On the Construction of Reliable Device Drivers

Leonid Ryzhyk

Ph.D.

2009

UNSW
THE UNIVERSITY OF NEW SOUTH WALES
SYDNEY • AUSTRALIA

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation, and linguistic expression is acknowledged.'

Signed ....

Date ....

Abstract

This dissertation is dedicated to the problem of device driver reliability. Software defects in device drivers constitute the biggest source of failure in operating systems, causing significant damage through downtime and data loss. Previous research on driver reliability has concentrated on detecting and mitigating defects in existing drivers using static analysis or runtime isolation. In contrast, this dissertation presents an approach to reducing the number of defects through an improved device driver architecture and development process.

In analysing factors that contribute to driver complexity and induce errors, I show that a large proportion of errors are due to two key shortcomings in the device-driver architecture enforced by current operating systems: poorly-defined communication protocols between drivers and the operating system, which confuse developers and lead to protocol violations, and a multithreaded model of computation, which leads to numerous race conditions and deadlocks. To address the first shortcoming, I propose to describe driver protocols using a formal, state-machine based, language, which avoids confusion and ambiguity and helps driver writers implement correct behaviour. The second issue is addressed by abandoning multithreading in drivers in favour of a more disciplined event-driven model of computation, which eliminates most concurrency-related faults. These improvements reduce the number of defects without radically changing the way drivers are developed.

In order to further reduce the impact of human error on driver reliability, I propose to automate the driver development process by synthesising the implementation of a driver from the combination of three formal specifications: a device-class specification that describes common properties of a class of similar devices, a device specification that describes a concrete representative of the class, and an operating system interface specification that describes the communication protocol between the driver and the operating system. This approach allows those with the most appropriate skills and knowledge to develop specifications: device specifications are developed by device manufacturers, operating system specifications by the operating system designers. The device-class specification is the only one that requires understanding of both hardware and software-related issues. However writing such a specification is a one-off task that only needs to be completed once for a class of devices.

This approach also facilitates the reuse of specifications: a single operating-system specification can be combined with many device specifications to synthesise drivers for

multiple devices. Likewise, since device specifications are independent of any operating system, drivers for different systems can be synthesised from a single device specification. As a result, the likelihood of errors due to incorrect specifications is reduced because these specifications are shared by many drivers.

I demonstrate that the proposed techniques can be incorporated into existing operating systems without sacrificing performance or functionality by presenting their implementation in Linux. This implementation allows drivers developed using these techniques to coexist with conventional Linux drivers, providing a gradual migration path to more reliable drivers.

Acknowledgements

I am grateful to my supervisors Gernot Heiser and Ihor Kuz for their guidance throughout the project. Working with them, I enjoyed a lot of freedom in choosing the research direction and exploring various ideas by way of trial and error, while getting advice, feedback, and support when I needed them.

Throughout the project, several people have helped develop the ideas presented in this work. In particular, collaboration with Timothy Bourke on modelling device drivers in Esterel inspired the design of the Tingu driver protocol specification language. Rob van Glabbeek's lectures on process calculus were a major influence on the design of the Termite driver synthesis tool. Introduction to the theory of two-player games by Franck Cassez was instrumental in developing the Termite synthesis algorithm.

I want to thank Peter Chubb and Etienne Le Sueur for their collaboration in implementing and evaluating the Dingo driver framework. I want to thank Balachandra Mirla for his input in the SD controller driver synthesis case study. I want to thank John Keys for his help in implementing the Termite compiler.

Finally, I want to thank all my colleagues at ERTOS whose knowledge, team spirit, and sense of humour have helped reduce the inevitable stress of being a graduate student.

Related Publications

[1] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with Termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, October 2009.

[2] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th EuroSys Conference*, Nuremberg, Germany, April 2009.

[3] Leonid Ryzhyk, Ihor Kuz, and Gernot Heiser. Formalising device driver interfaces. In *Proceedings of the 4th Workshop on Programming Languages and Operating Systems*, Stevenson, Washington, USA, October 2007.

[4] Leonid Ryzhyk, Timothy Bourke, and Ihor Kuz. Reliable device drivers require well-defined protocols. In *Proceedings of the 3rd Workshop on Hot Topics in System Dependability*, Edinburgh, UK, June 2007.

Contents

1 Introduction **1**
1.1 Improving OS support for device drivers with Dingo 4
1.2 Automatic device driver synthesis with Termite 4
1.3 Contributions 6
1.4 Chapter outline 6

2 Background **7**
2.1 The history of device drivers 8
2.2 I/O hardware organisation in modern computer systems 9
2.2.1 Peripheral Component Interconnect bus 10
2.2.2 Universal Serial Bus 13
2.3 Device drivers in modern operating systems 15
2.4 Generic OS services 17
2.4.1 Memory management 17
2.4.2 Timers 20
2.5 Concurrency and synchronisation in device drivers 20
2.5.1 The Mac OS X workloop architecture 21
2.5.2 Windows Driver Foundation synchronisation scopes 21
2.6 Hot plugging 22
2.7 Power management 22
2.8 A taxonomy of driver failures 23

3 Related work **27**
3.1 Hardware-based fault tolerance 28
3.1.1 Drivers in capability-based computer systems 29
3.1.2 User-level device drivers in microkernel-based systems 29
3.1.3 Device driver isolation in monolithic OSs 34
3.1.4 User-level device drivers in paravirtualised systems 36
3.2 Software-based fault isolation 37
3.3 Fault removal using static analysis 39
3.4 Language-based fault prevention and fault tolerance 42

3.5 Preventing and tolerating device protocol violations 46
3.6 Fault prevention through automatic device driver synthesis 47
3.7 Conclusions 48

4 Root-cause analysis of driver defects 49
4.1 Methodology 50
4.2 Example 51
4.3 Defects caused by the complexity of device protocols 53
4.4 Defects caused by the complexity of *operating system (*OS*)* protocols . . 55
4.5 Concurrency defects 58
4.6 Generic programming faults 60
4.7 Limitations of the study 60
4.8 Conclusions 61

5 Device driver architecture for improved reliability 63
5.1 Overview of Dingo 64
5.2 An event-based architecture for drivers 65
5.2.1 C with events 69
5.2.2 Implementation of C with events 70
5.2.3 Dingo on Linux 71
5.2.4 Selectively reintroducing multithreading 72
5.2.5 Comparison with existing architectures 75
5.3 Tingu: describing driver software protocols 75
5.3.1 Introduction to Tingu by example 77
5.3.2 Discussion 86
5.3.3 Detecting protocol violations at runtime 86
5.3.4 From protocols to implementation 87
5.4 Evaluation 90
5.4.1 Code complexity 91
5.4.2 Reliability 91
5.4.3 Performance 93
5.5 Conclusions 98

6 Automatic device driver synthesis with Termite 99
6.1 Motivation 99
6.2 Overview of driver synthesis 101
6.2.1 Device-class specifications 102
6.2.2 Device specifications 104
6.2.3 OS specifications 106
6.2.4 The synthesis process 107
6.3 A trivial example 108

6.4 The Termite specification language . . . 110
6.4.1 Restrictions on device-class specifications . . . 115
6.5 A realistic example . . . 116
6.5.1 Overview . . . 116
6.5.2 The device-class specification . . . 118
6.5.3 The OS protocol specification . . . 118
6.5.4 The device protocol specification . . . 123
6.5.5 The driver state machine . . . 130
6.6 The synthesis algorithm . . . 131
6.6.1 Symbolic representation of protocol state machines . . . 131
6.6.2 Computing the product state machine . . . 133
6.6.3 Computing the strategy . . . 134
6.6.4 Computing the strategy symbolically . . . 142
6.6.5 Generating code . . . 143
6.7 Debugging synthesised drivers . . . 143
6.8 Evaluation . . . 145
6.8.1 Synthesising drivers for real devices . . . 145
6.8.2 Performance . . . 148
6.8.3 Reusing device specifications . . . 149
6.9 Limitations and future work . . . 150
6.10 Conclusions . . . 151

7 Conclusions 153

A The syntax of Tingu and Termite 155
A.1 Component, protocol, and type declarations . . . 155
A.1.1 Common definitions . . . 155
A.1.2 Types . . . 157
A.1.3 Protocols . . . 157
A.1.4 Components . . . 160
A.2 Protocol state transition labels . . . 161
A.3 Protocol state machines . . . 163
A.4 Termite processes . . . 167

B Tingu protocol specification examples 169
B.1 The `Lifecycle` protocol . . . 169
B.1.1 `Lifecycle` methods . . . 169
B.2 The `PowerManagement` protocol . . . 171
B.2.1 `PowerManagement` methods . . . 171
B.3 The `EthernetController` protocol . . . 174
B.3.1 `EthernetController` methods . . . 174

B.3.2 The EthernetController protocol state machine 176
B.4 The USBInterfaceClient protocol 179
B.4.1 USBInterfaceClient methods 179
B.4.2 USBPipeClient methods 179
B.4.3 The USBPipeClient protocol state machine 181
B.5 The InfiniBandController protocol 185
B.5.1 InfiniBandController methods 187
B.5.2 IBPort methods 190
B.5.3 IBProtectionDomain methods 193
B.5.4 IBQueuePair methods 196
B.5.5 IBSharedRQ methods 199
B.5.6 IBCompletionQueue methods 202

C The OpenCores SD controller device specification 205

List of Figures

1.1 A generalised device-driver architecture. 3

2.1 I/O bus hierarchy of a typical desktop system. 10
2.2 Architectural patterns used in the design of operating system I/O frameworks. 15
2.3 Superposition of Driver and Bus patterns. 17
2.4 Device driver stacking. 18
2.5 A refined device-driver architecture, including the generic services interface. 19

4.1 Linux USB storage driver interfaces. 53
4.2 A defect in the rtl8150 controller driver. 56
4.3 Relative frequency of the four categories of driver defects. 62
4.4 Summary of defects by bus. 62

5.1 Dingo driver for the ax88772 USB-to-Ethernet adapter and its ports. . . . 64
5.2 The stack ripping problem in event-based drivers. 67
5.3 Implementation of the probe method using the Dingo preprocessor. 68
5.4 Dingo interface adapters using the example of the ax88772 controller driver. 72
5.5 Handling of synchronous and asynchronous requests sent by the Linux kernel to a Dingo driver. 73
5.6 Handling of requests sent by a Dingo driver to the Linux kernel. 74
5.7 The use of Tingu protocol specifications. 77
5.8 Tingu declaration of the ax88772 driver component. 78
5.9 The `Lifecycle` protocol declaration. 79
5.10 The `Lifecycle` protocol state machine. 80
5.11 The `PowerManagement` protocol declaration. 81
5.12 The `PowerManagement` protocol state machine. 82
5.13 A fragment of the `USBInterfaceClient` protocol declaration. 84
5.14 The `USBInterfaceClient` protocol state machine. 85
5.15 A fragment of the `USBPipeClient` protocol declaration. 85
5.16 A fragment of the `USBPipeClient` protocol state machine. 85
5.17 A fragment of the `EthernetController` protocol state machine. . . 88
5.18 A fragment of the ax88772 driver. 89

5.19 ax88772 UDP latency results. 94
5.20 ax88772 UDP throughput results. 95
5.21 InfiniHost UDP latency benchmark results. 96
5.22 InfiniHost TCP throughput benchmark results. 97

6.1 Driver synthesis with Termite. 103
6.2 Specification of the transmit operation of the RTL8139D controller derived from its data sheet. 105
6.3 Specification of the transmit operation of the RTL8139D controller derived from a reference driver implementation. 105
6.4 A fragment of the Ethernet controller driver protocol specification. 107
6.5 Specification of a trivial network controller driver. 109
6.6 Product state machine representing combined constraints of the two driver protocols. 110
6.7 Synthesised driver algorithm. 110
6.8 A simple Termite process and the corresponding state machine. 112
6.9 SD host controller device. 116
6.10 SD host controller driver and its ports. 117
6.11 The SD host controller driver component specification. 117
6.12 The SD host controller device-class specification. 119
6.13 The SD host controller driver OS protocol specification. 120
6.13 The SD host controller driver OS protocol specification *(continued)*. . . . 121
6.13 The SD host controller driver OS protocol specification *(the end)*. 122
6.14 The OpenCores SD host controller device architecture. 124
6.15 The OpenCores SD host controller device specification. 127
6.15 The OpenCores SD host controller device specification. 128
6.15 The OpenCores SD host controller device specification *(continued)*. . . . 129
6.16 A fragment of the SD host controller driver state machine generated by Termite. 130
6.17 Helper functions used in computing the driver strategy. 137
6.18 The main Termite algorithm for computing the driver strategy. 138
6.19 The **AddControllableTransition** procedure. 139
6.20 The **AddUncontrollableTransition** procedure. 139
6.21 Example of a reachability tree constructed by the **MoveToGoal** procedure 140
6.22 The **MoveToGoal** procedure. 141
6.23 Termite debugger screenshot. 146
6.24 ax88772 TCP throughput benchmark results. 149

A.1 Example of a simple statechart. 163
A.2 Example of a statechart with OR-superstates. 164

A.3 Example of state collapsing. 165
A.4 Example of a statechart with a history pseudo-state. 166
A.5 Example of a statechart with an AND-superstate. 166

B.1 The `Lifecycle` protocol declaration. 170
B.2 The `Lifecycle` protocol state machine. 170
B.3 The `PowerManagement` protocol declaration. 172
B.4 The `PowerManagement` protocol state machine. 173
B.5 The `EthernetController` protocol declaration. 175
B.6 The top-level `EthernetController` protocol state machine. 177
B.7 The `link_status` state of the `EthernetController` protocol state machine expanded. 177
B.8 The `properties` state of the `EthernetController` protocol state machine expanded. 178
B.9 The `tx_rx` state of the `EthernetController` protocol state machine expanded. 178
B.10 The `USBInterfaceClient` protocol declaration. 180
B.11 The `USBInterfaceClient` protocol state machine. 181
B.12 The `USBPipeClient` protocol declaration. 182
B.13 The `USBPipeClient` protocol state machine. 184
B.14 Types of objects exported by an InfiniBand controller driver to the OS. . . 186
B.15 The `InfiniBandController` protocol declaration. 188
B.16 The `InfiniBandController` protocol state machine. 189
B.17 The `BPort` protocol declaration. 191
B.18 The `IBPort` protocol state machine. 192
B.19 The `IBProtectionDomain` protocol declaration. 194
B.20 The `IBProtectionDomain` protocol state machine. 195
B.21 The `IBQueuePair` protocol declaration. 197
B.22 The `IBQueuePair` protocol state machine. 198
B.23 The `IBSharedRQ` protocol declaration. 200
B.24 The `IBSharedRQ` protocol state machine. 201
B.25 The `IBCompletionQueue` protocol declaration. 203
B.26 The `IBCompletionQueue` protocol state machine. 203

C.1 The OpenCores SD host controller device specification. 206
C.1 The OpenCores SD host controller device specification *(continued)*. . . . 207
C.1 The OpenCores SD host controller device specification *(continued)*. . . . 208
C.1 The OpenCores SD host controller device specification *(continued)*. . . . 209
C.1 The OpenCores SD host controller device specification *(continued)*. . . . 210
C.1 The OpenCores SD host controller device specification *(continued)*. . . . 211

C.1 The OpenCores SD host controller device specification *(continued)*. . . . 212
C.1 The OpenCores SD host controller device specification *(the end)*. 213

Acronyms

ADT abstract data type

API application programming interface

ASIC application-specific integrated circuit

BDD binary decision diagram

BIOS Basic Input/Output System

BNF Backus-Naur Form

CPU central processing unit

DNF Disjunctive Normal Form

DMA direct memory access

EHCI Enhanced Host Controller Interface

FPGA field-programmable gate array

FSB Front-Side Bus

FSM finite state machine

HDL hardware description language

IOMMU input/output memory management unit

I/O input/output

IPC inter-process communication

ISA instruction set architecture

LOC lines of code

MMU memory management unit

MSI message signaled interrupts

OHCI Open Host Controller Interface

OS operating system

PCI Peripheral Component Interconnect

PCIe PCI Express

RCA root-cause analysis

RDMA remote direct memory access

RTL register-transfer level

SATA Serial AT Attachment

SCSI Small Computer System Interface

SD Secure Digital

SFI software-based fault isolation

SMT satisfiability modulo theories

TLB translation lookaside buffer

UDI Uniform Driver Interface

UHCI Universal Host Controller Interface

UML Unified Modeling Language

USB Universal Serial Bus

VM virtual machine

VMM virtual machine monitor

Chapter 1

Introduction

This dissertation is dedicated to the problem of device driver reliability. According to recent studies [GGP06, Mur04], software faults in device drivers are responsible for 70% of *operating system (OS)* failures, making drivers the leading source of instability in modern computer systems.

Several factors contribute to this situation. First, device drivers are an integral part of the system software stack, providing critical services to other system components (e.g., network stacks and file systems), as well as to user-level applications. As a result, a failure of a device driver can trigger a system-wide failure, potentially causing downtime and data loss. This is true for both in-kernel drivers and, to a lesser degree, for drivers that execute as user-level processes.

Second, the quality of driver code is inferior to that of other OS components. Writing a device driver requires profound understanding of both device and OS internals. In practice, driver developers are usually experts in at most one of the two areas and are likely to introduce errors when dealing with the less familiar part of the driver functionality.

In addition, drivers do not get tested as thoroughly as the rest of the OS. Many corner cases that may trigger driver defects arise from interleavings of hardware and software events that are difficult to reproduce deterministically during testing. Another problem is that many drivers are intended to support a range of similar devices from a single or multiple vendors; however it is often impractical to test the driver with all supported hardware.

As a result, the density of errors in driver code is an order of magnitude higher than in the core parts of the system, e.g., the scheduler or the virtual memory manager [CYC+01].

Finally, drivers account for a large fraction of OS code and therefore have a strong effect on OS reliability. For example, the total size of device drivers shipped with the Linux 2.6.27 kernel is 3,448,000 *lines of code (LOC)*, which constitutes 69% of the entire kernel tree, including all supported file systems, network protocols, and x86-specific code.[1] Not all drivers run simultaneously on a real system. As an example of a representative

[1] These measurements reflect lines of code, excluding comments, measured using David A. Wheeler's `SLOCCount` tool [Whe] run with default parameters.

Driver	**LOC**
ACPI drivers	
ACPI processor driver	3804
ACPI fan driver	268
ACPI battery driver	765
ACPI smart battery system driver	889
ACPI AC adapter driver	292
ACPI button driver	428
ACPI PCI slot driver	245
ACPI system management bus driver	293
ACPI video driver	1755
ACPI PATA controller driver	179
ACPI dock station driver	558
ACPI thermal zone driver	1414
Interconnect drivers	
PCI bus driver	8464
Parallel port driver	4730
Secure Digital host controller driver	1900
AGP controller driver	4137
PCI hotplug controller driver	2322
Intel PATA/SATA controller driver	1028
IEEE-1394 OHCI controller driver	2753
IEEE-1394 SBP-2 protocol driver	1746
Sonics Silicon Backplane driver	1134
USB EHCI controller driver	720
USB UHCI controller driver	851

Driver	**LOC**
Storage	
Generic CDROM driver	2483
SCSI CDROM driver	1337
SCSI disk driver	1833
Sound	
Intel HD audio controller driver	1749
PC speaker driver	103
Video	
Pixart PAC207BCA USB webcam driver	2031
Intel 965 Express graphics card driver	2409
Networking	
Intel PRO/Wireless 3945ABG adapter driver	9077
Broadcom 44xx/47xx Ethernet controller driver	2169
Human interface devices	
Synaptics mousepad driver	1760
USB HID driver	4450
Other	
Intel TCP watchdog timer driver	734
Generic parallel printer driver	712
Total	**71522**

Table 1.1: Device drivers running on a typical Linux laptop.

configuration, Table 1.1 lists drivers running on the Linux laptop used to typeset this thesis. The list includes 36 different drivers, comprised of 71,522 lines of C code, which constitutes approximately 30% of the entire code running in the kernel on this system. Given the much higher density of errors in drivers compared to the rest of the kernel, this explains why OS failure statistics is dominated by device drivers.

Much of the previous research on device driver reliability has focused on developing runtime isolation and recovery techniques for drivers. The idea of this approach is to place device drivers inside hardware or software-enforced protection boundaries, making sure that a faulty driver cannot overwrite memory used by other parts of the OS. This forms the basis for a failure detection and recovery infrastructure responsible for detecting misbehaving drivers and preventing failures from propagating throughout the system.

Another common approach consists of detecting errors statically, by analysing the code of the driver. This approach is enabled by recent advances in static analysis and model checking, which made these techniques applicable to large programs written in a low-level

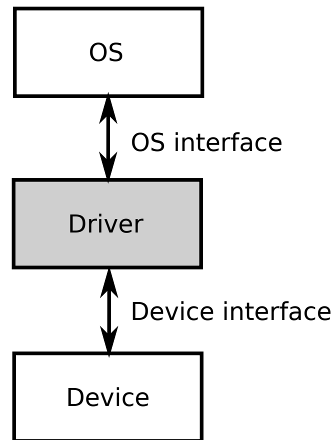


Figure 1.1: A generalised device-driver architecture.

language.

As will be shown in the related work survey in Chapter 3, while offering considerable improvements, both approaches suffer serious limitations. In particular, runtime isolation is associated with substantial performance overhead. More importantly, in order to make a runtime failure transparent to the system, one must have a stateful driver recovery mechanism in place. Such a mechanism must intercept all interactions between the driver and the OS and keep track of data needed for transparent recovery. It turned out that complex and poorly defined driver interfaces in current OSs make it hard to implement this in practice. Static techniques do not involve any runtime overhead; however, despite many improvements in the area, they are still only capable of finding a limited subset of driver bugs.

Given that these existing approaches do not fully neutralise the effect of driver errors, I claim that a complementary solution that enables the creation of drivers with fewer errors has the potential to greatly improve the device driver and hence the overall system reliability.

To this end, this dissertation first analyses the root causes leading to driver errors, with a view of identifying those of them that can be overcome with the help of an improved device driver architecture or development process. Based on a study of a large sample of real drivers defects, described in Chapter 4, I conclude that the majority of these defects are related to handling the device and the OS interfaces of the driver (Figure 1.1). Both interfaces tend to be complex and poorly defined, which confuses driver developers and induces errors. The situation is exacerbated by the common device driver organisation where interactions with the device and the OS are tightly intertwined, forcing the driver developer to deal with the complexity of both interfaces simultaneously and leading to poor separation of concerns inside the driver.

1.1 Improving OS support for device drivers with Dingo

These findings point to potential areas of improvement. In particular, focusing on the OS interface of the driver, I identify two key shortcomings of this interface, as it is defined in current systems. First, most operating systems enforce a multithreaded model of computation on device drivers. In this model, the driver must handle invocations from multiple concurrent threads, which puts the burden of synchronisation on driver developers and leads to numerous race conditions and deadlocks.

The second problem is the lack of well-defined communication protocols between drivers and the OS. Modern OSs impose complex constraints on the ordering and content of interactions with device drivers. These constraints are implicitly defined in the source code of the system and are not captured in documentation, which generally focuses on describing individual driver and OS entry points, while ignoring their possible orderings. This confuses developers and leads to protocol violations. Moreover, as the OS evolves, these constraints may change in subtle ways, often breaking previously correct drivers.

In order to address both shortcomings, I propose an improved device-driver architecture, called Dingo[2]. In particular, I suggest abandoning multithreading in device drivers in favour of a more disciplined event-based model of computation, which eliminates most concurrency-related issues. In order to reduce protocol violations, I propose to describe driver protocols using a visual, state-machine based, language, called Tingu[3]. Tingu specifications serve as documentation, providing easy-to-use guidelines to driver developers, thus avoiding confusion and ambiguity and helping the developers implement correct behaviour.

In order to demonstrate that these improvements can be incorporated in existing OSs without sacrificing performance or functionality, I present a Linux-based implementation, which provides a set of wrappers that make drivers developed in compliance with the Dingo interface appear as regular Linux drivers to the rest of the kernel. This enables Dingo and conventional Linux drivers to coexist, providing a gradual migration path to more reliable drivers. Experimental evaluation of the Dingo driver architecture shows that it eliminates most synchronisation errors and reduces the likelihood of protocol violations, while introducing negligible performance overhead.

1.2 Automatic device driver synthesis with Termite

Further reduction in the number of errors can be achieved with the help of an improved driver development process. The task of writing a device driver consists of defining a mapping from OS requests into sequences of device commands that satisfy these requests. To do so, the driver developer relies on two sets of documentation: a specification of the device interface, which describes how device functions can be controlled from software, and a

[2]The Dingo is Australia's wild dog.

[3]Tingu is an Australian aboriginal name for a Dingo cub.

specification of the OS interface, which describes the service that the OS expects the driver to implement, as well as OS services available to the driver. Given these two specifications, the developer derives a driver algorithm that translates any valid sequence of OS requests into a matching sequence of device operations—a straightforward, yet error-prone task.

In this thesis I demonstrate that this task can be automated. I develop a tool, called Termite, that synthesises a driver implementation automatically based on three formal specifications: a device-class specification that describes common properties of a class of similar devices (e.g., Ethernet controllers), a device specification that describes a concrete representative of the class, and an OS interface specification that describes the communication protocol between the driver and the OS.

Separating the device description from OS-related details is a key aspect of the proposed approach to driver synthesis. It allows those with the most appropriate skills and knowledge to develop specifications: device interface specifications are developed by device manufacturers, OS interface specifications—by the OS developers who have intimate knowledge of the OS and the driver support it provides. The device-class specification is the only one that requires understanding of both hardware and software-related issues. However writing such a specification is a one-off task that only needs to be completed once for a class of devices.

The separation of specifications also facilitates their reuse. The OS specification need only be developed once for each OS and each device class. It is then combined with many concrete device specifications to synthesise drivers for these devices. As a result, the likelihood of errors due to incorrect OS interface specifications is further reduced because these specifications are shared by many drivers. Likewise, since device specifications are independent of any specific OS, drivers for different OSs can be synthesised from a single specification.

With Termite, the problem of writing a correct driver is reduced to that of obtaining correct specifications. The practical utility of this approach is therefore subject to cooperation from device and OS manufacturers, who are in the best position to develop the respective specifications. For device manufacturers, the driver synthesis approach allows for a reduction in driver development effort and an increase in driver quality. Furthermore, once developed, a driver specification will allow drivers to be synthesised for any supported OS, increasing the compatibility of the device. For OS developers the quality and reputation of their OS depends greatly on the quality of its device drivers: major OS vendors suffer serious financial and image damage because of faulty drivers. Driver quality can be improved by providing and encouraging the use of tools for automatic driver synthesis as part of driver development toolkits.

