation and shutdown messages and the power management operations of an Ethernet driver. The actual network connection management and data transmission happen inside the link_status and tx/rx states. The link_status diagram specifies how the driver reports network connection status changes to the operating system. For instance, it asserts that the driver must initialise its physical-layer interface and report link status within 10 seconds of the start_complete notification. The tx/rx diagram describes rules of interfacing with the receive and transmit engines of the controller. For example, the topmost region of the enabled superstate details how the driver pulls packets from the outgoing packet queue of the operating system: whenever the driver has space for another packet in its transmit buffers, it calls txPacketPull to pull a packet from the queue. If the queue is empty (txPacketPull returns false), the driver stops polling the queue until the operating system notifies it that there is more data available for transmission with a txContinue notification.

4. Extensions

Some types of constraints cannot be captured using the formalism as just presented. One example are constraints that specify relationships among different ports of the driver. For instance, the RTL8139D driver is not allowed to issue any PCI bus transactions after the controller enters the power-saving mode. Capturing this constraint requires the state machine of the csr port to react to a subset of power management methods of the ethernet port. Figure 3 shows the resulting state machine for this port. It specifies when the read and write events may occur relative to occurrences of the sleeping and wakeup events of the EthernetController protocol.

In general, it is too limiting to consider a protocol in isolation from the protocols of other ports of the same driver. We therefore define the semantics of multiple protocol state machines applied to a single driver: an event complies with the driver protocol if and only if it triggers a valid state transition in all protocol state machines associated with driver ports that contain this event in their scopes.

Another group of constraints that require a more expressive formalism are those related to states of individual I/O transactions. In the EthernetController protocol, every buffer obtained by the driver using a rxPacketAlloc request should be either filled with data and returned to the operating system via a rxPacketInput operation, or reclaimed by a call to rxPacketAbort when the driver is in the process of being stopped or disabled, and before it enters a disabled state.

The lifecycle of a packet is specified by the RxPacket

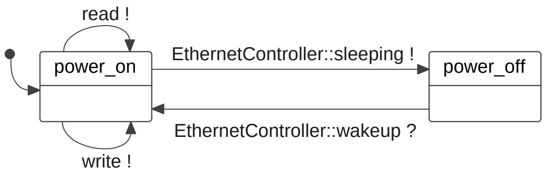


Figure 3. `PCIIORegion` **protocol state machine.**

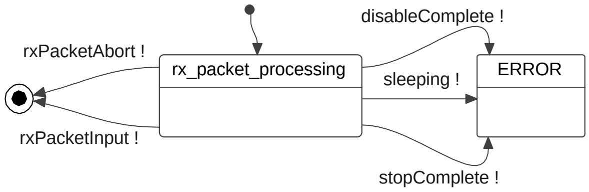


Figure 4. Incoming packet lifecycle statechart.

statechart in Figure 4. In order to incorporate per-packet state information in the protocol specification, we spawn a new `RxPacket` state machine on each successful `rxPacketAlloc` call. Relevant messages are delivered to all per-packet state machines. This extension complicates static analysis of drivers and increases the overhead of runtime failure detection (see Section 5 below). It is, however, essential for maintaining an adequate model of driver behaviour. It enables the definition and detection of common types of resource management errors in drivers. The exact syntax and semantics of such dynamic state machines is part of ongoing work.

5. Using protocol state machines to improve driver reliability

In this section we describe how protocol state machines help to improve the quality of device drivers.

Fault avoidance As mentioned above, in existing driver frameworks the driver developer is required to maintain an implicit mental model of interaction between a driver and its environment. As a result, driver control logic becomes complicated, and susceptible to errors that are often manifested in corner cases, when the driver is handling an uncommon combination of I/O requests. In addition, programmer attention is diverted from the primary purpose of the driver: controlling the hardware.

Our approach addresses this problem by making explicit the state of the protocol along with the set of events that the driver should be prepared to handle or generate in any state. We believe that a driver framework that includes protocol state machine specifications (similar to the one shown in Figure 2) as part of its documentation has the potential to greatly simplify device programming and thereby to reduce the number of bugs in device drivers.

Fault detection via static analysis Static analysis is a set of techniques for validating properties against the source code of a program. A protocol state machine could be treated as a property to be checked against an abstracted version of a device driver. Recent approaches [1] process the source code of device driver to produce abstract models which may then be verified by exhaustive simulation (model-checking). Ball et al. [1] note that devising the properties to check was difficult due to the complexity and ambiguity of the Windows device driver API. Our approach seeks to avoid such problems by developing a clearly-defined API backed by a formal semantics.

