Formalising Device Driver Interfaces *

Leonid Ryzhyk[‡§] Ihor Kuz[‡§] Gernot Heiser[‡§¶]
[‡]NICTA[†] [§]University of New South Wales [¶]Open Kernel Labs
Sydney, Australia
Leonid.Ryzhyk@nicta.com.au

ABSTRACT

The lack of well-defined protocols for interaction with the operating system is a common source of defects in device drivers. In this paper we investigate the use of a formal language to define these protocols unambiguously. We present a language that allows us to convey all important requirements for driver behaviour in a compact specification and that can be readily understood by software engineers. It is intended to close the communication gap between OS and driver developers and enable more reliable device drivers.

Categories and Subject Descriptors: D.4.4 [Operating systems]: Input/Output; D.3.2 [Programming Languages]: Specialized Application Languages

General Terms: Languages.

1. INTRODUCTION

In order to use advanced features of I/O devices, such as hot-plugging, power management and vectored I/O, operating systems define complex protocols for interaction with device drivers. Failing to comply with a protocol leads to subtle errors that are often manifested in corner cases, when the driver handles an uncommon combination of events, e.g., receiving a hot-unplug notification while processing a power management request. Such errors are particularly difficult to detect during testing and code revision.

We believe that this type of defect is caused in part by the lack of a precise specification of how a correct driver should behave. Typically, OS documentation defines driver interfaces in terms of a set of functions that the driver must implement and a set of callbacks that the driver can invoke. This leaves constraints on ordering, timing and arguments of invocations implicit in the OS implementation. As a result, the driver developer is forced to infer these constraints, which easily leads to errors. Interestingly, as Ball et al. [1] have discovered during an effort to formalise Windows driver API rules, in more complex cases I/O framework developers themselves *would disagree with one another about subtle points in the rules*. Clearly, in the absence of a common agreement on how a correct driver should behave, development of error-free drivers is, at the very least, problematic.

As part of an effort to develop a highly-reliable I/O framework, we are investigating the use of a formal language to specify devive driver protocols. The requirements for such a language are that it (1) is sufficiently expressive to capture important constraints of the protocols, (2) is readily understood by driver developers and (3) provides mechanisms for modular specification of complex protocols. We found that existing software protocol specification languages do not satisfy these requirements. As a result we have developed a new specification language called Driver Protocol State Machines (DPSM). DPSM is a visual language, which uses a subset of the Statecharts [4] syntax and enhances it with several new constructs, namely protocol dependencies, subprotocols and protocol variables. DPSM specifications are readily understandable to programmers and can be used as central design documents that guide the development of both device drivers and the I/O framework itself. The language has well-defined semantics, which enables its use for automated static and runtime analysis of driver behaviour. The semantics are expressed by means of translation from DPSM specifications to terms in the CSP process algebra [5].

The design of DPSM is driven by our experience specifying and implementing real driver interfaces. We introduce a construct to the language only if it has proved necessary to model the behaviour of several types of drivers and can not be easily expressed with other constructs.

Note that DPSM protocols are device-class specific, i.e. a protocol describes common functionality of a class of devices, such as Ethernet controllers or USB hubs, rather than a particular device model. A protocol is defined by the OS designer when support for a new device class is introduced to the OS. Therefore, DPSM cannot be applied directly to legacy systems that have been developed without a rigorous model of driver behaviour in mind. Such systems could gradually move to DPSM-defined protocols as they evolve; existing drivers would continue to use old protocols, while newly-developed drivers would use DPSM protocols. This approach is not uncommon. The recent introduction of the Windows Driver Foundation is one example of an OS gradually switching to a new driver API.

*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

PLOS '07, October 18, 2007, Stevenson, Washington, USA.

In this paper we improve upon our earlier work [9] by developing a complete DPSM syntax and enhancing the language with dynamic constructs, namely protocol variables and subprotocols. In addition, we demonstrate the use of DPSM on a realistic example: a USB hub driver protocol.

2. THE DINGO DRIVER FRAMEWORK

The DPSM language has been developed in tandem with Dingo—a user-level driver framework that we are building for the L4/Iguana [6] embedded OS. A driver in Dingo is represented as an object whose functionality is accessed through ports (Figure 1). A port is a typed bidirectional message-based communication point between the driver and the environment. For instance, a driver for a USB hub has three ports: the lc port of type Lifecycle for messages related to driver lifecycle control, the hub port of type USBHub that exports the hub functionality to the operating system, and the usb port of type USBInterface through which the driver communicates with the USB bus driver.