The key components of the driver synthesis methodology are the specification language used to describe device and OS interfaces and the synthesis algorithm that processes these specifications and generates the driver implementation. The Termite specification language is a dialect of the Tingu language. It shares most concepts with Tingu, but uses a textual rather than visual syntax.

The Termite synthesis algorithm is based on game theory. The driver synthesis problem is formalised as a two-player game between the driver and its environment, consisting of the device and the OS. Rules of the game define constraints on legal sequences of interaction between the players and are given by the interface specifications. The objective of the game is to ensure that the driver will satisfy all OS requests in any well-behaved environment.

I evaluate Termite by using it to synthesise drivers for two real devices: an Ethernet controller and a *Secure Digital (SD)* host controller. I demonstrate that the performance of the synthesised drivers is virtually identical to the performance of their hand-written counterparts.

1.3 Contributions

This dissertation makes three main contributions. First, it performs root-cause analysis of device driver defects and identifies the complexity of the device and the OS interfaces of the driver as the key factors that provoke the majority of errors.

Second, it develops a new device driver architecture aimed at reducing errors related to the interaction between the driver and the OS. This is achieved by replacing the multi-threaded model of computation with a more disciplined event-based model and by using a formal visual language to specify the driver-OS interface clearly and unambiguously.

Third, this dissertation proposes a new method of driver construction, which consists of automatically generating the implementation of the driver based on a formal model of its device and OS interfaces. This approach has the potential to dramatically reduce driver development effort while increasing driver quality.

The proposed techniques are evaluated in the context of the Linux kernel; however the results of this work are applicable to other OSs and to both in-kernel and user-level drivers.

1.4 Chapter outline

The rest of this dissertation is structured as follows. Chapter 2 provides background information about device drivers and driver development. Chapter 3 surveys previous research on device driver reliability. Chapter 4 carries out root-cause analysis of driver defects by analysing a sample of real defects found in Linux device drivers. New approaches to improving the driver reliability are presented in Chapters 5 and 6, which describe the Dingo driver architecture and the Termite driver synthesis methodology respectively. Chapter 7 draws conclusions.

Chapter 2

Background

This chapter introduces key notions related to device drivers and I/O programming.

A device driver is the part of the OS that is responsible for controlling an *input/output (I/O)* device. In collaboration with other OS subsystems, it fulfils the following functions:

- *Abstraction.* The driver hides the complexity of the low-level device protocol from its clients, allowing them to use the device through a set of high-level operations. For example, the low-level device protocol for sending a network packet through a network controller device may involve creating a packet descriptor in memory, writing the location of the descriptor to a device register, writing another device register to trigger the transfer, and then waiting for a transfer completion signal from the device. All of these operations occur inside the driver. A client of the driver sends a packet simply by calling the `send()` function of the driver.

- *Unification.* By providing a unified interface to a class of similar devices, drivers hide the differences between the devices from their clients. For example, another network controller may support a different packet descriptor format and implement its own set of registers. However, since the drivers for both controllers implement the same interface, their clients remain unaware of these distinctions. In some cases, unification requires the driver to perform extensive data processing in order to abstract different levels of hardware implementation. For example, while conventional dial-up modems perform all signal processing in hardware, software modems delegate most of the modulation functions to the driver.

- *Protection.* Access to an I/O device is a sensitive operation, subject to the OS access control policy and to physical constraints of the device. The OS enforces the access control policy by making sure that only authorised applications can use the driver.

- *Multiplexing.* The device driver cooperates with the OS in order to enable multiple applications to access the device concurrently. For the most part, multiplexing is performed by the OS outside the driver. For instance, in case of a network controller driver, the OS queues packets obtained from multiple clients and delivers them to the

driver one by one. Some types of drivers, however, maintain per-client contexts and distinguish between requests from multiple clients. For example, InfiniBand [Inf08] controller drivers use this approach to achieve traffic isolation between clients.

2.1 The history of device drivers

The early predecessors of modern device drivers were I/O library routines introduced in the days of batch processing systems for mainframe computers, such as the IBM 709 (1958). The primary motivation for the use of these routines was protection: a user program was expected to access its data on the tape via I/O routines, rather than directly, to prevent corruption of data belonging to the OS or to other jobs in the batch [Ros69]. In the absence of hardware enforcement mechanisms, this facility only guarded against accidental rather than malicious damage.

The reliance on standard software components to perform I/O increased with the introduction of more advanced I/O architectures in later computers, e.g. IBM 7090 (1960). The most prominent innovation was the use of I/O channels and interrupts, which enabled I/O and computation to overlap, the concept currently known as asynchronous I/O. While offering performance benefits, asynchronous I/O was much harder to program than synchronous I/O. It required buffering data that arrived from the device while the program was executing. In addition, since at the time computers did not support multitasking, it also required programming non-trivial interrupt logic, where a reentrant interrupt handler routine processed interrupts from multiple independent sources. According to Rosin [Ros69],

> the complex routines were required to allow even the simplest user program to take full advantage of the hardware, but writing them was beyond the capability of the majority of programmers. This necessitated a set of standard interrupt processing and I/O request programs for use by all programs to be run in the system.

Emergence of new types of I/O devices, such as magnetic disks, drum storage, graphics engines, etc., emphasised the unification role of device drivers. For example, the IBM 7094 mainframe computer (1962) supported several different types of remote terminals, including teletypes, IBM 1050 data communication systems, and flexowriters, and could store data on two types of storage devices: magnetic tapes and magnetic disks. The CTSS time-sharing OS used on these machines required all device drivers (called I/O adapter programs) to implement one of two standard interfaces. The following quotation is taken from a CTSS technical report [Sal65].

> Any character-type device can be attached to the system by providing an I/O adapter program which converts the raw hardware interface into the standard

> format of Interface I, which consists of one character/word in the character pool buffer.
>
> There is also another broad class of devices, such as magnetic tape, which work in terms of words, and blocks of words. A second interface is provided for these devices. ... For any input or output device for which Interface II appears to be appropriate, an I/O adapter module may be written to perform the function of matching the hardware characteristics to Interface II.

The proliferation of time-sharing computing increased the demand to protect hardware resources of the machine against unauthorised access. This led to the introduction of hardware protection mechanisms, such as the master *central processing unit (CPU)* mode in the GE-635 mainframe (1963) [GE-64]. Among other restrictions, only the privileged supervisor running in the master mode was allowed to execute I/O instructions. Non-privileged programs performed I/O by sending requests to I/O modules of the supervisor.

Further evolution of device drivers has been driven by hardware trends on the one hand, and changes in the OS architecture on the other. An example of the former is the invention of computer networks and network controller devices, which form a separate device category, distinct from character and block devices. An example of the latter is the switch to implementing OSs, and hence device drivers, in high-level languages like C.

2.2 I/O hardware organisation in modern computer systems

Before presenting the device driver architecture in modern operating systems, I give a brief overview of the hardware that these systems run on. A key difference between peripheral device organisation in modern computers compared to the early machines described in the previous section is the presence of the I/O bus hierarchy.

An I/O bus is a subsystem that connects one or more devices to the CPU. There exist a variety of I/O buses, providing different trade-offs among cost, performance, functionality, form-factor, etc. To achieve compatibility with a wide range of devices and to enable flexible system configuration, most computer systems contain several types of buses. Figure 2.1 shows a fragment of a typical desktop or server system, containing four different buses.

The *Front-Side Bus (FSB)* enables communication among the core components of the system, i.e. CPU(s) and memory. This bus is designed for fast CPU-to-CPU and CPU-to-memory transfers. For example, the latest revision of the HyperTransport [Hyp08] FSB architecture allows point-to-point communication at 51.2GB/s.

Some FSBs, including HyperTransport, allow direct connection of high-performance I/O devices. In most systems, however, devices are connected to a dedicated I/O bus, in this case the *Peripheral Component Interconnect (PCI)* bus. While being slower than the FSB, this bus provides sufficient bandwidth for efficient communication with devices, for example the current *PCI Express (PCIe)* 2.0 standard [PCI07] supports bandwidth of up to

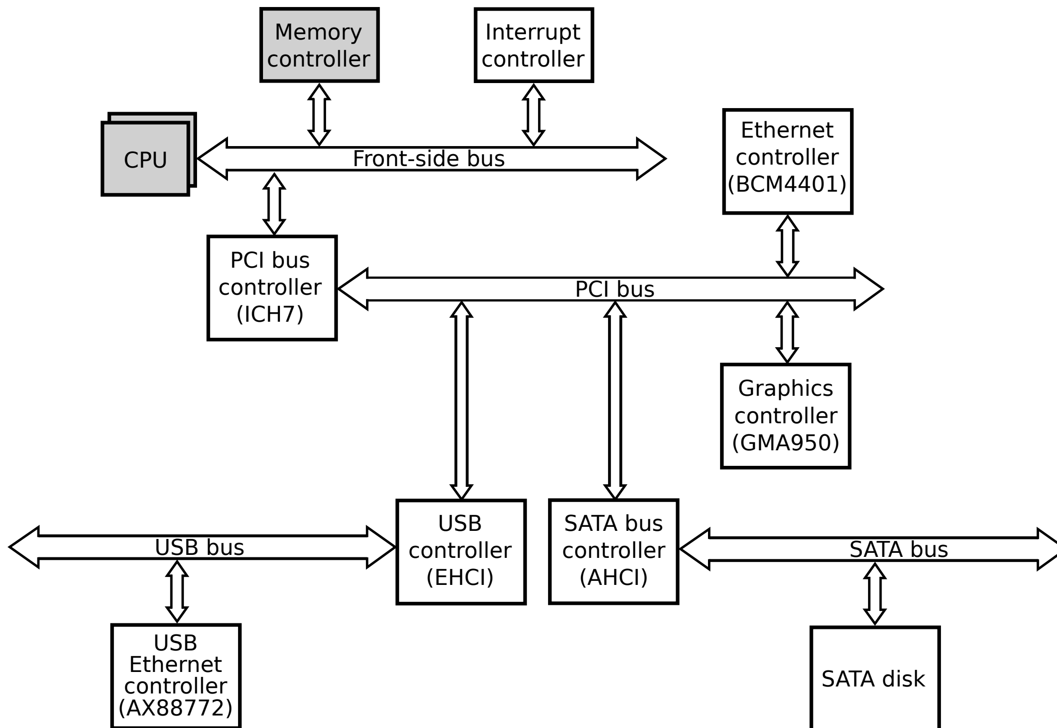


Figure 2.1: I/O bus hierarchy of a typical desktop system. Core system components (CPU and the memory controller) are represented by grey rectangles.

16GB/s per device. In addition, it allows longer physical links than most FSBs, provides support for expansion slots and expansion cables, and accommodates a large number of devices organised in a hierarchical topology.

The PCI bus is connected to the FSB through a PCI bus controller. A bus controller, or bridge, is a special device that interfaces with two buses: it appears as a client device on the parent bus and as a host device on the child bus. It receives requests destined to devices on the child bus via the parent bus protocol and forwards these requests to their destinations via the child bus protocol.

In addition to an Ethernet controller and a graphics controller, the PCI bus in Figure 2.1 hosts two secondary I/O buses. The *Universal Serial Bus (USB)* allows easy connection of a variety of external I/O devices to the computer. The *Serial AT Attachment (SATA)* bus is a specialised bus for storage devices, such as hard disks and optical drives.

Each type of bus supports its own protocol and provides its own set of operations for communication with devices on the bus. I consider PCI and USB buses in some more detail, since drivers for PCI and USB devices will be used as examples throughout the thesis.

2.2.1 Peripheral Component Interconnect bus

PCI is a ubiquitous I/O bus found in virtually all desktop, laptop, and server systems, as well as in many embedded devices. Since the introduction of PCI in 1992, three different standards have been developed in response to growing performance demands: conventional

PCI [PCI04], PCI-X [PCI02], and PCIe [PCI07]. PCI-X is a faster version of conventional PCI, whereas PCIe is based on a very different logical design. The three standards are software backwards compatible, i.e., any bus or device driver developed for conventional PCI will work correctly with a PCI-X or PCIe bus controller.

The PCI standard allows the host to communicate with devices on the bus using three separate address spaces: configuration, memory, and I/O. Every device can have one or more regions in each address space. Mappings of device regions to the corresponding global PCI address space are established during device configuration by the system *Basic Input/Output System (BIOS)* or by the OS.

The configuration space contains standard device descriptors used for device enumeration, identification, and configuration. It allows the OS to detect devices on the bus, locate a driver for each device, and control basic device capabilities, such as bus mastering, in a device-independent way.

Device-specific registers and data buffers that are accessible from software are mapped to the PCI memory space, which is in turn mapped into the physical address space of the CPU. When the CPU initiates a load or store transaction on the FSB, which falls into the address range of the PCI memory space, the PCI controller translates this transaction into a PCI memory transaction. One of the devices on the bus recognises the load or store address as belonging to its memory region and responds to the transaction.

The use of memory-mapped I/O raises memory ordering issues between the device and the CPU (or CPUs). Even though caching is normally disabled for I/O memory, other features of modern CPUs, including write reordering and load speculation, as well as compiler optimisations, can cause the device to observe stores issued by the same or different processors out-of-order. Device drivers can enforce ordering on sensitive memory operations using memory barrier instructions [How].

The PCI I/O space is an obsolete feature, which was originally introduced to improve the integration of the PCI bus into x86-based systems. In addition to the physical memory space, x86 family processors can address a 64KB I/O space. The PCI I/O space can be mapped into the processor I/O space in the same manner as the PCI memory space is mapped into the processor physical address space. One problem with this is that non-x86 architectures do not support the I/O space, and can only access the PCI I/O space indirectly through PCI controller registers. More importantly, the 64KB address range is too small to accommodate all devices in the system. As a result, most devices nowadays either do not use the I/O space or define I/O regions as aliases to memory regions.

Normally, configuration, memory, and I/O transactions are initiated by the PCI bus controller in response to a request from the CPU, however PCI devices are also allowed to initiate bus transactions if they have the bus mastering capability. This feature is used to implement the *direct memory access (DMA)* mechanism, where the device transfers data to or from the main memory without involvement of the CPU. To this end, the device starts a PCI memory transaction, specifying an address in the PCI memory space. This transaction

is recognised by the PCI controller, which translates it into a memory transaction on the FSB.

DMA is crucial for efficient I/O and is supported by all devices with non-trivial performance requirements. The simplest way to implement DMA is to keep the data exchanged with the device in a contiguous buffer, which means that the driver must copy data to be sent to the device into this buffer. High-performance devices are able to parse complex data structures that describe multiple buffers comprising one or more data transfers. Such data structures are called DMA descriptors. This facility can be used to reduce the number of data copies performed in the path from the application to the device and back.

DMA-capable devices pose potential security issues, since a misbehaving or misconfigured device can overwrite system or application data in memory. This is a particularly pressing problem in virtualised environments where several *virtual machines (VMs)* running on top of a *virtual machine monitor (VMM)* have direct access to I/O devices. A faulty or malicious VM can program an I/O device to read or write physical memory pages belonging to another VM, thus violating the isolation enforced by the VMM.

To address this problem, recent PCI bus implementations incorporate *input/output memory management units (IOMMUs)* [Int08, Adv00], which provide a mechanism to ensure that every I/O device can only access memory locations allocated to it by the OS. This is achieved using I/O page tables that map the device address space to the physical address space. Thus, the IOMMU controls physical memory accessible to devices much like the conventional MMU controls physical memory accessible to applications.

Another type of interaction initiated by the device is an interrupt notification, used to report an asynchronous event, such as a DMA transfer completion or an error condition, to the CPU. Interrupts are routed from the device to the CPU through the interrupt controller. The original PCI specification supported interrupt routing via separate interrupt pins and traces. This limited the number of available interrupts and restricted interrupt configurability. The PCIe standard and the latest versions of conventional PCI introduce a new interrupt delivery mechanism, *message signaled interrupts (MSI)*, which allows devices to signal interrupts using normal memory transactions. It enables devices to use a large number of interrupts and supports flexible assignment of interrupts to CPUs in a multiprocessor system.

Regardless of the specific interrupt delivery mechanism used, software control of interrupts can be performed at three levels. First, it is possible to completely disable the delivery of all I/O interrupts in the CPU. This facility is primarily used to achieve atomic execution of a small fragment of code, without being interrupted by external events. Second, the interrupt controller provides means to prioritise interrupts and disable individual interrupt sources (i.e., interrupts from individual devices or, in case of MSI, individual interrupts allocated to the device). The exact mechanisms for doing so are different for conventional and MSI interrupts. Third, each device provides its own device-specific interface for enabling and acknowledging interrupts, which usually consists of a set of interrupt control and status registers in the device's memory space.

A typical interrupt handling sequence in Linux proceeds as follows:

1. The device sends an interrupt message over the PCI bus. The message is routed through the PCI controller and the interrupt controller and is eventually delivered to the CPU.

2. The task currently running on the CPU is interrupted and control is transferred to the interrupt handler routine, which invokes the driver for the device through its interrupt entry point, also known as the top-half handler.

3. While the top-half handler is running, the delivery of subsequent interrupts to the CPU is postponed until the handler returns. Therefore, the top-half handler is required to return quickly, to allow interrupts from other sources to be delivered. The top-half handler interacts with the device by reading and writing registers in its memory space in order to identify the interrupt cause and clear the interrupt condition. If the given interrupt is level-triggered, meaning that the device keeps generating the interrupt signal as long as the condition that triggered the interrupt is present, then clearing the interrupt condition causes the device to deassert the interrupt line. This prevents the CPU from being interrupted immediately after returning from the primary interrupt handler. If any further long-running processing is required, it must be scheduled for execution in the context of a separate task, known as the bottom-half handler. In the unusual case when the driver is not able to clear the interrupt condition in the device in the top-half handler (e.g., some delayed processing is required), it must disable the interrupt source in the interrupt controller. Otherwise, the CPU will be interrupted immediately after completing the interrupt handler, thus going in a livelock state.

4. The bottom-half handler performs the remaining interrupt processing, e.g., delivers data returned by the device to the OS and submits new requests to the device. If the interrupt source was disabled in the top-half handler, the bottom-half handler must reenable it to allow subsequent interrupts from the device.

2.2.2 Universal Serial Bus

The USB 1.0 specification was released in 1996 and has by the time of writing undergone two major revisions: USB 2.0 [USB00] and USB 3.0 [USB08a]. In addition, a wireless USB standard [USB05] was introduced in 2005. The description in this section applies to wired USB version 2.0 and earlier.

USB is a host-centric bus, i.e., all transfers to and from USB devices are initiated by the bus controller. Every USB device implements a number of communication endpoints. The bus controller interacts with the device by writing or reading data through the endpoints.

The bus controller initiates a transfer by sending a setup packet, which identifies the target device and endpoint. This is followed by a series of data and handshake packets.

The bus controller broadcasts all packets to all devices on the bus, however only the target device responds to these packets.

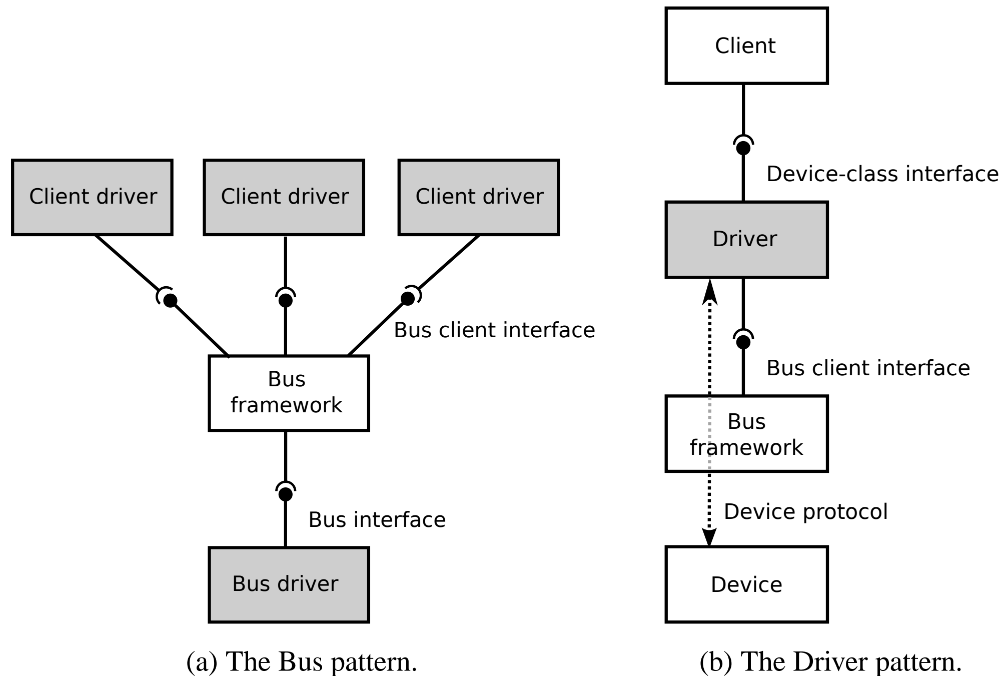
The USB standard defines four types of endpoints and, respectively, four types of transfers. Control endpoints are used to enumerate and configure devices. They typically use short sporadic transfers. Bulk endpoints are used to reliably exchange large amounts of raw data, such as network packets or disk blocks. Isochronous endpoints are best suited for real-time audio or video streaming. They guarantee access to USB bandwidth with bounded latency, but do not provide reliable transfer. If a packet is not delivered because of a bus error or software delay, the error is detected on the receiver side, but no retransmission occurs. Finally, interrupt endpoints are used to notify the host about device status changes.

USB interrupts are implemented differently from PCI interrupts. Since all USB transfers are initiated by the bus controller, there is no way for an interrupt endpoint to notify the host about an interrupt without being asked for it by the bus controller. Therefore the bus controller periodically polls interrupt endpoints to check for an outstanding interrupt.

The USB bus supports a tree topology with USB hubs acting as nodes and non-hub devices as leaves. The root of the tree is the root hub device integrated with the bus controller. A USB hub is able to detect and report to the bus controller when a new device is connected to it. The bus controller notifies the OS about the new device. The OS then assigns a USB bus address (ranging between 1 and 127) to the device, discovers device capabilities, and allocates a power budget (i.e., how much current the device can draw from the bus) to it by issuing a series of commands to the bus controller and the hub.

The mechanism for accessing devices on the USB bus from software is determined by the bus controller. There exist three standard bus controller architectures: *Open Host Controller Interface (OHCI)* [OHC99], *Universal Host Controller Interface (UHCI)* [Int99], and *Enhanced Host Controller Interface (EHCI)* [Int02]. All of them rely on the bus controller driver to maintain an in-memory data structure that describes a schedule of USB transfers. This data structure contains pointers to the actual data buffers to be sent to devices on the bus or to be filled with data received from devices. The address of this structure is stored in a bus controller register. The bus controller iterates over this schedule using DMA over the PCI bus and executes scheduled transfers in the prescribed order. Results of completed transfers (status codes, number of bytes read or written, etc.) are DMAed to another in-memory data structure.

Thus, communication with a device on the USB bus is mediated by the bus controllers located in the path between the CPU and the device in question, in this case the PCI and the USB controllers. A software architecture for managing this bus hierarchy is presented in the following section.



(a) The Bus pattern.

(b) The Driver pattern.

Figure 2.2: Architectural patterns used in the design of operating system I/O frameworks. Lollipop connectors represent interfaces; the dashed arrow represents the communication path between the driver and the device, consisting of the software bus framework and the hardware bus hierarchy (not shown in the figure).

2.3 Device drivers in modern operating systems

This section gives an overview of the OS subsystem responsible for managing peripheral devices, often referred to as the operating system I/O framework. While the details of the I/O framework design vary across different systems, any such framework must implement two common architectural patterns shown in Figure 2.2.

The Bus pattern (Figure 2.2(a)) is centred around a bus controller driver (or simply "bus driver"), which provides a generic interface to the bus transport, e.g., PCI or USB. Internally, it encapsulates the details of a specific bus controller device interface. The bus driver is managed by the bus framework, which is responsible for enumerating devices on the bus, instantiating a driver for every device, multiplexing the bus among multiple client drivers, and tearing down drivers for devices that get disconnected. The bus framework is implemented once for every type of bus (e.g., PCI or USB) and is independent of the particular bus controller implementation. It presents each client driver with a high-level interface to the bus. The Bus pattern is instantiated for every I/O bus in the system.

To illustrate this pattern, consider a PCI bus driver, which provides an interface for raw access to the PCI configuration, memory, and I/O spaces. The PCI bus framework uses the bus driver to enumerate PCI devices at startup, by reading the content of the configuration space, and creates a driver for every device on the bus. It provides each client driver with access to functions to read and write configuration, memory, and I/O regions that belong

to its device. It may also provide convenience functions to perform standard configuration actions on the device, such as reading device identification data from the configuration space.

The Driver pattern (Figure 2.2(b)) consists of a driver for an I/O device that communicates with the device via the bus transport interface provided by the bus framework. It hides device details, such as the register layout or the format of DMA descriptors supported by the device and exports a device-class-specific interface to its client. For example, an Ethernet controller driver implements functions to send and receive network packets. The client of the driver in this case is the network protocol stack. The Driver pattern is instantiated for every I/O device in the system, including bus controller devices.

Since this thesis is concerned with the design of device drivers and driver interfaces, the Driver pattern will frequently occur in the following chapters. This pattern can be viewed as a refinement of the abstract driver architecture introduced in Figure 1.1. It decomposes the OS interface of the driver into two separate interfaces: the bus client interface and the device-class interface. It also emphasises the fact that communication with the device is mediated by the bus infrastructure.

The two patterns overlap. If the device driver in the Driver pattern is a bus driver, then its device-class interface is the bus interface from the Bus pattern, and its client is the bus framework. Figure 2.3 illustrates this overlap by showing two bus drivers stacked on top of each other.

Thus, when applying these patterns to instantiate an I/O framework for a specific hardware configuration, the resulting software architecture mirrors the hardware topology, with device drivers stacked on top of their corresponding bus drivers. This is illustrated in Figure 2.4 using the example of the system in Figure 2.1. Figure 2.4(a) shows an alternative representation of the same system in the form of a tree, with bus controllers in tree nodes and other devices in leaves. Figure 2.4(b) shows the corresponding software architecture.