Runtime failure detection Alternatively, protocol violations can be detected at runtime by capturing exchanges across the driver interface and tracking the state of the underlying state machine. This approach to failure detection is completely transparent to the driver developer and does not require any per-driver effort, as the protocol state machine needs only be fefined once for a class of drivers (e.g., Ethernet or audio). Moreover, given the protocol state machine specification, it is possible to automatically generate a corresponding failure detector. For example, we have generated a failure detector for the RTL8139D driver by compiling the statecharts in Figures 2 and 3 with the off-the-shelf Rhapsody UML tool. The resulting state machine implementation is instantiated at runtime and triggered on each call to or from the driver. The occurrence of a call in a state where no corresponding transition exists represents a fault in the driver. Other driver faults may be detected by instantiating and monitoring timers when in timeout states. Runtime failure detection introduces the overhead of triggering the protocol state machine on each protocol event, but we expect this overhead to be low compared to the cost of the actual driver operations.

6. Related work

Several projects have attempted to formalise device driver interfaces using FSMs. Vault [4] and SLAM [1] both represent invariants of the driver framework API as FSMs and statically check a driver against these invariants. However, these tools are aimed at existing systems that have not been developed with a rigorous model of driver protocols in mind. Therefore, they capture only selected aspects of the API, and do not allow the complete description of the state space and state transitions of the driver protocols.

In the Singularity [5] operating system, components, including device drivers, interact through channels. The channel contract is described by a FSM that specifies the sequences of messages that may be exchanged. Channels are entities of the Singularity implementation language, Sing#. The Sing# compiler statically verifies that the driver respects channel contracts. Thus, in Singularity support for protocol state machines is integrated with the programming language, which is simultaneously the major advantage and limitation of the approach. In contrast, we aim to provide a separate formalism to specify driver protocols and to make this formalism as expressive as necessary in order to capture a rich set of constraints on driver behaviour. As a result, the formalism presented here supports a number of features unavailable in Singularity, including orthogonal states and super-states, timeout states, dynamic spawning of per-transaction state machines, protocol variables, and constraints involving multiple ports of the driver.

7. Current status and future work

At this stage we have implemented a state-machine based Ethernet framework, an RTL8139D Ethernet driver and a runtime failure detector for it, all running on real hardware. Our experience with protocol state machines has, to date, been positive: we found that interface definition in the form of a statechart makes it easier to implement and understand the control logic of the driver. We tested the failure detector by introducing various faults into the driver implementation. We were able to successfully detect protocol violations, such as the driver polling the output packet queue despite being notified that there are no packets available for transmission, or the driver not being able to wake up from a sleep state.

Current and future work includes a more thorough evaluation of the proposed approach on a variety of driver types, development of complete formal semantics of protocol state machines, extensions of the protocol state machine model with new capabilities, such as protocol inheritance and optional protocol features, and static checking of device drivers against protocol specifications.

8. Conclusions

We have presented an approach to specifying the complex protocols between device drivers and the operating system using a state-machine formalism. We demonstrated via the Ethernet protocol example that protocol state machines allow a natural specification of many important aspects of driver behaviour including advanced features such as power management. Protocol state machines have the potential to greatly simplify device programming and to reduce bugs. In addition, they facilitate static analysis of driver implementations against protocol specifications and the detection of incorrect behaviours at runtime.

9. Acknowledgements

We want to thank Nicholas Fitzroy-Dale, Charles Gray, Gernot Heiser, and Sergio Ruocco for their valuable feedback on an early draft of the paper.

References

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys Conf.*, pages 73–85, 2006.

[2] T. Bourke and L. Ryzhyk. Semantics of driver protocol state machines. URL http://www.cse.unsw.edu.au/~leonidr/psm.pdf, 2007.

[3] A. Chou, J.-F. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *18th SOSP*, pages 73–88, Lake Louise, Alta, Canada, Oct 2001.

[4] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, pages 59–69, 2001.

[5] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *EuroSys Conf.*, 2006.

[6] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson. Safe hardware access with the Xen virtual machine monitor. In *1st OASIS*, 2004.

[7] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, Jun 1987.

[8] J. N. Herder, H. Bos, and A. S. Tanenbaum. A lightweight method for building reliable operating systems despite unreliable device drivers. Technical Report IR-CS-018, Department of CS, Vrije Universiteit, The Netherlands, Jan 2006.

[9] Iguana operating system. URL http://www.ertos.nicta.com.au/iguana/.

[10] B. Leslie, N. FitzRoy-Dale, and G. Heiser. Encapsulated user-level device drivers in the Mungi operating system. In *WS Obj. Syst. & Softw. Arch. 2004*, 2004.

[11] J. LeVasseur, V. Uhlig, J. Stoess, and S. Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *6th OSDI*, Dec 2004.

[12] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *4th OSDI*, 2000.

[13] OMG. UML 2.0 specification. URL http://www.omg.org/technology/documents/formal/uml.htm, 2005.

[14] Y. Padioleau, J. L. Lawall, and G. Muller. Understanding collateral evolution in Linux device drivers. In *EuroSys Conf.*, pages 59–71, Leuven, Belgium, Apr 2006.

[15] M. Swift, B. Bershad, and H. Levy. Improving the reliability of commodity operating systems. In *19th SOSP*, Oct 2003.

[16] F. Zhou, J. Condit, Z. Anderson, I. Bagrak, R. Ennals, M. Harren, G. Necula, and E. Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *7th OSDI*, pages 45–60, 2006.