In Dingo, drivers are single-threaded and non-blocking. Operations that involve waiting for an external event, e.g., a hardware interrupt, are split into two non-blocking phases: request and completion. This design decision is motivated by reliability concerns. Multithreading in driver framework APIs has been shown to constitute a major source of driver bugs [2], while the benefits of multithreaded drivers are minimal, if any, since the performance of most drivers is bounded by device rather than CPU speed[1]. Having said that, we do allow different drivers to be scheduled in different threads and on different CPU cores.

3. DPSM SYNTAX AND SEMANTICS

The type or *protocol* of a port determines which sequences of incoming and outgoing messages are permitted through the port. A protocol specification in DPSM consists of several sections, described below. To make the presentation clear yet concise, we introduce the DPSM syntax using a realistic example of a USB hub driver.

Messages. The messages section defines a protocol's signature—the set of messages that can be exchanged through the port. Message declarations follow the C++ syntax, but without return type and with the addition of a direction specifier. For instance, the Lifecycle protocol that all Dingo drivers must implement has the following signature:

```
protocol Lifecycle {
  messages:
    in  start();
    out startComplete();
    out startFailed(error_t error);
    in  stop();
    out stopComplete();
    in  unplugged(); /*Hot unplug notification*/
    ...
```

Transitions. The transitions section defines protocol states and state transitions. Designers and implementers of the protocol work with the visual representation of the state machine; the textual representation is only intended as input to software tools.

[1] One notable exception involves high-performance network controllers with independent receive and transmit paths. In a single-threaded framework such devices can be managed by two separate drivers in order to exploit intrinsic parallelism.

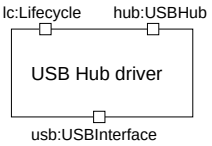

Figure 1: Ports of a USB hub driver.

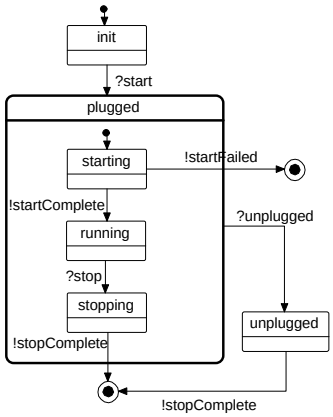

Figure 2: The Lifecycle protocol state machine.

Figure 2 shows the state machine of the Lifecycle protocol. States represent the conceptual states and activities of the driver distinguishable at the protocol level. The syntax of state transition labels is <trigger>[<guard>]/<action>, the [<guard>] and <action> components are optional[2] (Figure 3 shows an example label using all three components). Transitions are triggered by messages sent to and from the driver. Question marks ("?") in trigger names denote incoming messages, exclamation marks ("!") denote outgoing messages. Guards and actions are expressions defined over protocol variables (see below) and message arguments. A transition is taken only if its guard evaluates to true.

The compact representation of complex protocols is achieved by organising states into a hierarchy—a feature borrowed from Statecharts. Several primitive states can be clustered into a super-state. A transition originating from a super-state preempts any of its internal states (e.g., the ?unplugged transition in Figure 2). If the state machine 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.

The protocol state machine is interpreted as follows: any message that triggers a valid state transition complies with the protocol specification. A message that does not trigger any valid transitions violates the protocol specification.

Dependencies. Most drivers implement several ports and thus participate in several protocols. To define the behaviour required of a driver, it is often necessary to specify ordering constraints between messages of different protocols. In DPSM, such constraints are expressed by adding the relevant messages of one protocol to the state machine of another.

For example, a USB hub driver can only send or receive messages through its hub port after the device has been started and before it is stopped or unplugged. Start, stop and unplug events correspond to messages of the Lifecycle protocol. Therefore the USBHub protocol state machine (Figure 4) has to synchronise with lifecycle messages. In

[2] "[" and "]" are literals, not meta characters.

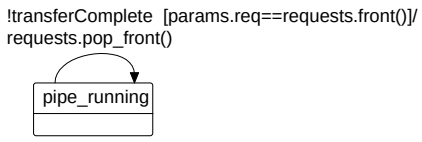

Figure 3: Example of a complex transition label.

the USBHub state diagram, messages that belong to the Lifecycle protocol are prefixed with the protocol name. According to Figure 4, no USBHub messages are allowed before ?Lifecycle::start and after !Lifecycle::stopCompleted or ?Lifecycle::unplugged, which reflects the above requirement.