Some of the drivers in Figure 2.4(b) are drivers for specific device models, e.g., the BCM4401 Ethernet adapter, while others are generic drivers that can manage analogous devices from multiple vendors. Examples of the latter are the generic SCSI disk driver and the USB EHCI controller driver. This reflects the trend towards device standardisation, when a regulatory body that defines a family of hardware protocols also defines a standard architecture for devices that implement this protocol. This approach reduces the number of poorly engineered devices. In addition, it allows hardware vendors to avoid developing their own drivers: as long as the device complies with the standard, it can be managed by a generic driver provided with the OS. Thus, by unifying hardware interfaces, standardisation reduces the number of drivers that need to be developed and maintained, which leads to better tested drivers.

Nevertheless, there still remains much diversity among devices. In order to maintain a competitive advantage, many vendors define their own interfaces or extend existing interfaces to achieve better performance. Others try to cut down on development costs by

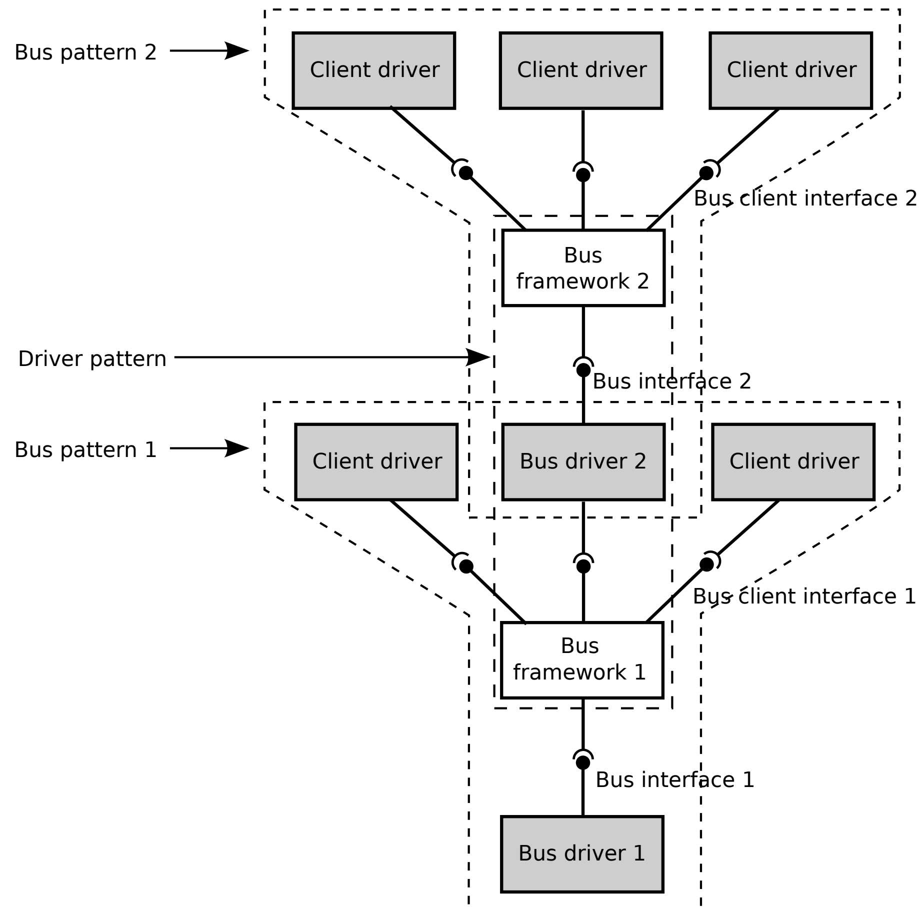


Figure 2.3: Superposition of Driver and Bus patterns.

implementing a scaled down or modified version of the standard. If the deviation from the standard is minor, it can be handled as a special case in the generic driver. Otherwise, a separate driver needs to be developed.

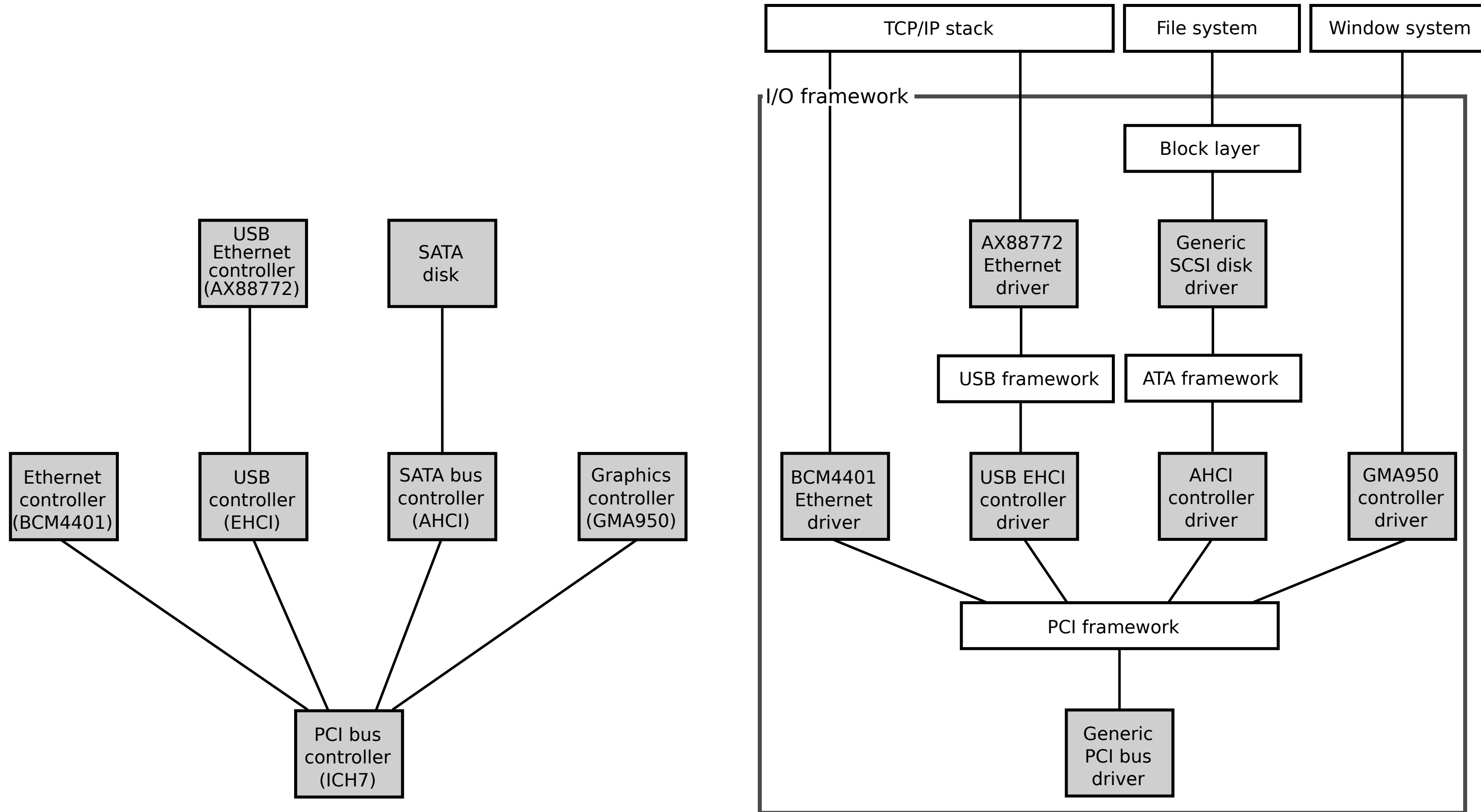
2.4 Generic OS services

In order to accomplish their function of managing I/O devices, most drivers rely on some generic (i.e., device-independent) services provided by the OS, most noticeably, memory management, timing, and synchronisation services. Thus, in addition to the device-class interface and the bus client interface (Figure 2.2b), the driver interacts with the OS through the generic services interface, as shown in Figure 2.5.

2.4.1 Memory management

Most drivers use conventional `malloc`-style kernel memory managers to allocate storage for their internal dynamic data structures. In addition, drivers for devices that support DMA must manage memory buffers for communication with the device.

The concrete interface for I/O buffer management is OS-specific, however the function-



(a) A tree representation of the device hierarchy in Figure 2.1.

(b) The software architecture used to manage this device hierarchy.

Figure 2.4: Device driver stacking.

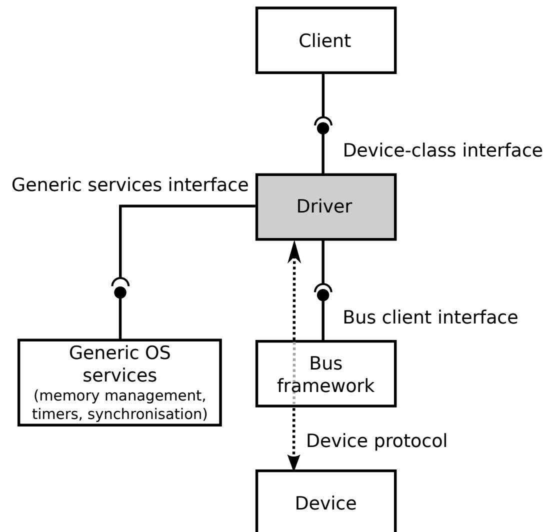


Figure 2.5: A refined device-driver architecture, including the generic services interface.

ality provided through this interface can be expressed in terms of five basic operations.

- Allocate a region of virtual memory with the given size and alignment. The driver uses this operation to allocate buffers for data exchanged with the device.

- Pin down the entire region or some of its pages to physical memory. This ensures that buffers transferred to or from the device are located in the physical RAM, where the device is able to access them. In some cases, the driver may request that physical pages for the region are allocated contiguously. This is useful when the DMA engine of the device is only capable of dealing with contiguous buffers.

- Mark the region as uncacheable, so that reads and writes to the region are forwarded directly to the PCI bus and are not intercepted by the CPU cache.

- Perform a virtual-to-bus address translation inside an I/O region. Since devices are only capable of issuing load and store operations in the bus address space (e.g., the PCI memory space), buffer pointers passed to the device must contain addresses in this space.

- Perform a physical-to-virtual address translation on a pointer received from the device. Devices that deal with multiple data buffers store physical addresses of completed buffers in memory or register data structures. The driver reads these physical addresses and converts them to virtual addresses in order to be able to access the buffers and perform further processing on them (or simply hand them back to the OS).

2.4.2 Timers

There exist two common situations when a device driver needs to use the OS timer service. The first one occurs when the driver issues a long-running command to the device that does not generate an interrupt upon completion. For example, this is the case for the reset operation in most devices. The device specification defines the upper timing bound for such operations, which allows the driver to synchronise with the device by waiting for the specified amount of time using the timer service.

Second, timers can be used to detect a device malfunction, when the device fails to reply to an I/O request within a certain time frame. A device-specific action, e.g., a reset can then be taken to return the device to a normal state.

The OS enables time-driven behaviour through asynchronous and synchronous timer services. An asynchronous timer invokes a specified callback function when the timeout expires. A synchronous delay suspends the current thread for a requested amount of time.

2.5 Concurrency and synchronisation in device drivers

Most OS kernels are multithreaded, meaning that all parts of the kernel, including device drivers, must be prepared to handle invocations in the context of multiple concurrent threads. Programming in the multithreaded environment requires careful use of synchronisation primitives to avoid race conditions and deadlocks.

Further complications arise from the fact that in some systems, including Linux, driver entry points can be invoked in the primary interrupt context (see the discussion of top and bottom halves in Section 2.2.1). At this time the kernel is running with interrupts disabled, hence an attempt to acquire a lock or invoke a potentially blocking operation may deadlock the system. The driver developer must be aware of which driver methods can be invoked in the interrupt context and structure the driver to avoid blocking in these methods. The only synchronisation primitives that can be safely used with interrupts disabled are spinlocks. Spinlocks are busy locks, suitable for protecting small critical code sections against concurrent access from multiple processors. Whenever blocking is unavoidable, the driver must postpone operations that involve blocking to be executed in the bottom half context. Linux provides two mechanisms for constructing bottom halves: tasklets and work queues. Bottom halves execute in a separate thread, concurrently with the rest of the driver, and hence must be synchronised with other driver functions.

Operating system I/O frameworks provide driver developers with varying degrees of support for dealing with concurrency. Some systems, including Linux and versions of Windows before Windows Vista, treat device drivers as regular kernel components responsible for their own synchronisation. Others provide generic synchronisation facilities for device drivers, which shift much of the synchronisation complexity from the driver into the framework. Two examples of such facilities are the workloop architecture implemented in the

Mac OS X IOKit framework [App06] and synchronisation scopes in the Windows Driver Foundation [Mic06].

2.5.1 The Mac OS X workloop architecture

The workloop architecture associates a lock with the entire stack of drivers connected to a single hardware interrupt. For example, a USB host controller driver and drivers for all USB devices connected to the controller share the same lock. The framework guarantees that most driver operations are protected by this lock. Other operations can optionally acquire the lock, thus ensuring that their execution is serialised with respect to other workloop-protected operations.

The main limitation of this architecture is that the workloop lock should not be held for extended periods of time, since this would delay other requests to the driver and negatively impact the performance. Furthermore, executing a blocking operation in the workloop context can cause a deadlock. Two mechanisms are provided to perform such operations outside the workloop. First, the driver may temporarily drop the lock before calling a blocking function or waiting for an I/O completion. When used in this way, the workloop effectively functions as a monitor [Hoa74].

Second, the driver may schedule the blocking operation for execution in the context of a separate kernel thread. This approach is used to implement driver functions that are not allowed to block. The helper threads may require access to driver state variables and therefore need to be synchronised with each other and with other driver threads. The synchronisation is achieved using conventional locking primitives such as mutexes and semaphores.

2.5.2 Windows Driver Foundation synchronisation scopes

The Windows Driver Foundation I/O framework was introduced in the Windows Vista OS. Among other improvements aimed at enabling simpler and more reliable drivers, the framework provides configurable support for automatic serialisation of driver invocations. The driver may choose one of the following serialisation scopes:

- No scope: any driver entry points can be invoked concurrently.
- Queue scope: requests from individual I/O queues are serialised, while requests from different queues can be delivered concurrently.
- Device scope: all I/O requests to the driver are serialised.
- Complete serialisation: all I/O requests, interrupts, and timer events are serialised.

Serialised operations are required to complete in a non-blocking fashion to avoid deadlocks and performance degradation. Long-running operations must be executed in a separate thread, outside the serialisation scope and must be synchronised with other operations using conventional locking primitives.

Both the Mac OS X workloop mechanism and Windows synchronisation contexts reduce the amount of concurrency that the driver developer has to handle, but do not eliminate it completely. Specifically, drivers are responsible for scheduling long-running operations in the context of separate kernel threads and for synchronising their execution with the rest of driver code.

In addition, these serialisation architectures associate one or several coarse-grained locks with the driver, which can be detrimental to performance on multiprocessor systems. This is the reason why both frameworks allow some flexibility in choosing which operations should be serialised, offering configurable trade-off between performance and programming convenience.

Chapter 5 presents a device driver architecture that overcomes both limitations, i.e., it allows complete serialisation of all driver invocations without significant performance overhead.

Another example of a driver architecture taking systematic approach to concurrency management is the *Uniform Driver Interface (UDI)* standard [Pro01]. The standard was developed to enable driver portability across different OSs. In particular, it specifies that the driver should be partitioned into one or more regions. Invocations of every region are serialised by the UDI framework. It did not, however, specify how drivers were expected to handle blocking operations. The standard has not been adopted by the industry and no production-quality drivers have been developed based on it. As a result, the architecture and performance of UDI drivers have not undergone practical evaluation.

2.6 Hot plugging

Many modern I/O bus architectures, including PCI and USB, allow devices to be connected and disconnected from the bus at runtime. Device connection is detected by the bus driver, which notifies the bus framework about the event and reports device identification information. The bus framework uses this information to locate and load a driver for the device.

Device disconnection is also detected by the bus driver, which sends a notification to the bus framework. Before unloading the device driver, the framework notifies it about the hot-unplug event, giving the driver an opportunity to release any resources that it is holding. Since the hot-unplug event happens asynchronously to all other operations of the driver, the driver must be prepared to handle this event in any state.

2.7 Power management

Reducing power consumption is an important concern for most computer systems, from portable embedded appliances to servers. Peripheral devices typically account for a large fraction of the overall power used by the system. Hardware support for reducing I/O power consumption can be classified into device-internal power management and bus power man-

agement. Device-internal power management features allow changing the power consumption of the device without changing the power state of its bus interface. For instance, a hard drive can be spinned down, which substantially reduces the amount of power it uses.

Bus power management provides mechanisms to limit the amount of power the device can draw from the bus and to put the device in a sleep state where most device functions are disabled and the device configuration can be partially or completely lost. Different sleep states differ in the amount of power that the device can draw from the bus and the time required to resume normal operation from the given state. Lower-power states correspond to longer resume times.

Inside the OS I/O framework, sleep requests propagate from leaf devices towards the root of the bus hierarchy. Leaf devices are suspended first based on the OS power-management policy (e.g., after a period of inactivity). When all devices on an I/O bus have been suspended, it is possible to suspend the bus controller as well. Before suspending the device, the OS notifies the driver about the upcoming suspend, giving it the opportunity to complete any outstanding operations and save context information that will be lost after the transition to the low-power mode.

Resume requests propagate from the bottom to the top of the stack: the entire hierarchy of bus drivers must be resumed before resuming a leaf device. Once the device power state is restored, its driver is notified by the OS, so that it can restore the saved context and prepare the device for handling I/O requests.

2.8 A taxonomy of driver failures

This section sets the stage for the following discussion of driver reliability techniques by enumerating the various ways in which a driver can misbehave. A definition of incorrect behaviour, or failure, can only be given in relation to a selected vantage point, where the system execution is observed. For example, from the end-user perspective, observable failures include system crashes and hangs, lack of network connectivity, file access errors, etc. At the other extreme, if we trace the behaviour of the driver at the level of individual programming language operators, then observable failures include type and memory safety violations, some of which may not even lead to any externally visible consequences.

The taxonomy of failure proposed here uses the interface between the driver and its environment, comprised of the OS and the device, as the vantage point. All driver failures observable at this level can be classified into *OS protocol violations*, *device protocol violations*, and *memory access violations*.

The OS protocol of a driver defines constraints on the communication between the driver and the OS. This communication must be restricted to the three interfaces shown in Figure 2.5 (i.e., the bus framework interface, the client interface, and the generic OS services interface) and must follow the rules on the ordering, timing, and content of interactions, associated with each interface. OS protocol violations include:

- **Ordering violation.** OS protocols tend to be stateful, meaning that the set of protocol operations that can be performed at a given time depends on the history of previous interactions. For example, a network driver in Linux is not allowed to feed incoming packets to the OS before registering its network interface with the TCP/IP layer using the `register_netdev` function.

- **Data format violation.** Data structures exchanged between the driver and the OS must follow certain format restrictions, which include static invariants, e.g., a linked list must not contain cycles, as well as invariants that depend on the context in which the data structure is used, e.g., an incoming packet passed by a network driver to the OS must not be empty.

- **Access to unauthorised services.** A misbehaving driver may attempt to invoke an OS function that is not part of its interface or directly call a privileged CPU operation. For example, an in-kernel driver can call a function that modifies page table entries and is only intended for use inside the virtual memory system.

- **Excessive use of system resources.** Drivers consume system resources, most importantly physical memory and CPU cycles. In an OS that does not enforce resource allocation limits on device drivers, a buggy or malicious driver may consume excessive amount of resources, leading to resource starvation in other parts of the system. Typical errors that cause such failures include memory leaks, infinite loops, and misplaced spinlocks.

- **Temporal failure.** A failure of a driver to respond to OS requests in a timely manner may affect the quality-of-service for both real-time and non-real-time applications. In the extreme case of an infinite delay, e.g., due to a deadlock, the entire system or its parts may become permanently unavailable.

Any of the above OS protocol violations may lead to arbitrarily severe consequences, including compromised system security and integrity.

Device protocol violations occur when the driver behaves in a way that violates the required hardware protocol, and typically result in a failure of the hardware to provide its required service. They include:

- **Incorrect use of the device state machine.** In response to an OS request, a device driver must take the device through a sequence of states, resulting in the request being satisfied by the device hardware. An incorrect driver implementation may fail to do so. Typical violations include submitting a malformed command to the device, issuing data transfer commands to an incompletely configured device, failing to correctly recover the device from a transient hardware fault, and issuing a sequence of commands that render the device temporarily or permanently unusable.

- **DMA violation.** The DMA mechanism enables shared-memory communication between the device and the driver. As mentioned above, the format of DMA data structures can be rather complicated. An error in the management of these data structures can cause the device to read incorrect data from the memory or to issue a write transaction to a random memory location. In the absence of an IOMMU, such runaway DMA transactions can corrupt the system state. Typical DMA-related errors include incorrectly formatting DMA descriptors, forgetting to pin buffers down to physical memory before passing them to the device, and race conditions between the device and the driver, e.g., when the driver acknowledges the receipt of a block of data from the device before actually processing the data.

Finally, a memory access violation occurs when the driver attempts to read or write a memory location that has not been granted to it by the OS (e.g., through an explicit or implicit memory allocation).

Chapter 3

Related work

Faulty device drivers have long been recognised as the biggest threat to system stability. A variety of techniques have been proposed for dealing with driver errors. This chapter presents a survey of these techniques and identifies their strengths and limitations.

Existing methods for improving software reliability fall into three major groups: fault prevention, fault removal, and fault tolerance. The term "fault" here refers to an algorithmic defect in the driver that, when triggered at runtime, causes a failure. I outline each of the three approaches below. The following sections describe these approaches and their existing implementations in more detail.

Fault prevention aims to prevent defects from being introduced in the system. In the context of device drivers, fault prevention has been achieved using two approaches. The first one consists of developing drivers using high-level programming languages where certain types of errors are not expressible. For example, languages that do not allow direct pointer manipulation eliminate errors in pointer arithmetic. The second approach is to generate a partial or complete implementation of the driver automatically from a formal specification of the required behaviour, thus avoiding the impact of coding errors on the driver reliability.

Fault removal techniques detect and eliminate defects before putting the system in production use. Early fault detection is performed by the compiler. Compiler-aided fault detection is particularly effective for languages with advanced type systems, such as Haskell and C#, where many failures can be expressed as type safety violations. Therefore, several research projects have investigated the use of such languages for implementing device drivers and other OS components. Other types of faults can be detected with the help of static analysis and model checking tools, which analyse the source or binary representation of the driver against a formal specification of some desired properties and identify behaviours that violate these properties. Finally, automated and manual testing remains the most common fault removal method for device drivers.

Fault tolerance techniques enable the system to continue normal operation in the face of driver failures. A complete fault tolerance solution must include fault isolation and recovery components. The former is responsible for detecting a driver failure and preventing it from propagating to the rest of the system. This can be achieved with the help of hardware protection mechanisms, e.g., the *memory management unit (MMU)* available in most modern processors, or using purely software-based techniques. The latter is responsible for performing compensatory actions, such as reporting a failure to all existing clients of the driver and creating a fresh instance of the driver to service new clients.

Any single fault prevention, fault removal, or fault tolerance technique is capable of preventing, removing, or tolerating only certain subclasses of defects listed in Section 2.8. Most existing implementations surveyed in this chapter combine several different techniques to achieve better reliability. For example, the Singularity OS [FAH+06] uses compile-time checking in combination with static analysis, and runtime fault isolation to protect against the majority of driver faults.

3.1 Hardware-based fault tolerance

This section surveys fault isolation and recovery techniques that rely on CPU protection mechanisms to encapsulate device drivers inside unprivileged protection domains. An encapsulated driver cannot directly invoke privileged CPU instructions and can only access memory locations granted to it by the OS. Communication and memory sharing are mediated by the OS, e.g., through the system call mechanism, and can be monitored for ordering or format violations. Driver protection domains are managed similarly to normal user tasks, in particular, they are subject to OS scheduling, memory allocation, and other resource management policies.

Fault recovery is implemented by destroying the entire protection domain and releasing all resources allocated to it, followed by the creation of a fresh copy of the driver in a new protection domain. Complete failure transparency can be achieved by keeping track of the state of the driver-OS interaction and brining the new driver instance to the state preceding the failure. This approach only works for transient failures that are unlikely to occur again after the recovery. Alternatively, the recovery mechanism may simply inform the OS about the failure, allowing existing users of the driver to either fail gracefully or to perform application-specific recovery.

User-level device drivers were pioneered in Michigan Terminal System [Ale72], an OS for the IBM System/360 mainframe computer. However this approach has not found wide acceptance and the majority of OSs nowadays still implement device drivers as part of the privileged kernel.

3.1.1 Drivers in capability-based computer systems

Early implementations of hardware-based driver isolation were found in operating systems for capability-based computers, such as Plessey 250 [Eng72], Cambridge CAP [NW77], and MONADS [Kee78]. These computers were created from the late 60's through 70's, during the search for hardware mechanisms that would enable efficient and flexible protection, communication, and data sharing in multitasking environments. The concepts behind capability-based architectures were formulated by Dennis and Van Horn [DVH66]. A systematic description of a number of such architectures was given by Levy [Lev84].

Every process in a capability-based system executes in the context of a capability space, which contains a list of objects that the process can access. The two types of capabilities are segment capabilities and protected control transfer capabilities. A segment capability allows the process to read, write, or execute the content of a contiguous memory segment. A protected control transfer capability allows the process to invoke a procedure in a different capability space.

Protected control transfers are implemented in hardware, and therefore can potentially provide a more efficient *inter-process communication (IPC)* mechanism, compared to conventional OS-mediated communication (in practice, this is not necessarily the case, as examplified by the lacklustre performance of the Intel 432 processor [CGJ88]). This allows most system services, including device drivers, to be implemented as normal capability-protected processes, without loss of performance.

In capability-based architectures, I/O device registers are mapped into the system memory space and can be protected with capabilities, similarly to other types of resources. A device driver is granted a capability to the memory region containing device registers. All other system and application processes access the device by invoking services exported by the driver through protected entry points.

While this isolation architecture can be naturally extended with means for runtime monitoring and recovery, to the best of my knowledge, none of the above-mentioned systems provide such mechanisms.

3.1.2 User-level device drivers in microkernel-based systems

Microkernel-based OSs are built around a small privileged kernel, with the majority of system services implemented as user-level processes or libraries. In some cases, these include a user-level implementation of device drivers. Unlike systems discussed in the previous section, microkernel-based OSs run on architectures with conventional memory protection facilities.

There exist three distinct styles of microkernel-based OS architectures. Single-server systems implement the entire OS personality, including device drivers, in a single user-level server process. Different OS personalities can be supported by multiple concurrent servers. A fault in an OS server can bring down the entire server along with all its clients.

Multi-server systems decompose an OS personality into multiple server processes, each implementing a single service, such as a file system, a pager, or a device driver. These systems enable fault isolation for individual services at the cost of increased communication overhead. Finally, library-based OSs implement most of the OS *application programming interface (API)* in libraries that are linked against user processes. This architecture localises the effect of an OS fault to a single application. Shared services, such as device drivers, must still be implemented as separate processes or as parts of the kernel.

The original motivation for the microkernel-based design was put forward by Brinch Hansen [BH70] during the work on the OS for the RC 4000 computer. He argued that a multiuser system should not be restricted to a fixed OS API and a fixed set of resource management policies. Improved flexibility can be achieved by exposing multiple concurrent OS personalities to the user. These personalities are built on top of a policy-free kernel, or "nucleus", which plays the role of a "software extension of the hardware structure, which makes the computer more attractive for multiprogramming". The nucleus supports a hierarchical resource management model, where every process allocates memory and CPU resources for its children processes from its own resource pool. This model enables both single-server and multi-server implementations of OS personalities.

The idea of a policy-free kernel was further refined in the Hydra OS [WCC+74]. Hydra was designed to facilitate experimental exploration of the OS design space, which required the ability to easily modify existing resource management policies and to add new types of resources and policies to the system. These design goals required a highly modular system structure. To this end, Hydra adopted a multi-server architecture, consisting of a policy-free kernel and a collection of user-level services, which resided in separate protection domains and were accessible via a protected procedure call mechanism implemented by the kernel.

Both Brinch Hansen's nucleus and Hydra implemented device drivers as part of the kernel; however the concepts promoted by these systems naturally led to user-level driver architectures in later microkernel-based systems, such as Mach 3 [ABB+86, FGB91] and L3 [LBB+91, Lie93].

The first versions of Mach featured a hybrid design, with most of the OS functionality implemented in the kernel, which was structured as a collection of independent threads interacting via a message-based IPC mechanism. Later, various system components, including device drivers, were gradually moved out of the kernel [FGB91]. The transition was simplified by the location transparency of Mach IPC, which supported uniform communication with kernel, user-level, and even remote processes. In order to enable access to I/O devices from the user-level processes, Mach mapped device registers to the address space of the driver and vectored device interrupts to the driver thread.

Curiously enough, the original motivation for moving drivers out of the Mach kernel, cited by Forin et al. [FGB91] was improved performance for single-server systems, where the entire OS functionality was implemented in one server process, e.g., the UNIX simulation server. Incorporating device drivers into the server eliminated the overhead of switching

to the kernel on every driver invocation.

The L3 microkernel [LBB+91] followed a different and more principled approach. The kernel was initially designed to only incorporate a minimal set of mechanisms to enable user-level implementation of the complete OS functionality. These mechanisms included thread creation and scheduling, address space management, persistence, interrupt dispatching, and synchronous IPC. File systems, device drivers, and other system services were built on top of the kernel in the multi-server style.

The initial enthusiasm over the microkernel technology was undermined by the failure to build a high-performance commercial-quality microkernel-based OS. The most famous example is the multi-billion-dollar IBM Workplace OS [Fle97] project, which aimed to build a commercial multi-personality OS on top of Mach, but was closed when, after four years of development, it became clear that the project would not be able to achieve its functionality, reliability, and performance objectives.

In retrospect, the failure was caused by the overly ambitious goals set by Workplace and other microkernel OS projects. Apart from implementing conventional OS functions on top of a microkernel, these systems aimed to provide support for multiple personalities, distribution, persistence, checkpointing, multiprocessing, and other features. In pursuing all these goals, the developers underestimated the complexity of the basic task of building a reasonably efficient OS as a collection of multiple user-level servers. Microkernel proponents claimed that by moving functionality out of the kernel, microkernels would automatically enforce clean system design. In reality, microkernels provide basic mechanisms for modularising the system, but do not help in defining the right module boundaries and interfaces. Moreover, by complicating data sharing and increasing the cost of communication, they introduce additional concerns that must be addressed to build a performing system.

While focusing on the advanced features, designers of first-generation microkernels failed to solve the basic problem of providing an efficient inter-process communication mechanism, which is the key to building an efficient OS. In a microkernel-based system, IPC is used for communication between user applications and the OS, as well as for intra-OS interactions. High IPC cost in Mach and other microkernels led to substantial end-to-end performance degradation for applications and eventually forced the designers to move OS services, including device drivers, back to the kernel [CBMM94], thus defeating the reliability improvements offered by the user-level design.

Liedtke [Lie93] carried out a detailed analysis of the costs involved in inter-process communication, which led him to conclude that the poor IPC performance in systems like Mach and L3 was a consequence of the complex semantics of the selected IPC primitives and their suboptimal implementation. Based on these findings, he reimplemented the L3 IPC mechanism and later constructed a new L4 microkernel [Lie95], achieving a 20-fold performance improvement. Optimisations used in these systems included reduced data copy overhead, reduced kernel cache and *translation lookaside buffer (TLB)* footprints, and scheduling optimisations.

Härtig et al. [HHL+97] demonstrated that the performance of the Linux OS running as a user-level process on top of L4 is within 6-7% of native Linux on kernel compilation benchmarks. However they do not specifically report the performance of user-level drivers. In general, little published data exists on the performance of device drivers in microkernel-based systems. Among available results are those for the DROPS OS [HBB+98], based on a real-time version of L4 [Hoh02, HLM+03], and for the Fluke microkernel [FHL+99] user-level driver framework [VM99]. Both systems use second-generation microkernels, however their IPC performance was not optimised to the same degree as in L4. They achieve I/O throughput close to that of a monolithic kernel at the cost of around 100% increase in CPU utilisation under heavy I/O loads.

This overhead is due to several factors. First, the cost of handling interrupts at the user level involves an extra context switch to the driver process. Second, every I/O request sent to the driver, as well as a completion notification from the driver to its client, involves an extra context switch. Finally, every request and completion involves transfer of data across process boundaries. Whether it is accomplished by copying or mapping, the cost of this operation is quite high.

Techniques have been proposed to mitigate these overheads. In particular, the cost of interrupt delivery can be reduced via interrupt rate limiting [MR96] and more aggressive use of interrupt coalescing (i.e., reducing the number of interrupts by configuring the device to generate a single interrupt for multiple completed I/O operations). The overhead induced by the interaction between the driver and its client can be reduced by buffering requests and responses and by passing data via shared memory, as discussed below. Both interrupt coalescing and buffering decrease CPU utilisation at the cost of increased I/O latency. This tradeoff is acceptable for the majority of devices, which are optimised for throughput rather than latency.

An efficient buffering mechanism, called rbufs, was proposed in the Nemesis [LMB+96] microkernel-based OS. Nemesis was designed for real-time processing of multimedia data, therefore efficient data streaming between the network controller, the disk, the graphics card, and the main memory was one of the primary design goals. The rbufs mechanism is based on shared memory and asynchronous events. Shared memory is used to exchange data without copying or mapping memory to the receiver's address space on every transfer. An asynchronous event is a communication primitive, provided by the kernel, which increments a value in the receiving process's address space and makes the process runnable if it was waiting for the event, without blocking the sending process. It does not involve any data transfer and therefore allows efficient implementation.

With rbufs, one can establish a communication channel between two processes, consisting of a shared memory region and a pair of circular buffers. The shared memory region is used to store data buffers exchanged by the two processes. It is writable by the data originator and read-only to the receiver. The circular buffers also reside in shared memory and are used to queue transfer descriptors. The master process, which initiates the transfer (it

can be either the originator or the receiver), writes request descriptors to the first circular buffer. The slave process reads request descriptors from the first buffer and writes response descriptors to the second buffer. A buffer is writable by the writing process and read-only to the reading process. Each circular buffer has a pair of event channels associated with it. These are used to communicate head and tail pointers between the writer and the reader of the buffer.

Rbufs preserve strong isolation of the communicating processes, while avoiding copying and reducing the number of context switches, which enables good performance for user-level device drivers under I/O-intensive workloads. Unfortunately, none of the Nemesis publications show concrete device driver performance numbers.

In summary, microkernel-based systems allow the isolation of device drivers inside user-level processes at the cost of significant CPU overhead. Approaches to reducing this overhead have been proposed, including optimised IPC facilities and performance tuning (e.g., interrupt coalescing). Nevertheless, after two decades of active research no microkernel-based OS has demonstrated a user-level device driver framework whose performance would be close to that of a monolithic system.

User-level device drivers rely on OS protection mechanisms to detect and isolate memory access violations and a subset of OS protocol violations, such as access to unauthorised services and excessive resource allocation (see Section 2.8). A complete fault tolerance solution is required to also detect other types of failure and to provide mechanisms for failure recovery. While all of the systems discussed above pointed out the possibility of constructing such facilities, none of them has demonstrated a working implementation.

Recently, an attempt at building a complete driver reliability infrastructure for the MINIX 3 [HBG+06] microkernel-based OS was undertaken by Herder et al. [HBG+09]. They extended fault isolation facilities available in previous systems with a limited form of temporal failure detection based on heartbeat messages. In addition, MINIX 3 isolates the most severe form of device protocol violations using an IOMMU.

MINIX 3 also implements an original approach to fault recovery by outsourcing the task of restoring the state of the failed driver to its clients [HBG+07]. The OS simply creates a new copy of the driver and reconnects it to the client. The client is notified about the failure and can take compensatory actions to conceal the failure from user applications. The rationale behind this design is that the client often contains data required for recovery and hence can implement recovery without the overhead of runtime monitoring. For instance, the file system stores pending disk operations and can reissue them in case of the disk driver failure. As another example, a network protocol stack handles network controller driver failures by detecting and retransmitting lost packets as part of the TCP protocol; applications that use unreliable protocols, like UDP, implement application-specific mechanisms for tolerating lost packets.

A number of microkernels and similar systems were not covered in the above survey, because, despite their adherence to microkernel design principles, they implement drivers

as part of the kernel. These include V [Che84], Accent [RR81], the Cache kernel [CD94], Amoeba [TKRB91], early versions of Chorus [RAA+88], Exokernels [EKO95] and others. Among currently active microkernels that support user-level drivers are both commercial microkernels, such as OKL4 [OKL], QNX [Don00], and INTEGRITY [Gre] and academic systems like seL4 [EDE07], MINIX3 [HBG+06], and Nexus [SWSS05].

3.1.3 Device driver isolation in monolithic OSs

Research on user-level device drivers in microkernel-based systems inspired the use of similar isolation techniques in conventional monolithic OSs. In a monolithic system, the client of the driver, e.g., the TCP/IP stack or the file system block layer, is usually located in the kernel. This is in contrast to microkernel-based systems, where both the driver and its client execute as user-level processes. Interaction between the kernel and a user-level process involves fewer context switches than IPC between two processes, and hence can in principle be faster. In practice, however, current monolithic OSs are not optimised for such interaction and would require massive design effort to implement it efficiently.

User-level device drivers in Linux The communication overhead can be mitigated using buffering techniques, similar to those described in the previous section. Leslie et al. [LCFD+05] implemented this approach using an rbufs-like communication mechanism. Their performance evaluation shows up to 7% throughput degradation and up to 17% CPU overhead, compared to in-kernel drivers, for hard disk and Gigabit Ethernet controllers. These encouraging results were obtained at the cost of increased I/O latency, which was not measured in the paper.

Nooks The Nooks system, developed by Swift et al. [SBL03], adds memory protection to kernel-mode drivers in Linux. Drivers are encapsulated inside light-weight protection domains, called nooks. A nook executes in the kernel mode and has read access to all kernel memory, but can only write to its private heap and to device memory regions. Communication between a nook and the rest of the kernel is mediated by the Nooks isolation manager, which intercepts all function calls in both directions and makes sure that the driver can safely write data structures passed to it by reference. This is achieved by either maintaining a synchronised copy of the data structure inside the nook or by forwarding every write access to the data structure to the kernel.

The isolation manager detects three types of driver failures: illegal memory accesses, data format violations, and temporal failures. Illegal memory accesses are detected using hardware protection mechanisms. Data format violations are detected by validating parameters passed by the driver to the kernel. Finally, temporal failures are detected using timeouts.

The isolation manager also keeps track of all kernel objects allocated or accessed by the driver. In case of a driver failure, the Nooks recovery manager releases all kernel resources

held by the driver, creates and initialises a new instance of the driver. This basic recovery scheme ensures that the system remains in a consistent state after the failure; however existing users of the driver observe the failure, since the original copy of the driver becomes unavailable and its state is lost.

Nooks was later extended to support fully transparent recovery using a mechanism called shadow drivers [SABL04]. A shadow driver is a shim that is attached to an instance of a Nooks driver and passively intercepts its communication with the OS, recording information required to recover the driver in case of a failure. At the time of recovery, the shadow driver attaches to the new driver instance and switches to the active mode, where it generates a sequence of requests that recreate the state of the driver before the failure. During this time, all requests sent to the driver by the OS are blocked. OS calls issued by the driver are either handled by the shadow driver or forwarded to the OS. Once the recovery is complete, the shadow driver switches back to the passive mode. Shadow drivers are device independent and need to be developed once for a class of devices.

One issue with Nooks is the complexity of implementing the isolation and recovery infrastructure. An isolation manager is required to understand the semantics of all driver-OS interactions in order to perform argument validation and synchronise shared data structures. Likewise, a shadow driver in the active mode assumes the role of the OS and must correctly implement the semantics of all OS functions used by the driver. Since a Linux driver has access to a large number of internal kernel functions, Nooks runtime components are required to incorporate semantics of a substantial subset of the kernel interface. While the feasibility of this approach has been demonstrated on a small number of drivers, building a complete and robust solution appears to be problematic due to the size and complexity of this subset.

Another limitation of the Nooks architecture is the significant performance overhead induced by frequent protection boundary crossings. Linux drivers perform a large number of kernel calls, all of which must be intercepted and forwarded to the kernel. For example, the reported overhead for the Intel Pro/1000 Gigabit Ethernet adapter driver is 10% throughput degradation and up to 80% increase in CPU utilisation.

Microdrivers The Microdrivers [GRB+08] architecture offers a tradeoff between safety and performance by executing non-performance-critical parts of the driver at the user level. This includes management and configuration functions, which account for 65% of the driver code. The kernel-mode portion contains performance-critical data path functions.

An existing Linux driver can be turned into a microdriver by automatically partitioning it into user and kernel-mode components. To this end, the user specifies a set of performance-critical driver entrypoints that must execute in the kernel. A static analysis tool determines driver functions that are transitively called from these entrypoints. The resulting set of functions is included in the kernel-mode part of the microdriver; the remaining code will execute in the user mode. Data structures shared between the kernel and the user part are replicated and synchronised using a mechanism similar to the one implemented in the

Nooks isolation manager. The current implementation of microdrivers does not support fault recovery; however the authors point out that the Nooks recovery infrastructure can be easily adapted to this purpose.

The Microdrivers architecture was further enhanced in the Decaf [RS09] project. Decaf provides a mechanism for manually converting the user-level portion of a microdriver to the Java programming language. This enables the driver to take advantage of the Java type and memory safety properties. The conversion is performed incrementally, on a function-by-function basis. Data sharing and communication between the C and the Java parts of the driver is managed by another instantiation of the isolation manager mechanism, which marshals method arguments and maintains synchronised copies of shared data structures across language boundaries.

Microdrivers improve driver safety while introducing negligible performance overhead. The main limitation of this approach is that a significant portion of the driver remains in the kernel.

User-level drivers in mainstream products Due to performance problems and complexity associated with user-level device drivers, mainstream systems are slow to move in this direction. At the moment, most major OSs, including Windows, Linux and Mac OS X provide only a limited support for implementing certain types of drivers, such as drivers for USB-based storage and webcam devices, at user level [Nak02, Mic07, App06].

3.1.4 User-level device drivers in paravirtualised systems

In recent years user-level device drivers have found new applications in the context of the virtualisation technology. Specifically, they come into play in configurations where multiple virtual machines share the hardware resources of the host system. In such settings, access to I/O devices from a guest OS can be enabled using two approaches. The first approach, implemented for instance in VMWare [SVL01], is device virtualisation, where the VMM intercepts all I/O operations issued by the guest OS and emulates the behaviour of the device as if it was exclusively owned by the guest. The emulated device does not have to be the same as the actual physical device connected to the host. The advantage of device virtualisation is that it does not require any changes to the guest OS. It is, however, associated with very high performance overhead and therefore cannot be used in applications that require high I/O throughput.

The second approach is device paravirtualisation. The idea is to present the guest OS with an I/O interface that differs from the actual hardware interface, but can be efficiently implemented in the hypervisor. The guest OS must be extended with drivers that handle this interface. Thus, paravirtualisation is not fully transparent to the guest.

Device paravirtualisation has been implemented, for instance, in the Xen [BDF+03] VMM. Xen consists of a small privileged hypervisor (not unlike a microkernel [HWF+05]),

which hosts multiple guest OSs running with user-level privileges. Only one guest has direct access to I/O hardware. It runs an instance of Linux, which contains drivers for all peripheral devices. This dedicated guest is called Domain0. All other guests use the devices via Domain0. To this end they contain stub drivers, which appear as normal device drivers to the guest OS, but internally communicate with Domain0, rather than the actual hardware.

Communication with drivers in Domain0 involves control and data transfer across process boundaries, which poses the same performance problems as user-level drivers in microkernel-based systems. Not surprisingly, Xen addresses these problems using techniques borrowed from microkernels. Specifically, it implements a communication mechanisms that uses circular buffers in shared memory and asynchronous notifications in a manner similar to rbufs.

Nevertheless, the performance overhead of paravirtualisation remains quite high, generating up to 300% increase in CPU utilisation in the worst case of handling a stream of incoming data packets [MST+05]. Menon et al. [MCZ06] and Santos et al. [STJP08] have proposed a number of optimisations that reduce this overhead to 97% and to as low as 25% for network controllers that support multiple receive queues. Lagar-Cavilla [LCTSdL07] et al. obtained similar encouraging results for paravirtualised graphics adapters.

Härtig et al. [HLM+03] implemented device paravirtualisation in their workhorse architecture, using the L4 Fiasco [Hoh02] microkernel as a hypervisor. Unlike Xen, where all drivers reside in the Linux kernel running in Domain0, the workhorse architecture encapsulates every driver in its own protection domain. Reuse of existing Linux drivers is enabled by an emulation library that is linked against device drivers providing them with the illusion of running inside the Linux kernel. The CPU overhead of device paravirtualisation in the workhorse architecture runs up to 200% for network devices.

LeVasseur et al. [LUSG04] implemented a similar architecture on top of L4::Pistachio [Sys03], but instead of using an emulation library they run each driver inside a separate virtualised instance of Linux. They report a 100% CPU overhead.

3.2 Software-based fault isolation

Software fault isolation (SFI) enables safe execution of untrusted modules in the application or kernel address space, without relying on hardware protection mechanisms. The safety properties enforced by SFI include memory safety and control transfer safety (i.e., the module can only invoke a pre-defined subset of kernel entry points). SFI operates at the binary code level and therefore can be applied to programs written in low-level languages like C or Assembly.

SFI was first proposed by Wahbe et al. [WLAG93] and was originally used to isolate untrusted extensions for user-level programs. Wahbe's implementation performs binary code transformation on the untrusted module, inserting runtime checks that cannot be circumvented by the module. To perform these checks efficiently, restrictions are imposed on the

application memory layout. The address space of the application is divided into contiguous segments so that all addresses within a segment share a unique pattern of upper bits. An untrusted module is only allowed to execute code in its code segment and access data in its data segment. Immediate control transfer instructions, as well as immediate loads and stores can be statically checked to comply with these constraints. Indirect jumps and memory accesses are modified to use a small subset of registers that are not loaded by the untrusted code, but only by inserted code, which applies a bit mask to the address, ensuring that it falls into the right segment.

Inter-module communication is mediated by trusted stub routines. The code segment of a module contains an indirection table of jump instructions to stub routines. These jump instructions provide the only way to transfer control outside the module. Stub routines implement control transfer and argument marshalling to the callee module.

Performance results reported by Wahbe et al. show that SFI increases CPU utilisation by up to 39%. One drawback of this architecture is that it complicates data sharing between modules. Sharing can only be implemented at the page granularity using memory remapping. Another limitation is that this solution relies on a RISC load and store architecture and cannot be easily extended to CISC architectures, including x86.

Both limitations are overcome in the XFI [EAV+06] architecture, which enforces memory safety and control flow integrity without imposing any restrictions on the memory layout. Control-flow integrity is enforced using static analysis and runtime guards. In order to statically guarantee the integrity of data that influences the control flow, such as function return addresses, the program execution stack is split into a scoped stack, which stores all such sensitive data and can never be accessed using computed addresses, and an allocation stack, which stores all other stack data.

Memory safety is enforced using runtime guards, which verify that every computed memory access falls into one of contiguous memory regions assigned to the untrusted module. The cost of the check depends on the number of available regions, which can vary at runtime. Control-flow integrity properties enforced by XFI ensure that memory access guards cannot be circumvented.

The overhead of XFI on various application benchmarks is within 125%. In particular, it was applied to two Windows kernel drivers; however both of them were drivers for pseudo-devices implemented in software, so these results may not be representative of real hardware drivers.

The Vino [SESS96] OS uses the MiSFIT [SS98] SFI architecture to isolate and recover from failures in kernel extensions. MiSFIT implements similar techniques to those introduced by Wahbe et al., but adapts them to the IA32 *instruction set architecture (ISA)*. Like Wahbe's SFI architecture, MiSFIT does not allow data sharing between modules. Access to OS data structures is performed via trusted accessor functions. Vino detects the following types of failure: memory safety violations and control flow integrity violations (using MiSFIT), livelocks and deadlocks (using timeouts), and resource hoarding (using per-module

resource accounting).

Vino supports a transactional model of computation for kernel extensions, using transaction rollback as the fault recovery mechanism. Every invocation of an extension starts a new transaction. Whenever a transaction calls an OS function that modifies the state of a kernel object, the modification is recorded in the transaction's log and the object is locked until the transaction commits or aborts. For each such mutator function, Vino provides an undo function, which reverts its effect. If a transaction aborts due to an extension failure, Vino calls all undo functions from its log and unloads the faulty extension. While the Vino fault tolerance mechanism can in principle be adjusted to support recovery from driver failures, this has not been implemented.

The main limitations of the Vino architecture are its very high performance overhead (more than a factor of 10 for some extensions), and the requirement to implement an undo function for every mutator function callable from extensions, which increases kernel size and complexity.

3.3 Fault removal using static analysis

Static analysis tools detect software defects by analysing the source or binary code of the program, without actually executing it. Recent improvements in static analysis techniques have led to the development of tools, like Metal [ECCH00], SLAM [BBC^{+}06], and Blast [BHJM07], capable of detecting many common errors in device drivers written in C and C++. This section surveys some of these improvements.

Early static analysis tools, such as Lint [Joh77], performed syntactic analysis of the program in search of common programming bugs, such as reading uninitialised variables or passing a wrong number of arguments to a function. Most of these checks have been integrated into modern compilers.

Much more powerful analyses become possible with the help of model checking techniques. Model checking refers to verifying a formal model of a system against a specification of its required properties. The model is usually written in a formal language, such as Promela [Hol03] or LOTOS [LOT89]. The specification can be a temporal logic [Pnu77] formula or a program in the same language. In both cases it defines constraints on the ordering of a subset of operations performed by the model. Alternatively, the property specification can be incorporated into the system model in the form of assertions, in which case it describes a subset of illegal system states. A model checker explores all possible executions of the model and reports if any of them violate the specification. In order to efficiently explore very large state spaces encountered in real-world problems, most model checkers use symbolic methods, which manipulate compressed representations of the state space, e.g., in the form of a *binary decision diagram (BDD)* [Bry86].

Some model checkers, including Metal, SLAM, and Blast operate directly on the source code of the system, rather than on its formal model. This approach allows checking prop-

erties of the system without having to manually construct the model. However, it is harder to implement than conventional model checking. Any non-trivial program has an enormous state space, which cannot be explored using brute-force techniques. In conventional model checking, the user who builds the model of the system abstracts away any state information that is not essential to its correctness with respect to the properties of interest. If the resulting model is too complex for the model checker to explore within available time and memory resources, the user relaxes the abstraction to make it susceptible to formal analysis. If the model is too abstract, i.e., is missing information required to establish its correctness, the user refines it by adding the missing details.

Source-level model checkers must choose the right abstraction without human involvement. The simplest approach, implemented for instance in Metal, is to use a fixed abstraction. This may result in a high rate of false positives, especially when checking complex properties. Another solution is to use counterexample-guided abstraction refinement algorithms [CGJ+03] to adjust the abstraction automatically. Such algorithms start with computing a very rough approximation of the program behaviour. For example, they may abstract away all program variables and assume that any program branch can always be taken. Next, the algorithm checks whether the desired property holds for this abstract model and, if it does not, it attempts to produce a counterexample that violates the property. If such a counterexample exists for the original program, the algorithm terminates, otherwise it refines the model to avoid the spurious counterexample. The process iterates until a valid counterexample is found or the model is shown to satisfy the property.

One problem that arises when applying model checking at the source code level is the need to deal with complex semantics of programming languages. This is particularly difficult for system programming languages, which do not enforce type and memory safety and allow explicit pointer manipulation. The memory state of programs written in such languages cannot be represented by a set of typed values and must be modelled as an array of bytes. Furthermore, multiple pointers inside the program may point to the same memory location. Such aliases cannot always be reliably identified, making it hard to reason about the memory state of the program. In practice a good approximation can often be obtained using alias analysis [Das00].

In applying model checking to device drivers, an important question is: what properties should be checked? Modern model checking techniques scale well with the size of the analysed program, but are highly sensitive to the complexity of properties that this program is checked against. Complex properties, which depend on large subsets of the program state variables, reduce the effectiveness of abstraction techniques, forcing fixed abstractions to yield large numbers of false positives, and increasing the runtime of algorithms that use abstraction refinement. Hence, the best results can be obtained by capturing the most common driver errors using simple properties.