In order to make dependencies among protocols explicit, the protocol specification lists external messages that the protocol can synchronise with in a separate section. For example the USBHub protocol has the following dependencies:

```
dependencies:
  listens     Lifecycle::start;
  listens     Lifecycle::startComplete;
  listens     Lifecycle::startFailed;
  listens     Lifecycle::unplugged;
  restricts   Lifecycle::stop;
  listens     Lifecycle::stopComplete;
```

This specification implicitly requires any driver implementing a port of type USBHub to also implement a port of type Lifecycle. The restricts and listens keywords describe two types of protocol dependencies. The restricts dependency means that the message is only allowed to be sent if it triggers a state transition in both its main protocol (i.e., the protocol that declares the message in its messages section) and the dependent protocol. The listens dependency means that the dependent protocol may react to the message but does not restrict its possible occurrences.

Orthogonal regions. 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* [4], which make the logical concurrency apparent and reduce the number of explicit states. When the protocol state machine is in a state with several orthogonal regions, each of the regions simultaneously constrains the driver behaviour. On state diagrams, orthogonal regions are separated by dashed lines. For instance, the active state in Figure 4 contains two orthogonal regions: the top region describes the hub overcurrent notification functionality, while the bottom region describes how the driver reports the presence of external power.

Subprotocols. Many driver protocols involve dynamic creation of resources managed by the driver on behalf of the client. This is particularly common for bus drivers. For example, a USB bus driver may create hundreds of USB pipes through which other drivers communicate with devices on the bus. A resource is accessed according to its own protocol. We refer to it as a *subprotocol* of the main protocol. Since the number of dynamically spawned resources is always bounded, the resulting state machine of the protocol is finite. However it would be impractical to specify this state machine without appropriate syntactic support. We therefore introduce the concepts of subprotocols and dynamic subprotocol spawning in DPSM.

Dynamic spawning is expressed by means of the new operator, which appears inside the action part of a transition label and takes the name of a subprotocol and resource identifier as its arguments. For example, during initialisation the hub driver enumerates physical ports of the hub and sends a !portReportFeatures message to the OS for each detected port (see the ports_init state in Figure 4). This message establishes a new logical connection through which the OS will control the physical port. In the protocol state machine, connection establishment is expressed by the "new USBHubPort(params.portNum)" action, where USBHubPort is the name of the physical port subprotocol and params.portNum refers to the portNum argument of the !portReportFeatures message, to be used as resource identifier. All messages that belong to the subprotocol must carry a resource identifier as their first argument. Alternatively a new resource could use a port of its own rather than share the main port of the protocol. In this case the address of the new port is used as a resource identifier and subprotocol messages do not need the extra argument.

The USBHubPort subprotocol (Figure 5) follows the same syntax as normal protocol specification, except that subprotocol transition labels can use the parent keyword to refer to messages of the main protocol. For example, the USBHubPort subprotocol synchronises with power control messages of the USBHub protocol (see the ?parent::hubOvercurrent transition in the right-hand side of Figure 5). Main protocol messages referenced by the subprotocol state machine must be listed among subprotocol dependencies.

The list of subprotocols is part of the protocol specification. For instance, the USBHub protocol declares one subprotocol (the argument in parentheses defines the type and name of the connection identifier):

```
protocols:
  protocol USBHubPort(uint8_t portNum);
```

Protocol variables. Another important feature of protocol state machines is *protocol variables*. Some elements of a protocol are inconvenient to model with explicit states and are more naturally described using variables. Currently, the only permitted variable types are integers (which can also be used to store pointers) and a small number of abstract data types: queues, sets, and stacks. We expect the type system to grow in future. Variables are declared in a separate section inside the protocol specification and can be referenced from protocol transition guards and triggers.

One common use of variables is to store requests handled by the driver. For example, the USB bus driver maintains a queue of data transfer requests for every open pipe. Incoming requests are added to the tail of the queue, completed requests are removed from the head of the queue. The request queue can be declared as follows:

```
variables:
  queue<uintptr_t> requests;
```

The state transition in Figure 3 is extracted from the USB pipe protocol. It specifies that a pipe must complete transfers in the FIFO order by asserting that the request returned by a !transferComplete message must be the same as the one pointed to by the head of the requests queue.