Such properties can be generic or specific to device drivers. The Metal model checker has been successfully used in detecting generic errors in system code [CYC+01]. Some

examples of rules checked by Metal are: "release acquired locks; do not double-acquire locks", "do not dereference user pointers", and "check potentially NULL pointers returned from routine". While these rules are applicable to all kernel components, Metal found 3 to 7 times more errors per line of code in device drivers compared to the rest of the kernel, which illustrates the relatively poor quality of the driver code.

Driver-specific properties describe rules of the kernel API for device drivers and are used to detect OS protocol violations (see Section 2.8). The SLAM model checker and the commercial SDV [BCLR04] tool based on it have been used to detect many such violations in Windows device drivers. Examples of rules checked by SLAM are: "drivers do not return STATUS_PENDING if IoCompleteRequest has been called", "if a driver calls another driver that is lower in the stack, then the dispatch routine returns the same status that was returned by the lower driver", and "drivers mark I/O request packets as pending while queuing them". SLAM allows users to specify additional rules using a domain-specific language [BR01].

Input to the SLAM model checker consists of the source code of the driver, rules to be checked, and a model of the OS environment. The last component simulates the order and arguments with which the OS invokes driver entry points, as well as behaviour of OS APIs invoked by device drivers.

The developers of SLAM report that one of the main problems they encountered was the absence of well-defined OS API rules. According to Ball et al.,

> "The rules for these APIs were hard to get right. Often, kernel experts in these areas would disagree with one another about subtle points in the rules. As a result, we would develop a rule and have to iterate many times with experts, showing them errors found by SFV and then refining the rules if the errors were false. This took a tremendous amount of time and energy."

The lack of well-defined rules constitutes a problem not only for model checking, but also for manual driver development, since in the absence of a clear definition of correct behaviour writing correct drivers is problematic.

Other examples of static analysis tools that check driver-specific properties include Coccinelle [PLHM08] and Carburizer [KRS09]. Coccinelle addresses the problem of collateral evolution in Linux device drivers [PLM06], where changes made to internal kernel interfaces introduce errors in previously correct drivers. Coccinelle provides a language to specify such changes and a static analysis tool that identifies locations in the driver source code that need to be updated and generates a source patch that implements the necessary updates.

Carburizer is an automatic driver hardening tool that detects situations where the driver makes assumptions about device behaviour that can compromise its safety and inserts code to fix these situations.

Unfortunately, existing literature on model checking for device drivers does not discuss limitations of the approach. In particular, both SLAM and Metal report large numbers of

errors found by checking simple rules, including the ones cited above, however they do not give examples of more complex properties that cannot be efficiently validated using these tools.

Compared to runtime isolation techniques discussed in the previous sections, static analysis has the advantage of finding errors without introducing runtime overhead. On the other hand, static analysis only finds a subset of errors that can be detected at runtime. For example, memory safety of a C program cannot be checked statically.

In some cases, synergy between the two approaches is possible. The CCured [NCH+05] system uses static analysis to check memory safety of as many load and store instructions as possible and relies on SFI to check the remaining ones at runtime. This way, fewer runtime checks are required compared to conventional SFI, which reduces the runtime cost of isolation.

3.4 Language-based fault prevention and fault tolerance techniques for device drivers

High-level programming languages impose syntactic and semantic restrictions on the program, making certain types of errors impossible or difficult to express and allowing other errors to be detected by the compiler or the language runtime.

Memory safety is the most important property that can be enforced using language support. In low-level languages, like C, which allow explicit pointer manipulation, proving memory safety is undecidable. Languages that disallow explicit pointer manipulation eliminate invalid memory accesses due to incorrect pointer arithmetic. Type-safe languages additionally allow memory safety violations resulting from unsafe type casting to be detected by the compiler. Finally, languages with automatic storage management eliminate memory allocation errors, such as use after free and double free, and enable the detection of array boundary violations using a combination of compile-time and runtime checks. A combination of these techniques guarantees memory safety.

For performance reasons, most OSs are written in low-level languages. The increasing complexity of system software encourages the use of more advanced software development technology, including high-level languages. The rest of this section surveys research on the use of high-level languages for operating system and in particular device driver development.

Historical systems The use of high-level languages in OS development was pioneered in the Burroughs B5000 series of computers [Org73]. The B5000 featured a stack-oriented architecture, which enabled efficient execution of programs written in high-level languages. All system and application software for B5000 was developed in Algol.

An Algol-based programming language, called Mesa, was used at Xerox PARC to develop the Pilot [RDH+80] OS for personal computers, including device drivers. Mesa pro-

grams are structured as collections of modules, with module boundaries enforced by the compiler. The next version of the Mesa language, called Cedar [SZBH86], supported automatic storage management through garbage collection. Cedar was used to develop an OS of the same name.

Lisp machines [Moo87, Dus88, Deu73] are a family of computer architectures designed for efficient execution of Lisp programs. The entire software stack running on these computers, including device drivers, was developed in Lisp.

Java OSs Java is a popular object-oriented virtual-machine-based garbage-collected programming language. Several research and commercial OSs have been implemented in Java. These systems do not support hardware-based memory protection and only run applications written in Java.

JavaOS [Mad96] from Sun is based on a bare-metal implementation of the Java VM. Apart from the language run-time, the entire OS, including device drivers, is written in Java. All processes in JavaOS share a single object namespace. This design makes it difficult to implement security and resource isolation.

KaffeOS [BHL00] and JX [GKB01] address this problem by combining the Java runtime with traditional OS isolation techniques based on processes. A process is a unit of resource accounting. Every process uses a private garbage-collected heap for memory allocation. Process boundaries are enforced by an extended language runtime, without using MMU-based protection. JX supports inter-process communication via remote procedure invocations. Since processes have separate heaps, invocation arguments must be copied to the callee's heap. KaffeOS allows processes to create shared heaps and populate them with code and data. Communicating processes interact by accessing the shared heap. The prototype implementation of KaffeOS runs on top of Linux, with device drivers implemented inside the Linux kernel. In contrast, JX runs directly on the hardware. Drivers are implemented in Java and can take advantage of JX protection facilities.

An important limitation of Java-based OSs is that they only run programs written in Java, which limits their use to narrow application domains. In addition, the use of Java incurs high performance overhead. JX reports 50% slowdown on file system benchmarks, compared to Linux.

House House [HJLT05] is an experimental OS developed in Haskell, running on top of a bare-metal implementation of the Haskell runtime. In particular, House device drivers are written in Haskell. Unlike Java-based OSs, House can run programs written in any programming language inside hardware protection domains. Reliability benefits of Haskell are not limited to type and memory safety. Being a purely functional programming language, Haskell facilitates formal reasoning about program behaviour. House uses a version of Haskell that supports code annotations with formal logic formulae [Kie02]. These annotations can be used as the basis for checking program correctness using static analysis,

theorem proving, testing, or manual code inspection. For example, in case of device drivers, properties of interest include device and OS protocol compliance. Although the approach looks promising, no actual evidence of applying formal techniques to verifying the House OS have been published so far.

Vault Language support can be used to enforce higher-level properties than memory safety. In Section 3.3 we saw that checking even simple OS protocol rules in C-like languages requires complex algorithms and involves manual intervention to identify false positives. Incorporating these rules in a language type system enables much more efficient decision procedures.

This approach was implemented in the Vault [DF01] system programming language. In Vault, software protocols are specified by associating a set of states with a data type and annotating functions with pre- and post-conditions describing how the function changes the states of its arguments. The programmer must explicitly keep track of the state of every program variable by including the state of the variable, in its declaration. Furthermore, he must explicitly mark all aliases to the same variable. This way, the compiler is able to verify protocol compliance by performing only local type checking, e.g., it must check that arguments passed to a function satisfy its preconditions.

Vault has been used to develop Windows 2000 device drivers, where the Vault type system allowed enforcing resource management rules between the driver and the kernel. An example of such a rule is: for every I/O request received from the OS the driver either completes the request immediately by calling the `IoCompleteRequest` function or passes the request down the driver stack by calling the `IoCallDriver` function, or marks the request for delayed processing using `IoMarkIrpPending`.

A major drawback of the Vault approach is that it puts substantial burden on the programmer. In essence, it requires the programmer to incorporate a proof of protocol compliance in the source code of the program. In addition, Vault only keeps track of statically allocated resources. When dealing with dynamic data structures, the program must use anonymisation operations to prevent the compiler from keeping track of variable states, thus suppressing static protocol enforcement.

SafeDrive The SafeDrive [ZCA+06] system aims to improve the reliability of device drivers written in C by making the C language more secure. It defines a simple language extension that allows adding size annotations to pointer types, protecting the driver code against out-of-bound memory reads and writes. These annotations are enforced via a combination of static and runtime checks. SafeDrive does not track memory deallocation and hence does not protect against dangling pointer dereference.

SafeDrive also provides a mechanism for recovering faulty device drivers, similar to lightweight transactions mechanism in Vino: the SafeDrive runtime keeps track of changes made by the driver to the system state and applies compensatory functions upon failure.

The reported overhead of SafeDrive, in terms of CPU utilisation, is 23% in the worst case.

Cyclone The Cyclone programming language [JMG+02] is a dialect of C that uses a combination of language extensions, compile-time, and runtime support to achieve type and memory safety. Cyclone language extensions include tagged unions, fat pointers, region-based memory allocation, and data-flow annotations. These extensions enable compile-time checking of many pointer operations. More complex checks are performed at runtime.

One interesting feature of the language is that it supports two mechanisms for safe memory reclamation: garbage collection and data-flow annotations. The former supports automatic memory management at the cost of some performance overhead, while the latter requires additional manual effort, but does not incur any performance overhead.

Cyclone shares most of the C syntax and uses the C calling convention and data structure layout, which makes it easy to integrate modules written using Cyclone into an OS kernel written in C.

Several systems, including STP [PWW+03] and OKE [BS02], have used Cyclone as the basis for the construction of safe kernel extensions. They enhance the type and memory safety facilities of Cyclone with additional mechanisms that enable safe sharing of kernel data structures, restricted access to kernel services, and control over the use of CPU, and other runtime resources.

In addition to high CPU overhead (up to 300% for CPU-intensive workloads), a major drawback of Cyclone is its heavy reliance on manual annotations, which complicate the use of the language.

Singularity The Singularity [FAH+06] OS combines the advantages of most of the language-based techniques described above. The entire system is written in the Sing# language, with type and memory safety being enforced by the Sing# compiler and virtual machine. User-level programs written in unsafe languages execute inside hardware protection domains. Protection and resource isolation among OS components are achieved by running every component, including device drivers, as a separate software-isolated process with its own private heap. Inter-process communication is limited to messages exchanged through typed channels, with data passed through a shared non-garbage-collected heap. Similar to Vault, Sing# allows the formal specification of a channel's communication protocol and the static enforcement of protocol compliance. Furthermore, Singularity drivers can use Sing# facilities for specifying program invariants to formalize and statically enforce device protocol constraints.

3.5 Preventing and tolerating device protocol violations

Most driver reliability techniques surveyed in the previous sections deal with memory access violations and OS protocol violations. This section describes techniques for preventing and tolerating device protocol violations.

Modern I/O bus architectures provide hardware support for isolating the most severe failures in the form of an IOMMU (see Section 2.2.1). An IOMMU can be used to prevent device drivers within a single OS from accidentally overwriting kernel or application state by programming the device to perform a DMA transfer to an invalid location [BYMK+06, Lin]. In a virtualised environment, an IOMMU allows the VMM to protect guest OSs from erroneous or malicious I/O operations issued by other guests [BYMK+06, WRC08].

Willmann et al. [WRC08] evaluated the impact of using an IOMMU on the system performance. They found that creating a single-use mapping for every I/O transaction increases CPU utilisation by 30% for I/O-intensive workloads. This overhead can be significantly reduced by reusing mappings.

Williams et al. [WRW+08] developed a purely software-based replacement for IOMMU for the Nexus [SWSS05] microkernel-based OS. Their solution consists of a Reference Validation Mechanism (RVM), which intercepts and validates all device register accesses issued by the driver. The RVM recognises when the driver configures the device for a DMA operation and checks whether memory regions pointed to by the provided DMA descriptor are owned by the driver. The RVM is configured per-device using a Device Safety Specification (DSS), which describes the register layout of the device and its DMA protocol. The RVM is located in the kernel and cannot be circumvented by device drivers, which run at the user level. The RVM has modest impact on I/O bandwidth and latency and introduces a CPU utilisation overhead of around 30%. The main limitation of this architecture is the requirement to develop a safety specification for every supported device.

Several domain-specific languages have been developed to simplify the implementation of the low-level device interface of a driver, including Devil [MRC+00], NDL [CE04], HAIL [SYKI05], and Laddie [Wit08]. These languages allow declarative specification of the device register and memory layouts. They also provide a means to formally define valid sequences of register accesses in the form of finite state machines or temporal logic formulae. The driver developer creates such specifications based on informal documentation provided by the device manufacturer, usually in the form of a device datasheet.

Given these specifications, the language compiler generates device accessor functions in C or C++. These functions encapsulate bit-level arithmetics and low-level protocols involved in using device registers. Constraints that cannot be encapsulated inside a single function are enforced at runtime by adding appropriate checks to accessor functions. For example, a check may assert that a given function is only invoked when the device is in a certain state. In principle, these constraints could also be enforced using static analysis; however none of the above systems has implemented this approach.

3.6 Fault prevention through automatic device driver synthesis

A radical approach to improving device driver reliability consists of automatically generating the driver implementation based on a formal specification of its required behaviour. The specification is written in a high-level domain-specific language and is therefore expected to contain much fewer defects than manually developed drivers. In addition, the specification is more readily susceptible to formal analysis than low-level C code, which allows further reduction of errors due to incorrect specification.

There exist two approaches to automatic driver synthesis: hardware/software co-design and standalone synthesis. In the co-design approach [BCG+97], the designer specifies the structure and behaviour of the system in the form of communicating finite state machines. Once this abstract specification has been validated using simulation or formal verification, the designer may choose which components of the design should be implemented in hardware and which in software. The hardware components are generated in the form of *field-programmable gate array (FPGA)*s or *application-specific integrated circuit (ASIC)*s, whereas the software components are translated into a program in a low-level programming language.

Existing co-design techniques are intended for generating simple embedded microcontrollers and their drivers. Furthermore, they do not support OS-based drivers, i.e., drivers generated using these techniques run without OS support or with a minimal real-time kernel. These limitations are not inherent to the co-design methodology and may be overcome in the future, making it a promising approach to improving the quality of device drivers.

In the standalone synthesis approach [WMB03, ZZC03, KSF00, OOJ98], the device and its drivers are developed separately. Devices are usually designed using a *hardware description language (HDL)*; however the driver developer does not typically have access to the HDL specification. Instead, the device manufacturer publishes an informal description of the device interface in the form of a datasheet and sample code. Based on this documentation, the driver developer creates a formal specification of the driver using a domain-specific language. The data definition part of the specification describes register and memory layouts of the device, similarly to Devil and related languages. The behavioural part of the specification describes the functionality of the driver in the form of communicating finite state machines. This specification is automatically translated into a driver implementation in C or another programming language.

This approach has several advantages over implementing the driver in the conventional way. First, communicating state machines capture the driver behaviour more naturally than procedural or functional languages. A device driver is in essence a controller for the device state machine. Its state and operation reflect state and operation of the device and can be most naturally modelled as another state machine. Second, specification languages used for driver synthesis have simpler semantics than C (mainly due to their simpler memory models) and are therefore easier to verify formally. For example, Wang and Malik [WM03]

statically check properties like termination and deadlock freedom. More complex checks, such as adherence to OS protocols can also be checked in principle, however no such results have been reported in the literature. Finally, a formal specification can abstract away some low-level implementation details, such as CPU endianness, bus access method, OS API for timers and interrupts, etc. These details can be added by the synthesis tool. It should be noted that similar advantages can be achieved in conventional drivers by wrapping the corresponding functionality in library functions.

Practical merits and limitations of standalone driver synthesis tools are not yet fully understood. Published results are based on early prototypes, use simple embedded devices as examples, and do not report any performance results. Nevertheless, it is a promising approach that deserves further exploration.

3.7 Conclusions

Techniques surveyed in this chapter prevent, detect, and tolerate many types of software defects. However, all of them have serious limitations. In particular, runtime fault isolation, based on either hardware or software mechanisms, introduces substantial performance overhead. While, in principle, it can be used to isolate and recover all types of memory access and OS protocol violations, in practice the lack of well-defined driver-OS protocols in modern systems complicate the implementation of the isolation and recovery infrastructure.

Static analysis techniques do not introduce performance overhead; however they are only capable of detecting a limited class of errors. Their use is also complicated by the ambiguity of OS protocols, which makes it difficult to identify properties that the driver should be checked against.

Prevention and isolation techniques for device protocol violations rely on the driver developer to create a formal specification of the device interface, based on an informal device datasheet. Errors introduced at this stage cannot be detected automatically and lead to software defects in the driver.

In summary, existing solutions do not provide full protection against driver bugs. Therefore, an approach that helps driver developers produce better code, containing fewer bugs, has the potential to improve both driver and overall system reliability. The remainder of this thesis presents such an approach.

Chapter 4

Root-cause analysis of driver defects

This thesis aims to develop an improved device driver architecture and development process that will enable driver writers to create drivers with fewer defects. To achieve this, we must first identify root causes of defects in existing drivers, i.e., the aspects of driver development that introduce complexity and provoke errors. This chapter presents such a *root-cause analysis (RCA)*.

Software RCA [Car98, MJHS90] involves understanding human behaviour and is therefore difficult to formalise. It relies on informal analysis of a large number of defects by the developers, who are in the best position to determine whether a particular defect is provoked by ambiguous requirements, inadequate documentation, complexity of a particular algorithm or interface, a flow in the development methodology, or whether it is an unforced coding error.

The result of this analysis is documented and added to the source repository along with the fix for the error. Unfortunately, neither device manufacturers nor OS vendors make this information publicly available. One notable exception are open-source systems like Linux and FreeBSD, where a complete development history, including detailed explanation of most software patches and the defects that they eliminate, is freely accessible. In particular, the study of driver defects presented in this chapter is based on the Linux kernel development repository [Bit].

Other approaches to software defect analysis include statistical defects modelling [MIO87] and the Orthogonal Defect Classification methodology [CBC+92]. These approaches are quantitative in nature. They rely on predefined generic defect classifications and only require superficial analysis of every particular defect. As such, they are much easier to implement in industrial settings compared to RCA, but only provide limited information about types of defects that occur in a particular software product. Since our goal is to develop in-depth understanding of defects that are specific to device drivers, we rely on the more laborious RCA methodology.

Name	**Description**	**LOC**	**Defects**
USB drivers			
rtl8150	rtl8150 USB-to-Ethernet adapter	827	16
catc	el1210a USB-to-Ethernet adapter	710	2
kaweth	kl5kusb101 USB-to-Ethernet adapter	925	15
usb net	generic USB network driver	914	45
usb hub	USB hub	2234	67
usb serial	USB-to-serial converter	989	50
usb storage	USB Mass Storage devices	1864	23
IEEE 1394 drivers			
eth1394	generic ieee1394 Ethernet driver	1413	22
sbp2	sbp-2 transport protocol	1713	46
PCI drivers			
mthca	InfiniHost InfiniBand adapter	11718	123
bnx2	bnx2 Ethernet adapter	5412	51
i810 fb	i810 frame buffer device	2920	16
cmipci	cmi8338 soundcard	2260	22
	Total	33899	498

Table 4.1: Linux device drivers used in the study of driver defects. The table shows the size of each driver in lines-of-code and the number of defects found in the driver during the period covered by the source repository.

4.1 Methodology

I selected 13 device drivers for the study (Table 4.1), aiming to make the selection as diverse as possible, to ensure that it provides a representative sample of driver defects. In particular, I consider drivers for different device classes, such as Ethernet controllers, storage devices, video and audio adapters, an InfiniBand host controller, a USB hub, and a USB-to-serial converter. These devices are connected to the host via three different I/O buses: USB, IEEE 1394 (FireWire), and PCI. Furthermore, the selected devices cover the entire complexity spectrum, from simple devices, e.g., the CATC Ethernet controller, to high-end ones, represented by the Mellanox InfiniBand controller.

This selection contains 4 drivers for similar USB-to-Ethernet adapter devices (top 4 entries in Table 4.1). The defect statistics for these drivers will be used in the evaluation of the Dingo architecture presented in Chapter 5.

In order to identify the root causes of defects in the selected drivers, I analysed the complete history of updates made to their source code during the six-year period from 2002 to 2008 (which is the complete period covered by the repository). I only considered updates that fixed incorrect behaviours and ignored all other changes, including performance optimisations, functionality extensions, and modifications to keep the driver up-to-date with the

evolving kernel API.

Each update was analysed to understand the exact defects that it addressed, how these defects were fixed, and what factors provoked them. This information was obtained based on the source code of the driver, the patch file that implemented the update, and the developer's comments attached to it. In all, I analysed 498 defects. This does not include a small number of defects (less than 5%) that were not documented clearly enough to establish their precise nature. All the defects covered by the study were classified into four broad categories, based on the root cause of the defect:

1. *Defects caused by the complexity of device protocols.* These defects occur if the driver developer does not fully understand the details of the software interface of the device or fails to correctly express them in the driver code.

2. *Defects caused by the complexity of OS protocols.* These defects occur when the driver incorrectly implements or uses one of its interfaces with the OS (Figure 2.5).

3. *Concurrency defects.* These defects include race conditions and deadlocks due to incorrect handling of multiple threads of control inside the driver.

4. *Generic programming faults.* This category includes defects that are not specific to device drivers. This includes defects related to the use of a low-level programming language or to the inherent complexity of implementing a large software component.

This taxonomy is similar to the taxonomy of failures presented in Section 2.8. The principle difference is that failures represent incorrect behaviours observable at runtime, whereas defects represent algorithmic errors that may cause runtime failures. The correspondence between defects and failures is not one-to-one. A defect in the implementation of the OS interface may cause an OS protocol violation; however it may also cause a memory access violation. The latter may happen, for instance, if the driver programmer does not expect a certain OS request to occur in a particular state and therefore does not allocate the memory resources required to handle the request. Likewise, a concurrency bug may lead to a race between two threads executing inside the driver, which can manifest itself as invalid ordering of device or OS invocations or in a memory access fault. In general, any type of defect can lead to any type of failure.

Table 4.2 summarises the results of the study by showing the number of defects of each type found in each driver. Defects caused by the complexity of device protocols comprise the biggest group, with the remaining defects distributed evenly among the three other categories.

4.2 Example

Before going into a detailed discussion of the various types of defects, consider an example of how a Linux driver defect is analysed and classified into a particular category.

Driver	**Total defects**	**Root cause**			
		Device prot. complexity	**OS protocol complexity**	**Concurrency**	**Generic**
rtl8150	16	3	2	7	4
catc	2	1	0	1	0
kaweth	15	1	2	8	4
usb net	45	16	9	6	14
usb hub	67	27	16	13	11
usb serial	50	2	17	13	18
usb storage	23	7	5	10	1
eth1394	22	6	6	4	6
sbp2	46	18	10	12	6
mthca	123	52	22	11	38
bnx2	51	35	4	5	7
i810 fb	16	4	5	2	5
cmipci	22	17	3	1	1
Total	498	189 (38%)	101 (20%)	93 (19%)	115 (23%)

Table 4.2: Classified counts of driver defects. The maxima in each row are highlighted.

The defect in question was found in revision 1.142 of the `drivers/usb/storage/usb.c` file, which belongs to the USB storage driver, and was fixed in revision 1.143. The following comment by the author of the driver describes the problem:

> The problem was introduced recently along with autosuspend support. Since `usb_stor_scan_thread()` now calls `usb_autopm_put_interface()` before exiting, we can't simply leave the scanning thread running after a disconnect; we must wait until the thread exits. This is solved by adding a new struct completion to the private data structure.

The first step in analysing the defect involves studying the source code of the driver to understand its overall structure and in particular parts related to the defect.

The USB storage driver is a unified driver for a range of USB mass storage devices such as flash drives, hard drives, and SD card readers. All such devices are required by the USB mass storage class specification [USB08b] to support the *Small Computer System Interface (SCSI)* command set. Therefore the USB storage driver uses the USB bus transport interface to access the device and exports the SCSI host device-class interface to the Linux kernel (Figure 4.1).

During initialisation, the driver registers itself as a SCSI host driver and spawns a separate kernel thread to carry out SCSI-device scanning. The thread function (`usb_stor_scan_thread()`) runs holding a power management lock on

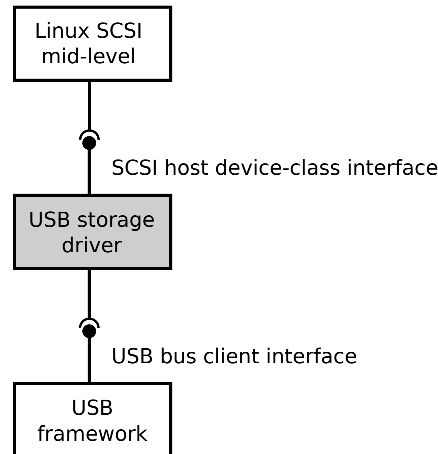


Figure 4.1: Linux USB storage driver interfaces.

the underlying USB interface. This is required to make sure that the device is not automatically suspended while being scanned. The lock is released using the `usb_autopm_put_interface()` Linux function before returning from the thread function.

The race condition mentioned in the defect description occurred if the driver was shut down (e.g., because the device was disconnected from the USB bus) while the scanning was in progress. The disconnect handler calls the `scsi_host_put()` function in the Linux SCSI layer, which unregisters the SCSI host and deallocates the data structure holding the driver's private state. Address of this data structure is passed to the `usb_autopm_put_interface()` function when it is invoked by the scanning thread, which results in an invalid memory access. The error is fixed by simply waiting for the scanning thread to terminate before calling `scsi_host_put()`.

At this point it is clear that the defect was provoked by the complexity of dealing with concurrency inside the driver. This is also confirmed by the author's description of the defect, which suggests that the confusion was caused by thread synchronisation issues.

4.3 Defects caused by the complexity of device protocols

This and the following sections discuss specific types of defects that fall into each of the four categories. In particular, defects related to the complexity of device protocols can be further classified into the following groups.

Value defects The driver and the device exchange data, including device descriptors, configuration commands, and I/O transfer descriptors, via memory and registers. Defects related to handling of device data include endianness errors, incorrect use of register bit fields, sending invalid data values to the device, and incorrectly interpreting values received from the device.

For example, when configuring a newly connected device, the USB hub driver retrieves a device descriptor, which specifies device capabilities and supported configurations. In interpreting the device descriptor, the hub driver made a false assumption that the number of USB interfaces supported by the device in each configuration must be greater than zero. However, the assumption contradicted the USB specification and could result in a misconfigured device.

Ordering defects In order to correctly control the device, the driver must keep track of the internal state of the device state machine. Errors occur when the programmer's mental model of this state machine diverges from its actual implementation. These errors may lead to the driver issuing a sequence of commands to the device that fails to meet its intended goal or even leaves the device in an invalid state.

A representative example of such a defect occurred in the USB hub driver, which erroneously tried to resume a suspended hub port when a new device was connected to it. However, the resume command is not allowed in this state, since the suspended status of the port is automatically cleared once there is a new connection. The error returned by the resume command prevented Linux from using the given hub port.

Timing defects Device state transitions can be triggered by the passage of real time. The driver simulates such transitions using timeouts. Forgetting to put a timeout statement in the appropriate place or using an incorrect timeout value is likely to lead to failure of subsequent commands issued to the device.

For instance, the USB hub specification mandates a delay in the hub port connection sequence after the device attached to the port has been reset and before any commands are issued to it; however the hub driver implementation ignored this requirement and attempted control transfers with the device immediately after reset, resulting in unpredictable outcomes.

Data races The device and the driver may engage in shared-memory communication using DMA and memory-mapped I/O. Access to shared memory regions is synchronised using interrupts, device registers, and memory barriers. Incorrect use of synchronisation can lead to a race between the driver and the device.

For instance, the bnx2 network controller driver contained a race condition where the driver acknowledged packets in the device receive ring before actually processing them. The device could overwrite the corresponding entries in the ring with new ones, causing lost packets and memory leaks.

Table 4.3 shows statistics for the various types of defects caused by the complexity of device protocols.

Type of defect	**# defects**	**Relative frequency**
Value defects	116	61%
Ordering defects	52	28%
Timing defects	15	8%
Data races	6	3%
Total	189	100%

Table 4.3: Defects caused by the complexity of device protocols.

4.4 Defects caused by the complexity of OS protocols

Defects related to the complexity of OS protocols can be classified into ordering and value defects. Ordering defects occur when the driver developer misunderstands or fails to correctly implement rules on the ordering of invocations exchanged by the driver and the OS. As a result, he may either invoke OS functions in the wrong order or incorrectly handle an unexpected invocation from the OS.

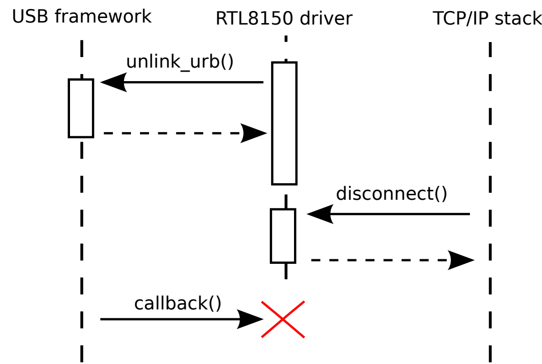
Different classes of drivers implement similar interfaces with the OS, which enables further classification of ordering defects. For instance, any driver must implement initialisation, shutdown, power management, and data transfer operations. Accordingly, we distinguish the following types of ordering defects.

Defects in the implementation of initialisation, shutdown, and configuration protocols These protocols define sequences of invocations exchanged by the driver and the OS during device initialisation, shutdown, and configuration. This includes establishing a connection with the underlying bus driver, allocating bus resources required to access the device, registering the device-class interface with the OS, responding to configuration requests, and handling of shutdown requests and hot unplug events.

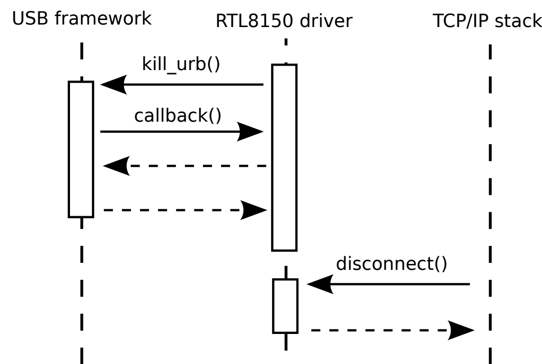
One example of a related defect was found in the USB storage driver. During startup, the driver first called the `scsi_add_host()` function to register itself as a SCSI host driver and then the `scsi_set_device()` function, which associates the underlying device with the SCSI host. The Linux SCSI layer, however, assumes that the SCSI host already has a device associated with it when `scsi_add_host()` is called, and in some cases attempts to access the device immediately, leading to an illegal memory access and a kernel panic.

Defects in the implementation of the data protocol During normal operation, the driver exchanges streams of I/O requests and responses with the OS and the underlying bus driver. This is the most frequently used and therefore the most thoroughly tested part of the driver functionality. Nevertheless, it may still contain defects, which typically manifest themselves in uncommon corner-case situations.

For instance, one such defect occurred in the rtl8150 USB-to-Ethernet adapter driver



(a) A failure scenario.



(b) Behaviour of the fixed driver.

Figure 4.2: A defect in the rtl8150 controller driver.

when the driver tried to cancel an outstanding USB transfer (e.g., because the transfer timed out). To this end, the driver called the `unlink_urb()` function, which attempts to cancel the transfer immediately, but if it is not possible because the transfer is currently being handled by the USB controller, it schedules the transfer for deferred cancellation and returns to the caller. When the transfer is finally cancelled, the USB framework calls a completion callback provided by the driver. In the rtl8150 driver this could happen after the driver terminated and all its state was deallocated, which caused the completion callback to crash the kernel. Figure 4.2a shows the possible sequence of interactions between the driver and the OS that was not correctly handled by the driver.

The fixed version of the driver replaced the call to `unlink_urb()` with `kill_urb()`, which always cancels the transfer before returning even if this requires blocking, as shown in Figure 4.2b.

Defects in the implementation of resource ownership protocols The Linux kernel uses reference counting to control the life time of various resources. The driver may obtain a reference to a resource either explicitly, by calling an operation that increments its reference count, or implicitly, by receiving a pointer to the given object in a request. Forgetting to release a reference to an object or trying to access the object without holding a reference to it may result in a resource leak or a kernel crash.

Type of defect	# defects	Relative frequency
Ordering defects		
Initialisation, shutdown and config. protocols	21	21%
Data protocols	9	9%
Resource ownership protocols	8	8%
Power management protocol	8	8%
Subtotal	46	46%
Value defects		
Incorrect use of OS data structures	48	48%
Returning invalid error code	7	7%
Subtotal	55	55%
Total	101	100%

Table 4.4: Defects caused by the complexity of OS protocols.

.

Defects in the implementation of the power management protocol These defects are related to the interaction between the driver and the OS during handling of device suspend and wakeup requests.

For example, the following defect found in the suspend function of the cmi8338 sound-card driver prevented the driver from correctly resuming the device after the suspension. During the suspend sequence, the driver called the `pci_set_power_state()` function, provided by the Linux PCI framework, to remove power from the corresponding PCI slot, followed by a call to the `pci_save_state()` function, which stores the content of the device's PCI configuration registers in memory. The intention was to restore the configuration registers during wakeup. However, being invoked in this order the functions did not produce the expected result, since the content of the configuration registers was lost once the power was removed from the device and therefore the values stored by `pci_save_state()` were bogus.

The top part of Table 4.4 shows statistics for the various types of ordering defects.

We distinguish two types of value defects in the implementation of the driver-OS interface:

Incorrect use of OS data structures The driver and the OS exchange and share data, including device descriptors and I/O request descriptors. Failing to properly allocate and initialise fields of a data structure before passing it to the OS or incorrectly interpreting data received from the OS may lead to severe runtime consequences.

An example of such defect occurred in the `port_query` function of the InfiniHost InfiniBand controller driver. This function returns a port descriptor, which describes properties of a physical port of the controller. The function failed to initialise the field of the descriptor which reports maximal data segment size supported by the port, preventing the

OS from correctly using the port.

Returning invalid error code This type of defect includes situations where the driver fails to correctly report the status of an I/O operation to the OS, e.g., it indicates successful completion of a failed request or returns a status code that does not correctly reflect the cause of the error.

Statistics of value defects are shown in the bottom part of Table 4.4.

4.5 Concurrency defects

As mentioned in Section 2.5, a device driver can be invoked in the context of multiple concurrent threads. Defects related to concurrency can be classified based on the types of concurrent activities that cause the given race condition or deadlock.

Races and deadlocks in the data path These defects result in incorrect synchronisation between functions responsible for streaming data to and from the device. For example, the bnx2 Ethernet controller driver contained a race condition between the function that queued a new packet in the transmit ring and the interrupt handler that removed completed packets from the ring. Both functions accessed common hardware data structures and could leave them in an inconsistent state, blocking the transmit operation of the controller.

Races and deadlocks in the configuration path These defects result in incorrect synchronisation among initialisation, shutdown, and configuration functions and between these functions and data path functions. The latter occur when the driver gets a configuration or a shutdown request while handling a stream of data requests. For example, a race between the packet transfer and the shutdown functions of the rtl8150 USB-to-Ethernet adapter driver could cause the driver to keep sending packets to the controller after the controller was disabled.

Races and deadlocks in the power management functions These defects result in incorrect synchronisation between power management functions and any of the above functions (i.e., configuration and data path functions). For example, a race in the bnx2 driver between the device suspend function and the data path function responsible for resetting the controller when a packet transfer timed out could cause the controller to be suspended while the reset operation was in progress.

Races and deadlocks in the hot unplug handler Hot unplug events occur asynchronously to all other driver activities, which makes them particularly difficult to handle correctly. For example, a hot unplug event may occur while the device is being suspended. While unlikely, such a situation is possible, for instance if the user unplugs a USB device

immediately after closing the laptop lid. In Section 4.2 we saw another example of a hot unplug race, which occurred when the driver received a disconnect notification before the device was completely initialised.

Calling a blocking function in an atomic context These defects occur when the driver calls a potentially blocking OS function while running in the primary interrupt context. Usually this happens if the programmer does not realise that the given driver entry point can be invoked from the primary interrupt handler or when he does not realise that the OS service that he uses can block.

For example, one such defect was introduced when fixing the incorrect use of the `unlink_urb()` function described in Section 4.4. One instance of `unlink_urb()` invocation that was replaced with `kill_urb()` was located in the transmit timeout handler function. This handler is invoked by the Linux kernel in the context of a timer interrupt and is therefore not allowed to block. Calling the `kill_urb()` function from this handler could deadlock the kernel. Ironically, the defect was addressed by changing the call to `kill_urb()` to `unlink_urb()`, thus reintroducing the original defect described in Section 4.4.

Using uninitialised synchronisation primitives Drivers rely on various types of synchronisation primitives, such as mutexes, semaphores, completions, etc., to avoid race conditions. Linux requires every synchronisation object to be initialised before use, using the appropriate initialisation function or macro. Forgetting to do this may cause a kernel crash or deadlock.

Imbalanced locks Every successful acquisition of a mutex or a semaphore must be balanced by the appropriate release operation. Forgetting to release the lock is likely to result in a deadlock.

Calling OS services without appropriate locks Some kernel services require the caller to acquire a specific lock before using the service. For instance, calls to the Linux SCSI layer must be protected by a lock associated with the SCSI host structure. Forgetting to acquire this lock created a race condition in the USB storage driver.

Table 4.5 provides statistics for the various types of concurrency-related defects. It shows that most of these defects are provoked by events that happen relatively infrequently, such as a a hot-unplug notification or a configuration request that arrives concurrently with another configuration or data request.

Type of defect	# defects	Relative frequency
Races and deadlocks in the configuration path	29	31%
Races and deadlocks in the hot-unplug handler	26	28%
Calling a blocking function in an atomic context	21	23%
Races and deadlocks in the data path	7	8%
Races and deadlocks in power management functions	5	5%
Using uninitialised synchronisation primitives	2	2%
Imbalanced locks	2	2%
Calling OS services without appropriate locks	1	1%
Total	93	100%

Table 4.5: Concurrency defects.

Type of defect	# defects	Relative frequency
Control flow defects	62	54%
Memory allocation errors	30	26%
Typos	5	4%
Missing return-value checks	16	14%
Arithmetic errors	2	2%
Total	115	100%

Table 4.6: Generic defects.

4.6 Generic programming faults

This category of defects includes common coding errors, such as memory allocation errors (memory leaks, use-after-free, double-free, etc), typos, missing function return-status checks, arithmetic errors, and control-flow defects that cannot be attributed to handling of either device or OS protocols. Table 4.6 shows statistics for these defects.

4.7 Limitations of the study

Several factors limit the representativeness and accuracy of the results produced by the study presented in this chapter. First of all, the study only considers Linux drivers. Statistics of driver defects in other systems, in particular Windows and Mac OS, would provide a more general picture. For example, since Windows drivers are typically developed by the device vendor, it is logical to expect defects related to handing the device interface to be less common and defects related to handling the OS interface to be more common in Windows drivers than in Linux drivers. Unfortunately, these systems use closed development processes, which do not allow free access to their source code, not to mention the complete development history. Therefore, the study was limited to open-source systems, of which Linux is the most important representative.

Second, the study does not take into account the severity of analysed defects, i.e., whether the given defect leads to a system crash, a permanent or transient device failure, degraded performance, etc. In many cases, the only reliable way to establish possible consequences of the defect is by means of an experiment. However, performing such experiments for a sufficiently large subset of defects covered by the study was infeasible within the scope of the thesis project. Besides, many defects can lead to different failures in different scenarios, which makes exhaustive analysis even harder.

Third, in analysing defects related to the complexity of OS and device interfaces, it would be helpful to distinguish four different situations: (1) the defect was introduced because the required behaviour was not described in the appropriate specification (the device datasheet or OS API documentation), (2) the specification was inaccurate, (3) the developer did not understand the specification, and (4) the developer understood the required behaviour, but failed to implement it correctly. This kind of analysis would have provided additional useful input for the study. Unfortunately, most driver patches in Linux do not contain sufficient information to perform this analysis.

Fourth, this study does not account for the interplay between different root causes. In a device driver, interaction with the device, interaction with the OS, and synchronisation are closely intertwined. Therefore, it is likely that reducing the complexity of one of these tasks will lead to the reduction of all types of defects.

Finally, the study was focused on technical problems causing driver defects and did not take into account other factors, such as potential problems in the driver development process and various social factors (e.g., communication between developers).

4.8 Conclusions

Despite the above limitations, the study provides useful findings that help to direct the efforts in improving driver reliability to where they are likely to achieve the most impact. Figure 4.3 represents the main results of the study by showing the relative frequency of the four categories of driver defects.

Firstly, the study has established that defects caused by the complexity of device protocols comprise the biggest group of driver defects (Figure 4.3). This is an expected result, since managing the device is the primary purpose of any device driver and therefore it is natural that much of the driver complexity is concentrated in the code responsible for device interaction.

More surprisingly, defects related to the interaction between the driver and the OS, namely, defects caused by the complexity of OS protocols and concurrency defects, are as common as device-related defects. These defects can potentially be reduced via a better design of the driver-OS interface. The next chapter presents one approach to constructing such an improved interface.

Figure 4.4 compares the relative frequency of different types of defects in USB,

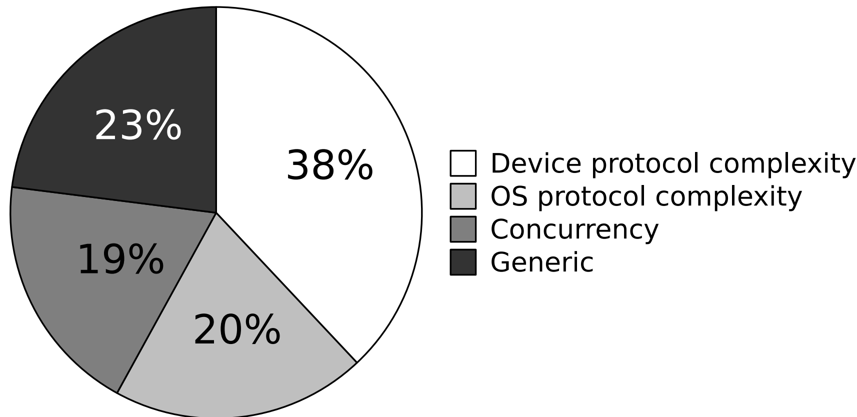


Figure 4.3: Relative frequency of the four categories of driver defects.

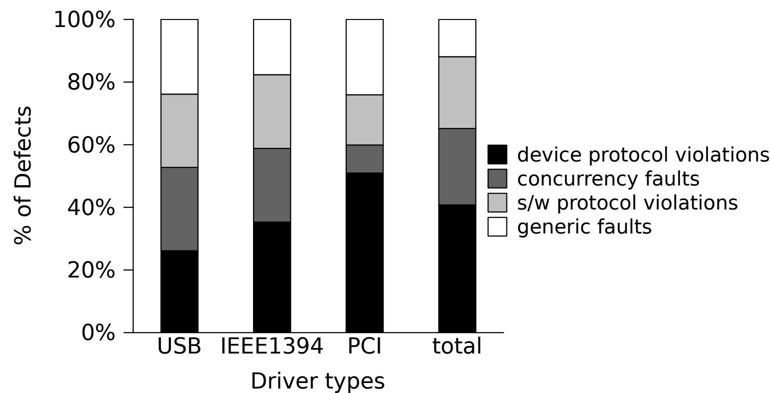


Figure 4.4: Summary of defects by bus.

IEEE 1394, and PCI drivers. It shows that OS-related defects are much more common in USB and IEEE 1394 drivers than in PCI drivers. Unlike the PCI bus, USB and IEEE 1394 buses are not memory mapped. Communication with the device is based on message passing. Even simple operations, such as reading or writing device registers are accomplished by preparing and sending a message to the device and waiting for a reply message. This complicates the bus transport interface provided by the OS and increases the amount of concurrency that the driver must handle, since reply messages are delivered asynchronously by a separate kernel thread.

In the rest of this thesis I develop techniques to mitigate each of the root causes identified in this study. In particular, the following chapter concentrates on improving the design of the driver-OS interface to reduce defects related to concurrency and OS protocol complexity. The automatic driver synthesis approach developed in Chapter 6 eliminates generic programming faults and substantially reduces device protocol defects.

Chapter 5

Device driver architecture for improved reliability

The driver defect study presented in the previous chapter has revealed areas where better OS support could improve driver reliability. In particular two categories of defects are directly related to how the driver interacts with the OS: defects caused by the complexity of OS protocols and concurrency defects. Together, these defects constitute 39% of the defects in our study, and are clearly a significant source of problems for drivers.

In this chapter I propose a new device driver architecture, called Dingo, that simplifies interaction with the OS and allows driver developers to focus on the main task of a driver: controlling the hardware. Dingo achieves this via two improvements over traditional driver architectures. First, Dingo reduces the amount of concurrency that the driver must handle by replacing the driver's traditional multithreaded model of computation with an event-based model. This model eliminates the majority of concurrency-related driver defects without impacting the performance. Second, Dingo provides a formal language, called Tingu, for describing software protocols between device drivers and the OS, which avoids confusion and ambiguity, and helps driver writers avoid defects in the implementation of these protocols.

Dingo does not attempt to provide solutions to deal with the other types of defects identified (i.e., defects caused by the complexity of device protocols and generic programming faults). These defects are provoked by factors that lie beyond the influence of the OS and will be addressed as part of the automatic driver synthesis approach presented in Chapter 6.

I present an implementation of the Dingo architecture in Linux, which consists of a set of wrappers that make drivers developed in compliance with the Dingo interface appear as regular Linux drivers to the rest of the kernel. This enables Dingo and conventional Linux drivers to coexist, providing a gradual migration path to more reliable drivers. Experimental evaluation of the Dingo driver architecture shows that it eliminates most synchronisation errors and reduces the likelihood of protocol violations, while introducing negligible performance overhead.

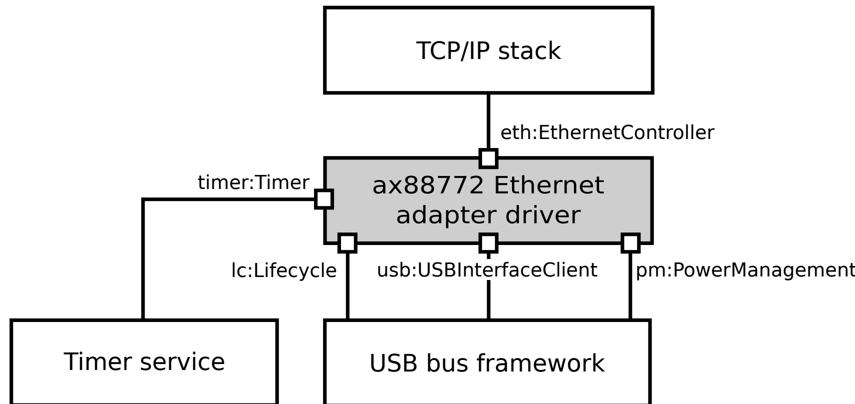


Figure 5.1: Dingo driver for the ax88772 USB-to-Ethernet adapter and its ports. White squares at the driver boundaries indicate ports.

5.1 Overview of Dingo

A Dingo driver is a software object that communicates with the OS over ports. A port is a bidirectional communication point that defines a set of methods that must be implemented by the driver and the OS. Every driver interface shown in Figure 2.5 is represented by one or several Dingo ports.

Execution of a driver is triggered by invoking a method through one of its ports. Dingo guarantees atomicity of driver invocations, i.e., at most one method of the driver can be running at a time.

Each driver port is associated with a protocol, which specifies a behavioural contract between the driver and the OS. It defines the methods that can be invoked over that port as well as constraints on the ordering, timing, and arguments of invocations. Protocols are specified as part of the OS driver framework and describe services that different types of device drivers must provide to the OS as well as services provided by the OS to device drivers.

When implementing a device driver, the developer declares its ports and chooses a protocol for each port among those supported by the OS. He must then provide an implementation of all incoming methods associated with the driver's ports.

Figure 5.1 shows the Dingo driver for the ax88772 USB-to-Ethernet adapter, its ports, and the parts of the OS that the driver interacts with. The driver provides services to the OS via `Lifecycle`, `PowerManagement` and `EthernetController` protocols. It uses the `USBInterfaceClient` protocol exported by the USB bus framework, and the `Timer` protocol exported by the OS timer service. In the figure, each port is labeled with the name of the port (`lc`, `pm`, etc.) and the name of the protocol that it implements.

The Dingo architecture can be implemented as a self-contained OS driver framework or it can be built as an extension of an existing driver framework, providing an improved interface for developing device drivers within that framework. We have implemented the latter

approach in Linux by constructing adapters between the multithreaded driver interfaces defined by Linux and the Dingo protocols. This approach allows Dingo and native Linux drivers to coexist in the same system, offering a gradual migration path to more reliable device drivers.

5.2 An event-based architecture for drivers

The concurrency problems highlighted in Chapter 4 are not unique to device drivers. In a multithreaded environment, concurrent activities interleave at the instruction level, leading to non-determinism and the explosion of the number of possible executions. As a result, many programmers are generally ineffective in dealing with threads, which makes multithreading the leading source of bugs in a variety of applications.

An alternative to multithreaded concurrency is event-based concurrency. In the event model, a program executes as a series of event-handlers triggered by events from the environment. Reactions to events are atomic; concurrency is achieved by interleaving events belonging to different activities. Thus, the event model replaces instruction-level interleaving with event-level interleaving. Event serialisation guarantees that the state of the program observed at the start of an event can be modified only by the current event handler. This simplifies reasoning about the program behaviour and reduces the potential for race conditions and deadlocks.

Comparison of threads versus events has been the subject of lasting debate in the systems community [LN78, AHT+02, vBCB03]. The main point of consensus in this debate is that different applications may favour different models.

One observation in favour of an event-based approach for drivers is that modern device drivers are already partially event-based for performance reasons. In particular, the handling of all performance-critical I/O requests is split into two or more event handlers: upon receiving a request, the driver adds it to the hardware queue and returns control to the caller immediately, without waiting for the request to complete. Later, it receives a completion event from the device and invokes a completion callback provided by the OS. Such asynchronous handling of requests enables improved performance by parallelizing I/O and computation. This interaction pattern of splitting long-running operations into request and completion steps is typical for event-based systems. Thus, while current drivers do not fully exploit the advantages of the event-based model, this style of programming is already familiar to driver developers.

Below I present an event-based architecture for device drivers and show that it eliminates most concurrency-related defects and can be implemented in a way that neither complicates driver development nor incurs a performance penalty. Thus it should be the preferred model.

In the Dingo event-based architecture events are delivered to the driver by invoking its methods through ports. Typical events include configuration and data transfer requests from

the client and interrupt notifications delivered by the bus framework on behalf of the device. Methods are executed in an *atomic* and *non-blocking* manner. The atomicity guarantee means that no new method invocation can begin while a previous method is running. This prohibits simultaneous invocations of the driver by different threads, as well as recursive calls from the same thread. Note that the atomicity constraint does not prevent the driver from being invoked from different threads or different CPU cores, as long as all invocations are serialised.

Inside the body of the method the driver may perform computation and invoke OS methods through ports. It is not allowed to block or busy wait, because such behaviour would delay the delivery of subsequent events.

This event-based design affects driver development in two ways. First, since Dingo serialises execution of the driver at the method level, there is no need for synchronisation among concurrent message handlers. Therefore, Dingo drivers do not use spinlocks, mutexes, wait queues, or other thread-based synchronisation primitives. However, the driver may have to synchronise tasks that span multiple method invocations. For example, when handling two long-running I/O requests that use the same hardware resource, the driver must ensure that execution of the second request begins after the first request completes. This is typically achieved by tracking the status of the shared resource using a state variable. As will be shown in Section 5.4, the number of cases where such synchronisation is required is much smaller than in multithreaded drivers. The event-based architecture also simplifies the use of I/O memory barriers. In particular, barriers that order accesses to I/O memory from different CPUs can be moved out from the driver into the framework. On architectures that require barriers to order I/O memory accesses on a single CPU, the programmer is still responsible for correctly placing such barriers.