Timeout states. Device driver protocols often involve timing constraints. To capture these constraints, DPSM allows annotating states with 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

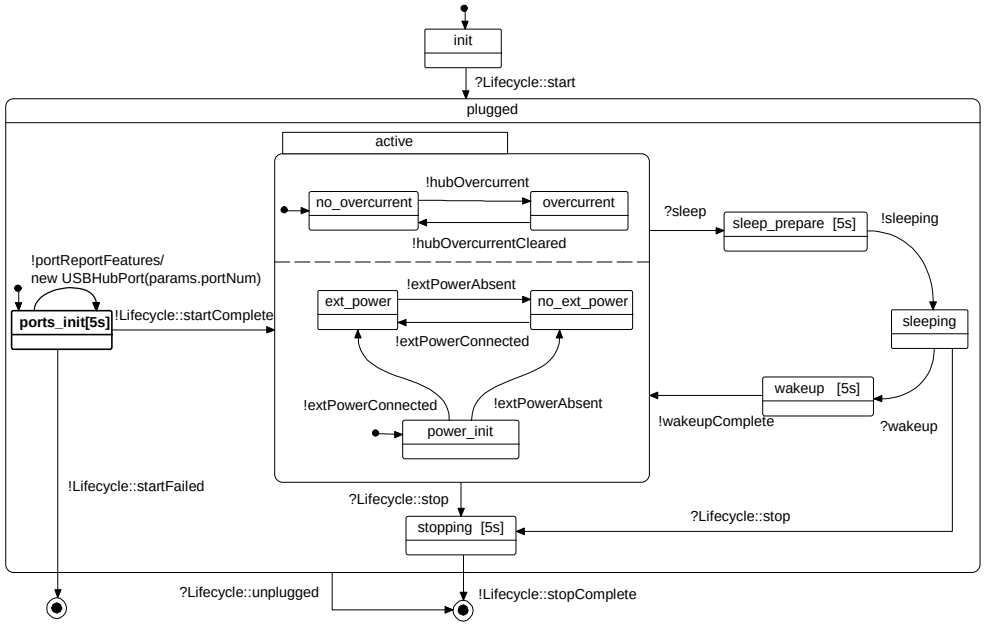

Figure 4: USBHub protocol

triggering of a transition leading to a different state. For example, the power_init state in Figure 4 has a label indicating that the driver should report the status of the external power supply within 5 seconds after completing initialisation.

Semantics. We assign precise meaning to DPSM constructs by defining a translation from protocol specifications to the Communicating Sequential Processes [5] formalism. CSP allows natural interpretation of advanced DPSM features, such as dynamic subprotocol spawning and protocol variables, including infinite-state variables such as queues. The idea of the translation is to construct a CSP process that accepts any sequence of messages permitted by the protocol but deadlocks on illegal sequences. Due to space limitations, we do not describe details of the translation. Given this translation, one could, in principle, specify driver protocols directly in CSP. However, CSP does not provide the right level of abstraction for this task and would not be a convenient tool for both protocol designers and implementers.

4. APPLICATIONS

DPSM encourages the state-machine approach to driver design. The developer defines a set of variables that reflect the protocol state and updates these variables on relevant events, as prescribed by the protocol. By making explicit the set of events that the driver should handle or generate in any state, DPSM simplifies the development and thereby reduces errors. One promising idea that we are exploring is to use Statecharts as an implementation language for drivers. The advantage of this approach is that protocol specifications can be used as a ready-made skeleton for the driver, which needs to be filled with device-specific operations.

The well-defined semantics of DPSM enables its use beyond documentation. First, DPSM specifications can be compiled into executable components that are interposed between the driver and the OS to detect runtime protocol violations. This forms the basis of the failure recovery infrastructure that we are building into Dingo.

Second, we are investigating the possibility of statically checking a driver for protocol compliance. Given a CSP representation of a protocol, we can transform it into an abstract model of driver behaviour. More precisely, we construct a new CSP process that deterministically accepts any messages that a real driver is required to accept and nondeterministically sends any messages that the driver is allowed to send. Any correct driver should be a refinement of this abstract model. Thus, the task of protocol compliance verification is reduced to a well-defined task of CSP refinement checking. A lot of work still remains to be done as the most complicated part of refinement checking is extraction of relevant control flow information from the driver source code. Feasibility of such source code analysis has been demonstrated by Ball et al. [1].

Other applications being considered include formal verification of the I/O framework implementation itself and construction of provably-correct driver recovery protocols.

5. RELATED WORK

The software engineering community has developed a number of software protocol specification languages, most noticeably UML Protocol State Machines (PSM) [8]. Like DPSM, PSM is based on the Statecharts visual syntax, which allows modelling complex protocols using hierarchy and concurrency and makes PSM specifications easy to understand. However, neither PSM nor other languages we have investigated allow the expression of protocol dependencies, protocol variables and subprotocols.