Second, since the driver is not allowed to block, there is no way for it to freeze its execution waiting for an event, such as a timeout or a hardware interrupt. Instead, the driver has to return from the current method and later resume execution in the context of the corresponding event-handler method. Driver interfaces must be designed to take into account this constraint. Specifically, any driver interface operation that may involve waiting must be split into a request method provided by the driver and a completion callback method provided by the OS. The driver may invoke the completion callback from the request method, if it is able to process the request immediately, or from another event-handler method, if the request requires waiting for external events.

Splitting a single operation into a chain of completions may lead to complex and unmaintainable code—the effect known as stack ripping [AHT+02]. Figure 5.2 illustrates the problem by comparing a stylised implementation of the `probe()` function in a conventional Linux driver and equivalent non-blocking Dingo driver code.

The `probe()` function initialises the device hardware. The Linux version of the function writes configuration settings into the device configuration registers and then waits for the device to complete internal initialisation by waiting for 10 milliseconds and then check-

```
1 int probe(...) {
2   /*Write config registers*/
3   ...
4
5   do {
6     msleep (10);
7     /*Read status registers*/
8     ...
9   } while (/*condition*/);
10  return 0;
11 }
```

(a) Linux

```
1 void probe(...) {
2   /*Write config registers*/
3   ...
4   timer->setTimeout(timer,10);
5   driver->state = PROBE_TIMEOUT;
6   return;
7 }
8
9 void timeout(...) {
10  switch(driver->state) {
11    case PROBE_TIMEOUT:
12      /*Read status registers*/
13      ...
14      if (/*condition*/)
15        lc->probeComplete(lc);
16      else
17        timer->setTimeout(timer,10);
18      break;
19    ...
20  }
21 }
```

(b) Dingo

Figure 5.2: The stack ripping problem in event-based drivers. Listing (a) shows a blocking implementation of the probe request in a conventional Linux driver. Listing (b) shows equivalent non-blocking Dingo code.

ing values in device status registers. The last steps are performed in a loop until the status registers indicate that the initialisation has completed.

In Dingo, the equivalent behaviour is spread across two driver methods. The `probe()` method writes the configuration registers and then calls the `setTimeout()` method of the OS timer service to schedule a 10-millisecond timeout (line 4). Here, the `timer` variable is a pointer to a port of the driver (see Figure 5.1), which is passed as the first argument to all methods invoked through the port. Before returning from the method, the driver sets its `state` variable to `PROBE_TIMEOUT` (line 5).

When the timeout expires, the OS notifies the driver via the `timeout()` method. This method picks up the logical flow of execution where the method that called `setTimeout()` dropped it. Since `setTimeout()` can be called from multiple places in the driver, the value of the `state` variable is used for disambiguation (line 10). The driver reads the values of status registers in lines 12 and 13 and checks whether the hardware initialisation has completed in line 14. If the check succeeds, then the driver signals the completion of the probe request to the OS by calling the `probeComplete` method of

```
1 reactive probe(...) {
2   /*Write config registers*/
3   ...
4
5   do {
6     timer->setTimeout(timer,10);
7     AWAIT(timeout);
8     /*Read status registers*/
9     ...
10  } while (/*condition*/);
11  return 0;
12 }
```

Figure 5.3: Implementation of the probe method (Figure 5.2) using the extended C syntax.

the `lc` port (Figure 5.1). Otherwise, it schedules another timeout (line 17).

The problem with this implementation is that it obfuscates the logical control flow, requires the introduction of an auxiliary variable, and is twice as long as the equivalent Linux code. The cause of the problem is that by splitting a function into multiple fragments we lose compiler support for automatic stack management and mechanisms that rely on it, such as control structures and local variables. The resulting complexity can easily exceed the complexity of synchronisation, thus cancelling out the advantages of the event-based architecture.

Fortunately, one can combine an event-based model of computation with automatic stack management. One way to achieve this has been demonstrated by the Tame [KKK07], and Clarity [CCJR07] projects, both of which developed C language extensions providing event-based programs with a sequential style of programming.

We have implemented a similar approach in Dingo. Our language extension, described in Section 5.2.1, provides several constructs that enable driver control logic to be expressed in the natural sequential way, avoiding stack ripping. The extension is implemented by a simple source-to-source translator described in Section 5.2.2. Using this extension, the example in Figure 5.2b can be rewritten as shown in Figure 5.3. The `AWAIT` statement in line 7 stops the logical execution flow until the timeout event occurs. It expands into code that saves the current execution context of the `probe()` method and returns control to the OS. When the timeout event is generated, it restores the execution context and resumes execution from the line following `AWAIT`.

The use of the language extension is optional. Many drivers contain only a few blocking operations and so stack ripping does not constitute a serious problem for them. For example, this is the case for most PCI-based device drivers. Such drivers can be written in pure C. In contrast, all USB drivers use blocking extensively, since every interaction with a USB device requires waiting for a response message. The only practical way to implement such

drivers in Dingo is using the extended version of C presented below.

5.2.1 C with events

The proposed C language extension allows a programmer to express event-based program logic in a sequential way by providing first-class constructs for event-based communication. It introduces the notion of a reactive function in addition to normal C functions. A reactive function executes in a separate logical thread of control and can send and receive (react to) events from other threads. Both events and threads are language-level objects and are distinct from Linux kernel threads and driver interface events.

Unlike kernel threads, which are scheduled preemptively, language threads are scheduled cooperatively. Only one language thread inside the driver can be runnable at a time. A switch to another thread can only occur when the current thread blocks waiting for an event. Thus, this form of multithreading preserves the atomicity of execution offered by the Dingo architecture.

A driver method can be declared as a reactive function. When such a method is invoked by the kernel, a new language thread is spawned and keeps executing in the context of the invoking kernel thread until blocking waiting for an event. During its execution the language thread may emit one or more events, waking up other language threads inside the driver. These threads are activated one by one until all of them become blocked when there are no more outstanding events. At this point, control is returned to the invoking kernel thread. Thus, while the driver can internally spawn multiple language threads, all these threads execute in the context of a single kernel thread, the one that is currently invoking the driver. This implementation results in a hybrid model of computation where language threads inside the driver are scheduled cooperatively with respect to each other, but preemptively with respect to regular kernel threads.

Table 5.1 lists the new constructs in C with events. The `EVENT` statement declares a new event variable. The first argument is the name of the variable; the remaining arguments describe event arguments, which describe the content of the event. Like normal C variables, events can be declared within any local or global scope and passed as arguments to functions. In addition, they can be used in `EMIT`, `AWAIT`, and `CALL` constructs described below. Events are implemented as C structures whose fields correspond to event arguments. The `EVENT_TYPE` statement declares a C type whose instances are events.

The `reactive` keyword is used to declare a reactive function. Invoking such a function creates a language thread, which terminates asynchronously to the calling language thread, therefore a reactive function cannot have a return type.

The `EMIT` statement emits an event. Event parameters must be set by the caller in advance. `EMIT` does not indicate the recipient of the event; any language thread can request to receive the event using `AWAIT`. The same event can be emitted multiple times; however once it is marked as emitted, subsequent `EMIT`'s have no effect until the event is received

Syntax	Description
`EVENT(name, t1 f1,..., tn fn)`	Event instance declaration
`EVENT_TYPE(name, t1 f1,..., tn fn)`	Event type declaration
`reactive f(...){...}`	Reactive function declaration
`EMIT(e)`	Emit an event
`AWAIT(e1,...,en)`	Wait for one of several events
`IF(e1)...ELIF(e2)...ELSE`	Check which event has been received
`CALL(f(...),e)`	`{f(...); AWAIT(e);}`

Table 5.1: C with events syntax.

by a thread.

The `AWAIT` statement blocks the language thread waiting for one out of a group of events. Note that `AWAIT` waits for specific event instances rather than for types of events. If one of the specified events is already marked as emitted, the thread continues without blocking. In the current implementation, only one thread can be waiting for an event. It is the programmer's responsibility to make sure that no two threads await the same event. The `IF...ELIF...ELSE` construct is used to determine the event received by the last `AWAIT` issued by the thread. Arguments of `IF` and `ELIF` clauses are evaluated in order and if one of them is marked as emitted, the corresponding branch is taken. Otherwise, the `ELSE` branch is taken.

The `CALL` statement implements a common pattern of calling a reactive function and waiting for a completion event from it.

5.2.2 Implementation of C with events

The source-to-source translator that handles C with events introduces two new types of run-time objects in the generated C code: continuations and events. A continuation is a data structure that represents the state of a reactive function that is blocked waiting for an event. It stores the values of its local variables and the program counter. It also contains a trampoline function that restarts the execution of the reactive function from the state described by the continuation.

As mentioned above, an event is a C structure that contains event parameters. It also contains two fields used by the generated code: a flag indicating whether the event is in the in-flight state (i.e., has been emitted but has not yet been delivered) and a pointer to the continuation of a reactive function awaiting this event. The latter can be NULL if no function is waiting for the given event at the instance when it is issued.

The `EMIT` statement marks the event as emitted and puts it on the event queue. The `AWAIT` statement checks whether one of the events listed as its arguments has been emitted and, if so, removes it from the queue and continues executing the following instructions. Otherwise, it allocates a new continuation and saves the state of the function in it. It stores

the address of this continuation in all events that the function is waiting for. Finally, it invokes the dispatcher function to determine which thread should run next. To this end, the dispatcher selects the first event in the queue that has a continuation associated with it and activates this continuation.

C-with-events language threads run on the stack of the Linux kernel thread that has invoked the driver. When a language thread blocks, the state of the currently executing reactive function is saved in a continuation, which is allocated from the heap, and another language thread is selected to run on the same stack. When there are no more runnable threads inside the driver (i.e., all threads are blocked), the dispatcher returns control to the kernel thread that invoked the driver. Execution of the driver resumes with the next methods invocation.

Note that blocking of a language thread does not cause the Linux thread in the context of which the driver is running to block. Instead, control is immediately passed either to another language thread or back to the Linux thread. Thus, driver invocations can be safely performed in any context, e.g., while the calling thread is holding a lock or is executing in a primary interrupt handler, without introducing the risk of a deadlock.

5.2.3 Dingo on Linux[1]

This section presents our implementation of the Dingo runtime framework for Linux. The framework is designed to allow driver developers to take advantage of the Dingo architecture when writing new device drivers, while being able to run existing drivers without any changes. To this end, the framework is designed as a collection of adapters that perform the translation between Linux and Dingo driver interfaces. These adapters are attached to a Dingo driver at runtime, making it appear as a normal Linux driver to the rest of the system (Figure 5.4).

Linux and Dingo driver interfaces differ in two aspects. First, Linux interfaces allow multithreading, whereas Dingo requires all driver invocations to be serialised. Second, Linux driver interfaces include both asynchronous and synchronous methods, whereas Dingo interfaces are completely asynchronous. An asynchronous method may return control to the caller before the I/O operation started by this method completes. The completion of the operation is signalled via a separate callback function. A synchronous method always completes the requested operation before returning to the caller, which may involve blocking.

In order to translate between Linux's and Dingo's driver interfaces, the Dingo framework associates a queue and a mutex with each driver. The queue is used to serialise requests delivered to the driver. Access to the queue is protected by a spinlock. The mutex is used to ensure that at most one thread can enter the driver. A request is delivered to the driver as follows. The Linux thread performing the request places the request on the queue. Then it

[1]The implementation described in this section was developed in collaboration with Dr. Peter Chubb.

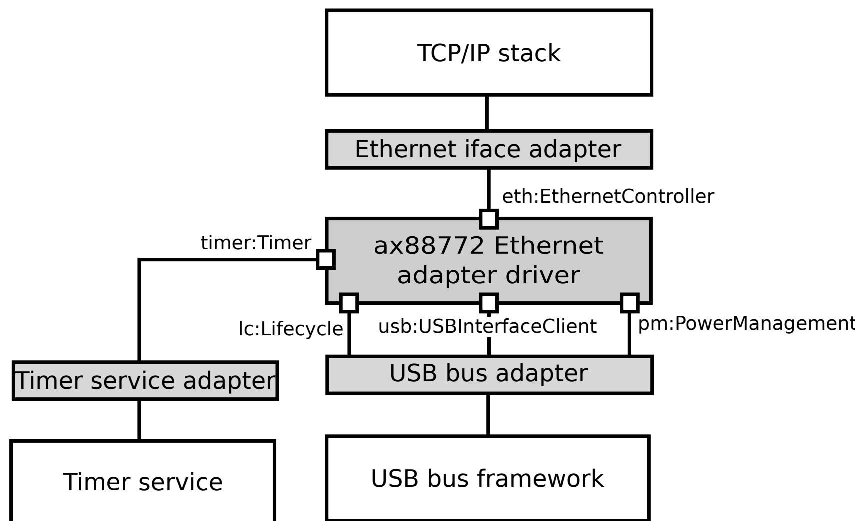


Figure 5.4: Dingo interface adapters using the example of the ax88772 controller driver.

attempts to acquire the mutex. If this succeeds, then it dequeues and delivers all requests in the queue to the driver, one by one. Otherwise, the calling thread relies on the thread that is currently holding the driver lock to deliver the request.

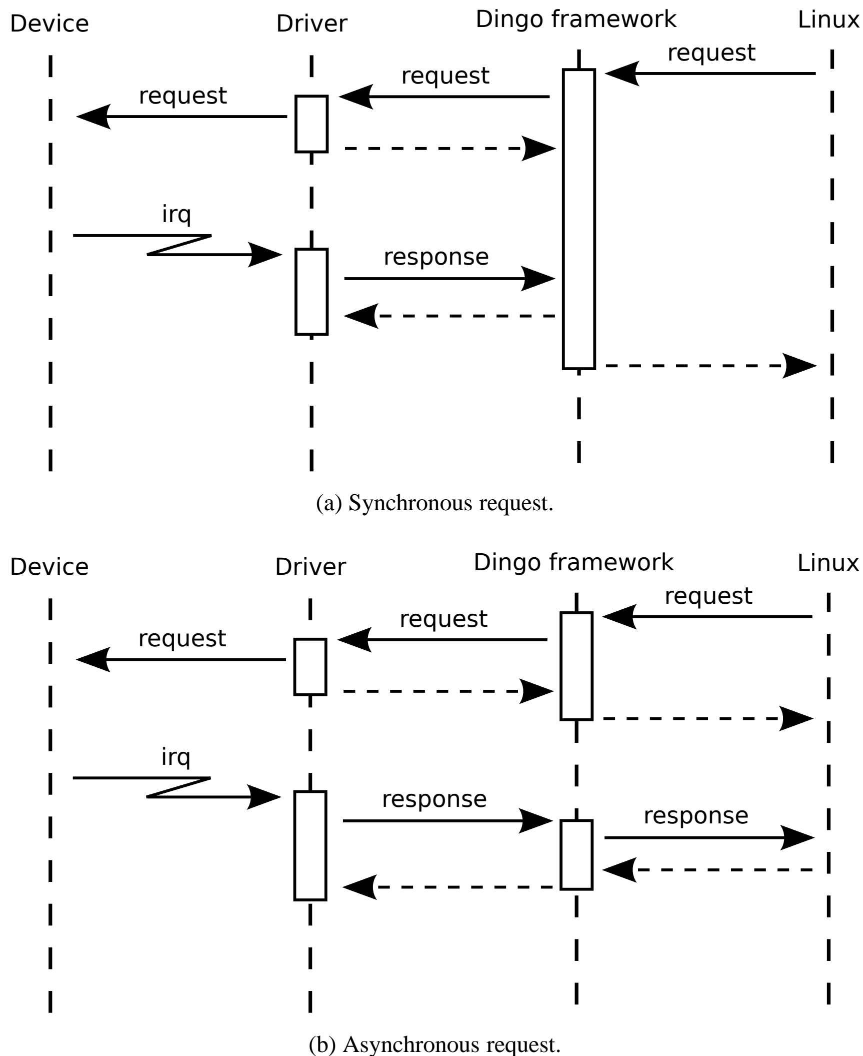
Further behaviour of the requesting thread depends on whether the request is synchronous or asynchronous. In the former case, the requesting thread blocks waiting for a completion notification from the driver, which may require waiting for a response from the device (Figure 5.5a). In the latter case, control is returned to the caller immediately and the driver response is delivered to the kernel by invoking the appropriate callback function (Figure 5.5b).

Figure 5.6 shows how the framework handles requests sent by the Dingo driver to Linux. If Linux handles this type of request asynchronously, then the request is simply forwarded to the kernel and later the response is forwarded to the driver (Figure 5.6a). Otherwise, if the corresponding Linux function may block then the request must be handled in a separate worker thread to avoid blocking the Dingo driver (Figure 5.6b). In the current implementation, the kernel-global worker thread is used for this purpose (via the `schedule_work` mechanism).

5.2.4 Selectively reintroducing multithreading

One limitation of the event-based model is that it allows at most one event handler to run at a time and therefore prevents the program from exploiting multiprocessor parallelism. Most I/O devices handle data at much slower rates than the CPU and can be easily saturated by a single processor core. Therefore this limitation is not an issue for the vast majority of drivers. As we will see in Section 5.4.3, a careful implementation of the event model enables event-based drivers to achieve the same I/O throughput and latency as multithreaded drivers.

However, there exist devices, such as 10Gb Ethernet or InfiniBand controllers, designed



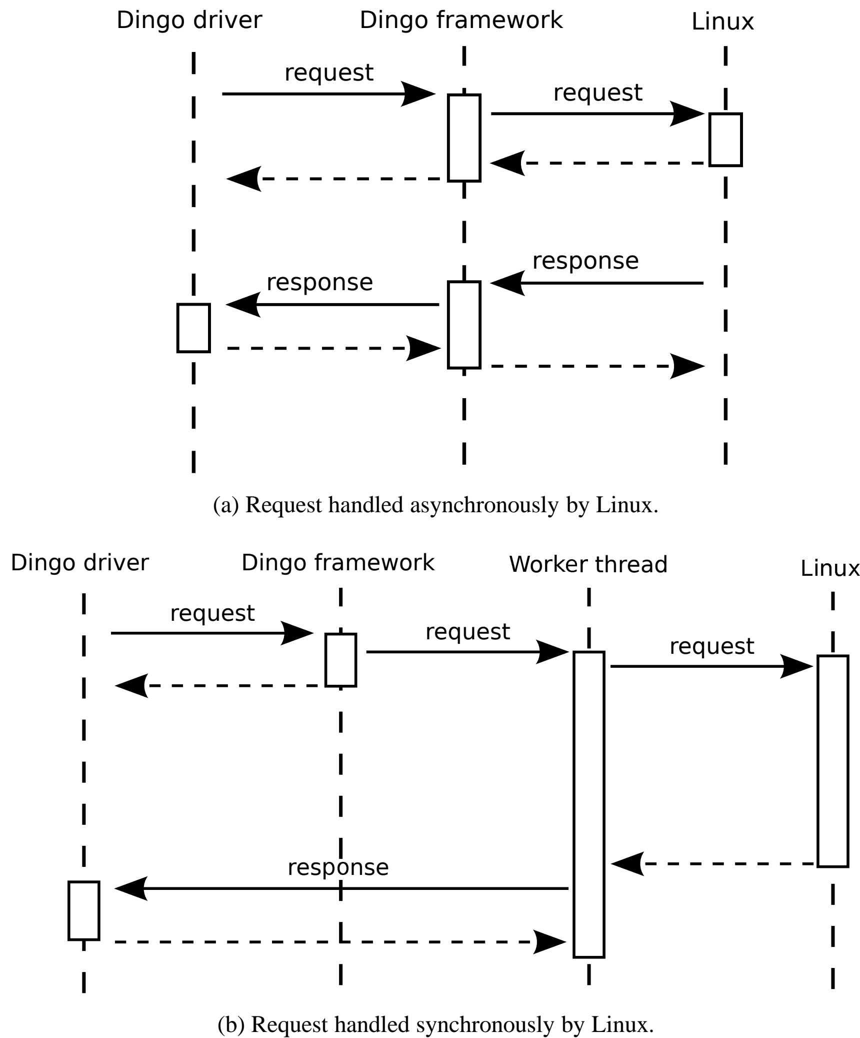
(a) Synchronous request.

(b) Asynchronous request.

Figure 5.5: Handling of synchronous and asynchronous requests sent by the Linux kernel to a Dingo driver. Solid arrows represent method invocations; dashed arrows represent returns from method invocations; zig-zag arrows represent interrupt notifications from the device.

for very high throughput and low latency, whose performance on multiprocessor systems may suffer from request serialisation. Evaluation presented in Section 5.4.3 has shown that the main source of performance overhead in Dingo drivers for such devices is cache bouncing of the spinlock variable used to synchronise access to the driver request queue (see Section 5.4.3). Although in experiments discussed in Section 5.4.3 event-based drivers perform well even for these devices, it is desirable to allow driver developers to use multithreading when absolutely necessary.

High-performance devices are designed to minimise contention and avoid synchronisation in the data path. As a result, the synchronisation complexity in their drivers is concentrated in the control path, whereas the data path is free of synchronisation operations. Based on this observation, I introduce a hybrid model in Dingo, which allows concurrency among data requests but not control requests. In this model, all control methods are serialised with



(a) Request handled asynchronously by Linux.

(b) Request handled synchronously by Linux.

Figure 5.6: Handling of requests sent by a Dingo driver to the Linux kernel.

respect to each other and to data methods. However, multiple data methods are allowed to execute concurrently. If there exist race conditions in the data path, they must be handled by the driver developer in the conventional Linux way, using spinlocks (the use of blocking synchronisation primitives is not allowed).

The driver implementer can choose whether the driver should run in the fully serialised mode or in the hybrid mode. Drivers running in the hybrid mode benefit from the advantages of the event-based model without experiencing any added overhead of serialisation in the data path.

The distinction between data and control methods is drawn by the protocol designer who labels methods that can be sent or received concurrently with the `concurrent` keyword.

We have implemented both modes for the InfiniBand driver described in Section 5.4. Our original implementation was fully serialised. We found that no changes to the driver were needed to run it in the hybrid mode, since the data path of the driver did not require any synchronisation.

Support for the hybrid model required only minimal changes to the Dingo runtime

framework. The mutex protecting access to the driver was replaced by a read-write lock. Concurrent requests acquire the lock in the read mode; all other requests acquire the lock in the write mode.

5.2.5 Comparison with existing architectures

Section 2.5 described two existing I/O framework architectures that simplify concurrency management in device drivers by serialising driver invocations, namely the Mac OS X IOKit and the Windows Driver Foundation. The Dingo event-based architecture differs from these existing solutions in three ways:

1. In Dingo, all driver operations, without exceptions, have non-blocking semantics.

2. With the exception of drivers for a small number of devices with extreme performance requirements, all driver invocations are serialised. This means that no two events can be delivered to the driver concurrently and under no circumstances does the driver need to create a parallel thread to perform any of its functions. As a result, all code in a Dingo driver is guaranteed to execute atomically.

3. As described in Section 5.2.3 and confirmed experimentally in Section 5.4.3, the Dingo serialisation architecture can be implemented with very low performance overhead. While I was unable to find performance data for Windows and Mac OS X I/O frameworks, their architecture [App06, Mic06] clearly leads to performance degradation due to serialisation on multiprocessor systems.

5.3 Tingu: describing driver software protocols

This section addresses the second shortcoming of the device driver architecture in current OSs, namely the lack of well-defined communication protocols between the driver and the rest of the kernel.

In Chapter 4 we saw that 20% of driver defects are violations in the ordering or format of interactions with the OS. A closer study of driver protocols in Linux shows that these protocols are stateful, i.e., operations that the driver must be prepared to handle and operations that it is allowed to invoke in a given state are determined by the history of previous interactions. However, these constraints are not adequately reflected in the OS documentation, forcing driver developers to guess correct behaviour. For example, details of how to react to a hot-unplug notification in the driver's different states, or how to handle a shutdown request that arrives during a transition to the suspend state (and whether such a situation is even possible), are not easy to find in documentation.

This problem is not specific to Linux. Other systems, including Windows [Mic] and Mac OS X [App06], define driver interfaces in terms of functions that a driver must implement and callbacks that a driver may invoke. Such a definition often leaves constraints on

ordering and arguments implicit in the OS implementation. Sample drivers provided with documentation kits may shed some light on these constraints, but relying on examples as an ultimate reference is a sure way to introduce bugs.

Therefore, improved OS documentation providing a complete and easy-to-understand description of the required driver behaviour has the potential to significantly reduce defects caused by the complexity of OS protocols.

Dingo facilitates the development of such documentation by specifying the communication protocols between drivers and the OS using a formal language. While informal descriptions tend to be incomplete and can easily become bulky and inconsistent, a well-chosen formalism can capture protocol constraints concisely and unambiguously, providing driver developers with clear instructions regarding the required behaviour.

Additionally, by providing a specification of driver protocols, we enable formal checking of driver correctness, both statically and at runtime. In particular, Section 5.3.3 presents a solution for automatic runtime checking of device drivers against protocol specifications. This is achieved using a runtime observer that intercepts all interactions between the driver and the OS and detects situations where either the driver or the OS violates the protocol.

The challenge in designing the protocol specification language is to satisfy both expressiveness and readability requirements. In order to be useful, driver protocol specifications must be easily understood by driver developers. This encourages the use of simple visual formalisms such as *finite state machine (FSM)* or *Unified Modeling Language (UML)* sequence diagrams [Obj09]. Unfortunately many aspects of driver protocols cannot be expressed using these simple notations.

One such aspect is the dynamic structure of driver protocols. For instance, the USB bus transport protocol allows client drivers to create multiple data connections, called pipes, through the USB bus to their associated devices at runtime. In the ax88772 USB-to-Ethernet adapter example (Figure 5.1), such functionality is used by the driver to open several data and control connections to the device. Each such connection operates in parallel with the others and behaves according to its own protocol.

This example also serves to highlight another complication common to driver protocols, namely protocol dependencies. In the ax88772 driver, the behaviour of each individual pipe is dependent on the state of the main USB bus protocol. For instance, no data transactions can be issued through pipes after the bus has been switched to a low-power mode. Given that pipes behave according to their own protocols, and that this behaviour is dependent on the behaviour of the USB bus as specified by its own protocol, we require a means to describe dependencies between different protocols.

Neither the dynamic spawning of concurrent behaviours, nor the dependencies among behaviours can be easily expressed using simple formalisms like FSM.