In the operating systems community, the SLAM [1] and Singularity [3] projects have used state-machine based formalisms to specify driver interfaces. Both formalisms provide a form of subprotocol spawning but do not support protocol dependencies nor infinite-state variables, such as sets and queues (Singularity does not support protocol variables at all). As a result, they only partially capture the requirements of driver protocols. Besides this, neither SLAM nor Singularity provide a means for structuring complex protocols and do not aim to make protocol specifications easy to

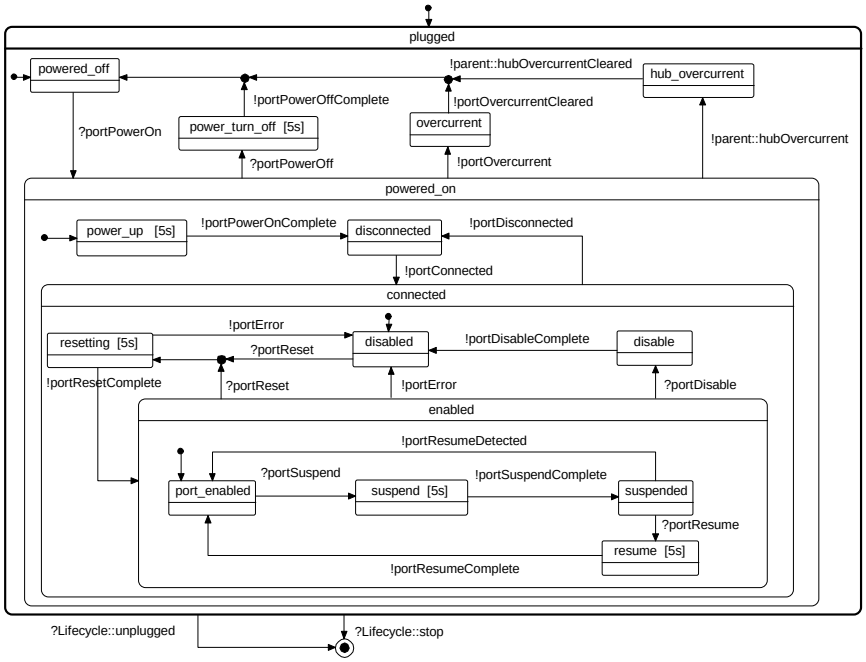

Figure 5: USBHubPort sub-protocol

understand and work with. The reason for these limitations is that SLAM and Singularity use formal protocol specifications solely for static-analysis purposes. Properties that cannot be statically verified, e.g., those involving infinite-state variables, are excluded from the formalism. Clarity and manageability are not among addressed issues. In contrast, we separate specification from verification. The primary purpose of DPSM specifications is to serve as guidelines for driver developers. As mentioned in the previous section, they can also be used as properties against which driver implementations can be statically verified.

The Devil [7] project has developed a language for describing the functional interface of a device in terms of its memory and register layouts, for use by device drivers. This is complementary to DPSM, which is intended for specifying the software interface of a driver.

6. EXPERIENCE AND CONCLUSIONS

To date, we have specified a number of protocols in DPSM, including protocols for Ethernet controller drivers, USB bus and hub drivers, interrupt controller drivers and others. We found the language to be sufficiently expressive to capture all the requirements for driver behaviour that we identified. Moreover, we found none of the language features to be redundant. In particular, protocol dependencies, protocol variables, and subprotocols are essential to modelling driver behaviour and cannot be expressed with other constructs.

DPSM specifications tend to be highly compact. Quite complex protocols fit into two or three screen-size diagrams and tens of lines of textual specification. For instance, the USB hub protocol is completely specified by diagrams in Figures 4 and 5, and 50 lines of associated text[3].

We have implemented each of the protocols in at least one driver and found DPSM specifications to help greatly with driver development, making the otherwise intricate driver control logic straightforward and helping to avoid errors. We also tested runtime failure detection on the example of the RTL8139 Ethernet driver and 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.

Development of a protocol for a new family of drivers is a difficult task that proceeds in many iterations and requires a deep understanding of relevant hardware specifications as well as driver and I/O framework design issues. The use of DPSM ensures that the result of this effort is not lost in the bowels of the OS code but is preserved as a structured specification that conveys a great deal of knowledge about driver behaviour in a compact form and can be readily understood by software engineers. Thus we see the primary role of the DPSM language in closing the communication gap between I/O framework and driver developers and providing a formal basis for the construction of reliable device drivers.

[3]The diagrams have been only slightly simplified to fit into the paper.

7. 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] 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, Oct 2001.

[3] 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.

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

[5] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[6] Iguana OS. URL `http://www.ertos.nicta.com.au/iguana/`.

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

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

[9] L. Ryzhyk, T. Bourke, and I. Kuz. Reliable device drivers require well-defined protocols. In *3rd HotDep*, Jun 2007.