The search for a formalism that supports both the required expressiveness and readability has led to the development of a new software protocol specification language called Tingu. The design of Tingu is driven by experience in specifying and implementing real

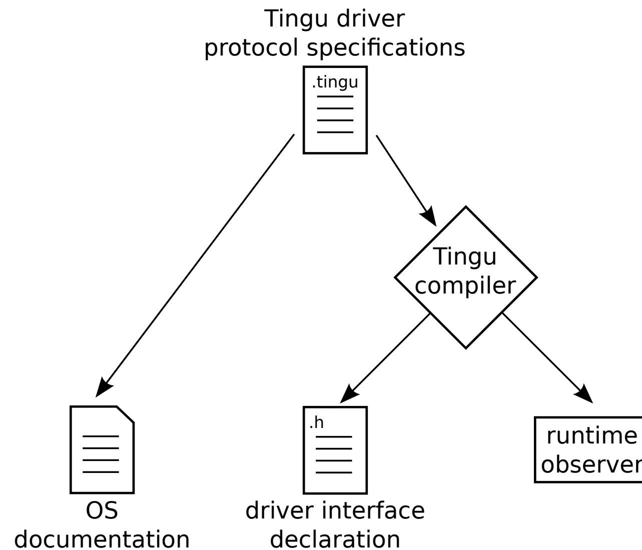


Figure 5.7: The use of Tingu protocol specifications.

driver interfaces. In particular, a construct is only introduced to the language if it has proven necessary for modelling the behaviour of several types of drivers and can not be expressed easily using other constructs.

Tingu has both a textual and visual component. The textual component is used to declare elements of a protocol, such as data types and methods. The visual component is used to specify protocol behaviour using a subset of the Statecharts [Har87] syntax extended with several new constructs that provide support for dynamic port spawning and protocol dependencies. With dynamic spawning one can specify a behaviour that leads to creation of a new port at runtime. Protocol dependencies define method ordering constraints across multiple protocols. This is achieved by allowing several protocols to constrain occurrences of the same method: the method can only be invoked when it is permitted by all involved protocols.

Figure 5.7 summarises the use of Tingu protocol specifications. Their primary purpose is to serve as part of the OS driver framework documentation providing intuitive guidelines to driver programmers. Tingu specifications are provided as input to the Tingu compiler, which generates C header files containing prototypes of driver interface methods. The compiler also generates runtime observers for Dingo drivers, which allow automatic checking of protocol compliance at runtime.

5.3.1 Introduction to Tingu by example

This section introduces the syntax and semantics of the Tingu language using the example of the ax88772 driver and its protocols.

Component The top-level entity in the Tingu language is the component, which describes the OS interface of a device driver by listing its ports. Fig-

```
component ax88772
{
  ports:
    Lifecycle lc;
    PowerManagement pm<lc/lc>;
    EthernetController eth<lc/lc,pm/pm>;
    USBInterfaceClient usb<lc/lc,pm/pm>;
    Timer ctrlTimer;
};
```

Figure 5.8: Tingu declaration of the ax88772 driver component.

ure 5.8 shows the Tingu specification of the ax88772 driver (Figure 5.1). Every line in the ports section declares one port of the driver using the <protocol_name> <port_name>['<'<port_substitutions>'>'] syntax. The optional port_substitutions clause is explained later.

Protocols A protocol specification declares methods to be exported and imported by a driver that implements the given protocol along with constraints on the ordering, timing, and arguments of method invocations.

A protocol describes common behaviour of all drivers that implement or use the functionality described by the protocol. For example, all Ethernet controller drivers must implement the EthernetController protocol, whereas all drivers for USB devices must use the USBInterfaceClient protocol to access the device.

Dingo protocols are OS-specific. For instance, the EthernetController protocol defined as part of the Dingo framework for Linux is closely modeled after the native Linux Ethernet driver interface, which is substantially different from corresponding interfaces in other OSs.

An alternative approach would be to define OS-independent protocols, which would enable the development of portable device drivers. Coming up with protocols that would reconcile differences among various OSs while permitting efficient implementation is a hard problem and is beyond the scope of this thesis.

A Dingo protocol is obtained from the corresponding Linux protocol (specified informally in the Linux documentation and source code) by replacing all blocking operations with request/completion method pairs, as described in Section 5.2. In some cases, additional minor changes were introduced in order to make the protocol easier to understand and implement.

Figure 5.9 shows the declaration of the Lifecycle protocol. This protocol defines initialisation and shutdown requests that must be implemented by all Dingo drivers in Linux. The methods section of the protocol declaration lists protocol methods and their signa-

```
protocol Lifecycle
{
methods:
  /*Probe and initialise the device*/
  in  probe();
  /*Initialisation completed successfully*/
  out probeComplete();
  /*Initialisation failed*/
  out probeFailed(error_t error);

  /*Stop the device and release all resources held by the driver*/
  in  stop();
  /*Deinitialisation complete*/
  out stopComplete();

  /*Hot-unplug notification*/
  in  unplugged();

transitions:
  import(format=rhapsody, location="LifecycleSM@ioprotocols.sbs");
}
```

Figure 5.9: The `Lifecycle` protocol declaration.

tures. Method declarations are similar to function declarations in C, but without a return type and with the addition of a direction specifier. By default, method arguments are passed from the caller to the callee. The `out` qualifier in front of an argument declaration denotes an argument used to return a value from the callee to the caller.

The `transitions` section describes the format and location of the protocol state machine, which defines legal sequences of method invocations.

Figure 5.10 shows the state machine of the `Lifecycle` protocol. Transitions of this state machine are triggered by method invocations between the driver and the OS, with question marks ("?") in trigger names denoting incoming invocations (the OS calling the driver) and exclamation marks ("!") denoting outgoing invocations (the driver calling the OS). The protocol state machine is interpreted as follows: any method invocation that triggers a valid state transition complies with the protocol specification. An invocation that does not trigger any valid transitions violates the protocol specification.

A compact representation of complex protocols is achieved by organising states into a hierarchy—a feature provided by Statecharts. Several primitive states can be placed inside another state, called super-state. A transition originating from a super-state (e.g., the `?unplugged` transition in Figure 5.10) is enabled when the state machine is in any of its internal states.

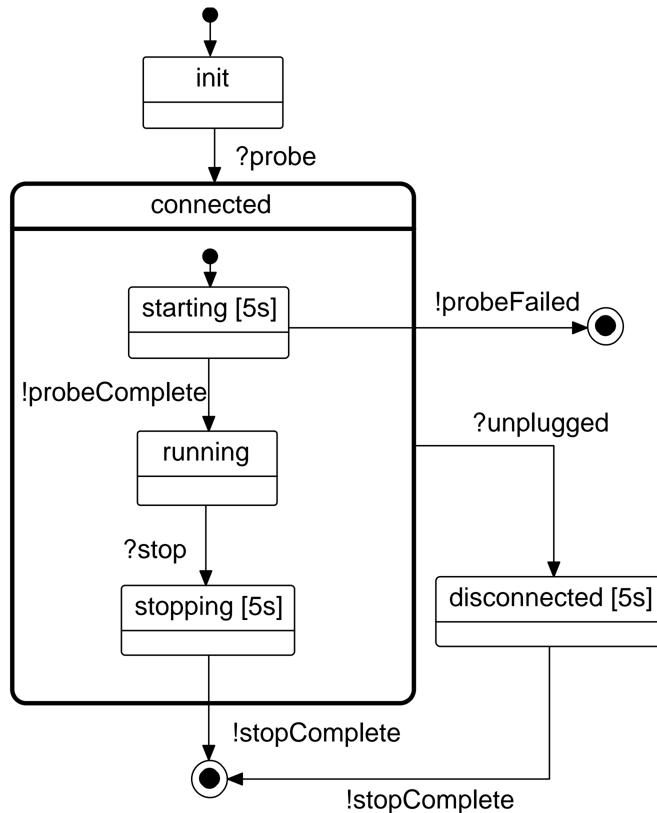


Figure 5.10: The `Lifecycle` protocol state machine.

When the driver is created, the protocol state machine is in its initial state, denoted by a dot and an arrow. The protocol terminates, i.e., no more methods of this protocol can be invoked, when it reaches one of its final states, denoted by a circled dot.

In Figure 5.10 some states include timeout annotations in square brackets. 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 instance, the driver is not allowed to stay in the `starting` state indefinitely. It must either complete initialisation or fail within five seconds after entering the state.

Other features of the Tingu language are illustrated by the `PowerManagement` protocol declared in Figure 5.11. This protocol defines device suspend and resume requests that must be implemented by all Linux drivers that support power management.

Types The `types` section of the protocol defines data types to be used in method arguments and protocol variables declarations. Tingu supports a subset of the C type system, including integers, enumerations, structures, and pointers. In addition, it supports a small number of built-in data types, such as lists and stacks, which will be explained later.

The `PowerManagement` protocol declares the `power_level_t` type (line 4), which is used to describe four standard device power states. An argument of this type is passed to the `suspend` request (line 13).

```
1  protocol PowerManagement
2  {
3    types:
4      enum power_level_t {
5        D0 = 0,
6        D1 = 1,
7        D2 = 2,
8        D3 = 3
9      };
10
11   methods:
12     /*Put the device into a low-power state*/
13     in  suspend (power_level_t level);
14     out suspendComplete ();
15
16     /*Resume the device*/
17     in  resume ();
18     out resumeComplete ();
19
20   variables:
21     power_level_t power_level;
22
23   dependencies:
24     Lifecycle lc {
25       listens   probeComplete;
26       listens   probeFailed;
27       listens   unplugged;
28       restricts stop;
29   };
30
31   transitions:
32     import(format=rhapsody,
33            location="PowerManagementSM@ioprotocols.sbs");
34 };
```

Figure 5.11: The `PowerManagement` protocol declaration.

Protocol variables Some protocol state information is inconvenient to model using explicit states and is more naturally described by variables. For example, the `PowerManagement` protocol models a device's current power level using an integer variable called `power_level` (line 21).

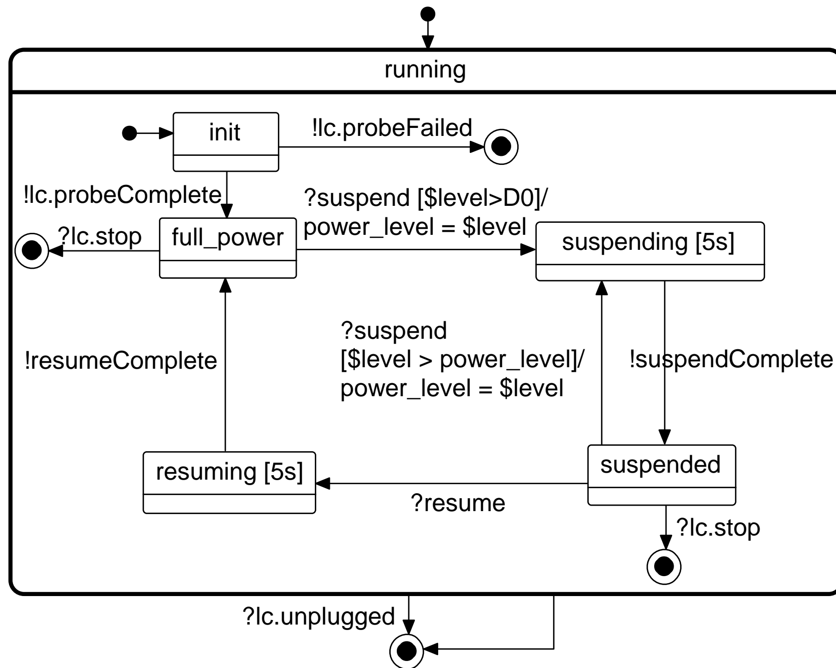


Figure 5.12: The PowerManagement protocol state machine.

Protocol dependencies The PowerManagement protocol also illustrates the use of protocol dependencies. When dealing with power management, a device cannot be suspended until it has completed initialisation. This rule can be expressed as a dependency between the Lifecycle and PowerManagement protocols: The PowerManagement state machine may accept suspend requests only after the probeComplete transition of the Lifecycle protocol. This is shown in Figure 5.12 with the transition from init to full_power.

The PowerManagement protocol declaration lists Lifecycle methods used in the power management state machine in the dependencies section (Figure 5.11, line 23). Line 24 states that any driver implementing the PowerManagement protocol must also provide the Lifecycle protocol through a port named lc. This port name is used to refer to lifecycle methods from the power management state machine. The actual port name may be different. The mapping between expected and actual port names is established in the component declaration (see expressions in angle brackets in Figure 5.8).

The restricts and listens keywords (Figure 5.11, lines 25–28) describe two types of protocol dependencies. The restricts dependency means that the method is only allowed to be called if it triggers a state transition in both its main protocol (i.e., the protocol that declares the method in its methods section) and the dependent protocol. The listens dependency means that the dependent protocol may react to the method invocation but does not restrict its possible occurrences.

The restricts dependency is useful in simplifying driver protocols to reduce the amount of concurrency that the driver must handle. For example, the power management state machine only allows the lc.stop request in full_power and suspended states,

but not in intermediate `suspending` and `resuming` states. This guarantees that the driver never receives a `stop` request while handling a power management request.

Note that the native Linux driver interface does not provide this guarantee, rather it is enforced in the Dingo runtime framework by introducing additional synchronisation between power management and lifecycle requests. This way, some of the complexity is shifted from the device driver into the framework. This kind of improvement is enabled as a byproduct of defining behavioural constraints on device drivers explicitly and formally.

Transition guards and actions A protocol state transition can include an optional guard and action. The guard is a boolean expression over protocol variables and method arguments that must be satisfied when the transition is taken. The action consists of one or more statements that specify how protocol variables are updated when the given transition is taken. Consider, for example, the transition from state `full_power` to `suspending` in Figure 5.12. The guard expression in square brackets specifies that the `level` argument of the `suspend` method must be greater than the `D0` constant, corresponding to the zero power saving mode ('$'-sign before an identifier denotes method argument). The action associated with the transition updates the value of the `power_level` variable to reflect the new power state.

Tingu guards and actions use a subset of the C syntax, limited to assignment, arithmetical, and logical expressions. No control structures, such as loops, branches, or function calls, are allowed. The primary motivation for this restriction is to preserve the simplicity and visual appeal of protocol specifications.

Dynamic port spawning With dynamic spawning one can specify a behaviour that leads to creation of a new port at runtime. I illustrate this feature using the example of the `USBInterfaceClient` protocol, which describes the service provided by the USB bus framework. As mentioned above, USB data transfers are performed via USB data pipes. The behaviour of an individual pipe is specified by the `USBPipeClient` protocol. Since the USB bus allocates these pipes dynamically, the driver determines which pipes it will use at runtime.

The relevant fragments of the `USBInterfaceClient` protocol declaration are shown in Figure 5.13. The `ports` section (line 3) lists ports that can be created by the `USBInterfaceClient` protocol at runtime. These ports are called *subports* of the main port that implements the `USBInterfaceClient` protocol. The identifier in square brackets (line 4) is the data type used to index dynamically spawned subport instances; in this case USB pipes are indexed by their endpoint address. The port substitution expression in angle brackets binds dependencies of a subport to ports that are visible in the namespace of the `USBInterfaceClient` protocol. In particular, `self` refers to the `USBInterfaceClient` port itself.

The `spawns` clause in line 10 states that a new pipe is created when the driver invokes

```
1 protocol USBInterfaceClient
2 {
3   ports:
4     USBPipeClient pipe[usb_endpoint_addr_t]
5       <self/iface,lc/lc,pm/pm>;
6
7   methods:
8     ...
9     out pipeOpen(usb_endpoint_addr_t address,
10                usb_xfer_type_t type) spawns pipe;
11
12   ...
13 };
```

Figure 5.13: A fragment of the `USBInterfaceClient` protocol declaration.

the `pipeOpen` method. The generated C function prototype for this method defines an additional argument that takes a pointer to the newly allocated port. When the driver invokes this method, the USB framework allocates a USB pipe and binds it to the provided port, so that the driver can immediately start using the pipe through this port.

The dynamic port spawning behaviour must also be reflected in the protocol state machine. Figure 5.14 shows the state machine of the `USBInterfaceClient` protocol. The `new` operator in the highlighted transition indicates that the `pipeOpen` method creates an instance of the `pipe` subport; the index of the new subport is equal to the `address` argument of the method.

Abstract data types Many device driver protocols allow the driver to handle multiple outstanding I/O requests. These protocols define constraints such as: the driver (or the OS) must complete requests in the FIFO order, or the driver must complete all outstanding requests before terminating. Modelling these constraints requires the protocol specification to incorporate a model of the request queue. To this end, the Tingu type system includes the `list` *abstract data type (ADT)*.

For instance, the `USBPipeClient` protocol uses a list variable to model the queue of outstanding USB transfer requests. The variable declaration is shown in Figure 5.15. New transfers are added to the tail of the queue, completed transfers are removed from the head of the queue. To this end, Tingu defines a set of standard list manipulation operations that can be applied to list variables.

Figure 5.16 shows a single transition of the `USBPipeClient` state machine that illustrates the use of the `transfers` variable. It specifies that a pipe must complete transfers in the FIFO order by asserting that the transfer request completed by the `!transferComplete` method must be the same as the one pointed to by the head of the

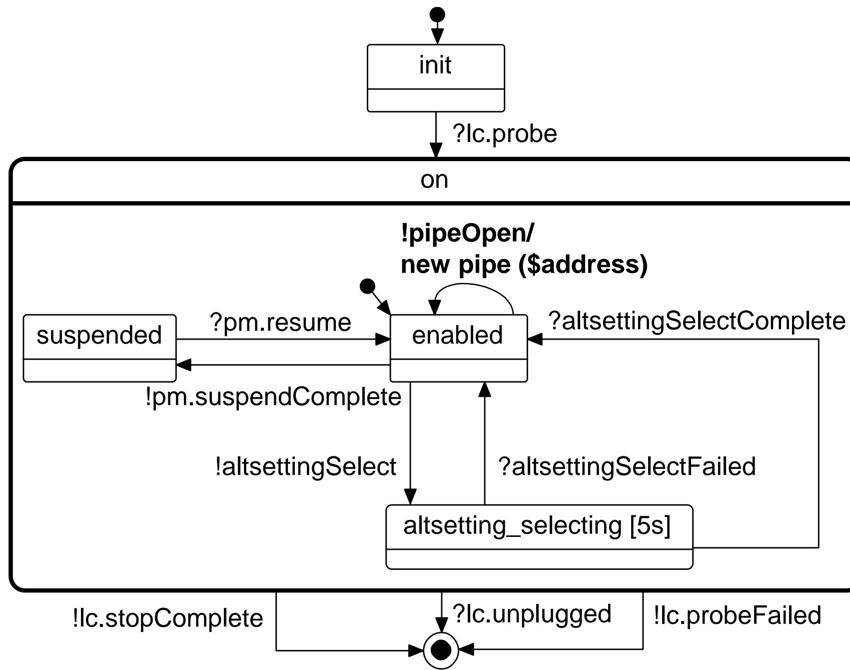


Figure 5.14: The `USBInterfaceClient` protocol state machine.

```
protocol USBPipeClient
{
  ...
  variables:
    list<dingo_urb*> transfers;
    ...
};
```

Figure 5.15: A fragment of the `USBPipeClient` protocol declaration.

transfers queue.

Note that protocol variables describe the state of the interaction between the driver and the OS rather than the internal driver or OS state. In the above example, the `transfers` variable helps specify the USB framework behaviour that the driver can rely upon. Neither the driver nor the framework is required to store a USB transfer list to implement this be-

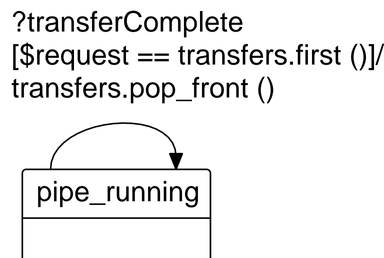


Figure 5.16: A fragment of the `USBPipeClient` protocol state machine.

haviour. In practice, the USB framework simply forwards USB transfer requests to the USB controller and relies on the controller to complete them in the FIFO order.

Currently, the only supported ADTs are lists and sets. Other ADTs, such as queues and stacks, can be added as required. Tingu does not currently provide a facility to specify user-defined ADTs, therefore support for new ADTs must be built into the Tingu compiler.

5.3.2 Discussion

Tingu protocol state machines do not model return-from-method events. This means that a Tingu protocol cannot specify when a method invocation must occur with respect to the completion of another method. For instance, the protocol state machine in Figure 5.10 states that the call to the `probeComplete` method must occur after the invocation of the `probe` method, but does not specify whether it should happen before or after the `probe` method returns. The correct interpretation of this specification is that both behaviours are allowed. The driver may complete device initialisation inside the `probe` method and notify the OS by invoking `probeComplete` before returning from `probe`. Alternatively, if the device initialisation involves waiting for a timeout or an interrupt from the device, then the driver returns from `probe` and completes the initialisation later, in the context of a different method.

Most driver operations share the same property: depending on the device interface they may or may not be able to complete immediately. Therefore, explicitly modelling return events would not help specify additional useful constraints in Tingu while making specifications larger and harder to understand.

One aspect of the driver interface currently not captured by Tingu protocols is I/O buffer management. Linux and other OSs define complex APIs for manipulating I/O buffers, including operations for cloning, merging, padding buffers, etc. These interfaces do not fit well into the state machine framework of Tingu. Rather they can be formalised using ADTs or a related formalism. While Tingu does provide limited support for ADTs, a full description of such interfaces written using the present version of the language would lead to bulky unintuitive specifications, which would defeat the purpose of Tingu. As such, these APIs continue to be specified using C header files and informal documentation.

This completes the overview of the Tingu language. Complete syntax of the language is described in Appendix A. Appendix B presents several examples of Tingu protocol specifications.

5.3.3 Detecting protocol violations at runtime

Tingu specifications help driver developers avoid protocol violations, but do not eliminate them completely, since the developer may still make an error in the implementation of a protocol, even if the protocol is clearly defined.

The use of a formal language to specify driver protocols opens up the possibility to

explore new techniques to verify protocol compliance of a driver, in addition to providing a reference to driver writers. One avenue that I have explored in this project is runtime verification, i.e., automatic validation of the driver behaviour against protocol specifications at runtime.

The Tingu compiler fully automates runtime verification by generating a driver *protocol observer* from the Tingu specification of its ports. The generated observer can be attached transparently to the driver. It intercepts all method invocations exchanged by the driver and keeps track of the state of all its protocols. Whenever the driver or the OS performs an illegal operation or fails to perform an operation within the time interval specified by a timeout state, the observer notifies the OS about the failure and outputs the current state of all driver protocols and the sequence of events leading to the failure.

Protocol observers have proved useful in testing and debugging device drivers during the development cycle. They can also be combined with any of the fault isolation and recovery solutions described in Section 3 to enhance the resilience of a production system to driver failures.

Another promising research direction that falls beyond the scope of this thesis is static verification of drivers against protocol specifications. One way to achieve this is to translate Tingu into a language supported by an existing model checker [ECCH00,CFH05,BBC+06]. Such translation is possible because these languages incorporate similar concepts to Tingu, but using textual rather than visual syntax. However, static analysis of drivers has not been investigated in this thesis, therefore no experimental evidence proving or disproving the feasibility of this approach has been obtained. In particular, it is unclear whether existing model checkers are sufficiently powerful to validate complex behaviours captured by Tingu protocols.

5.3.4 From protocols to implementation

Tingu protocols specify the externally visible driver behaviour and do not enforce any particular internal structure. In practice, however, the driver developer will typically closely follow the structure of the specification, maintaining correspondence between the driver code and protocol states. In this approach, driver protocol specifications are viewed as the first approximation of the driver design, which is refined into the implementation by adding device interaction code.

In principle, protocol specifications could be used to automatically generate skeleton code for the driver. However, Dingo currently does not provide tools for this. The driver developer must implement all driver methods manually, using protocol specifications as guidelines.

I illustrate this approach to driver development using an excerpt from the ax88772 driver implementation. Figure 5.17 and Figure 5.18 show a fragment of the `EthernetController` protocol that describes the packet transmission interface of an

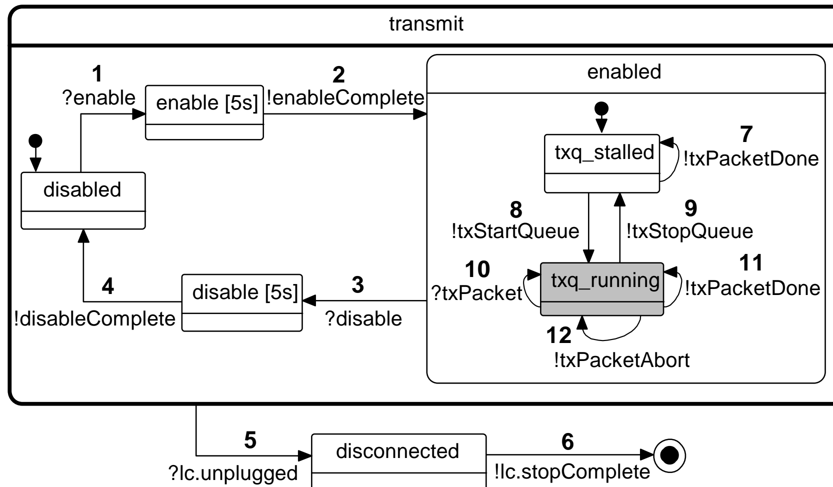


Figure 5.17: A fragment of the `EthernetController` protocol state machine. Numbers above transition labels are for reference only and are not part of the protocol specification.

Ethernet driver and a simplified version of the corresponding fragment of the ax88772 driver code.

The protocol We focus on the state labeled `txq_running`. According to Figure 5.17, in this state the driver must be prepared to handle one of the following requests from the OS: `txPacket` instructing the driver to queue a packet for transmission, `disable` requesting the driver to disable the device receive and transmit circuitry, and `lc.unplugged` notifying the driver about a hot-unplug event. It is allowed to invoke one of the following methods: `txPacketDone` to notify the OS about successful transmission of a packet, `txPacketAbort` to report an error that occurred while sending a packet, and `txStopQueue` to prevent the OS from sending new packets until more buffer space becomes available in the controller.

The implementation During initialisation (not shown in the listing), the driver calls two reactive functions, thus spawning two C-with-events language threads, one of which handles packet reception and the other packet transmission. Figure 5.18 shows the fragment of the transmission thread, which implements its behaviour in the `txq_running` state.

In accordance with the protocol specification, when the driver arrives in this state, it pauses, waiting for one of the enabled external requests. As discussed in Section 5.2.1, waiting in Dingo drivers is implemented using the `AWAIT` construct. However, `AWAIT` can only wait for events, which are language-level entities internal to the driver. It does not allow waiting for an interface method invocation. The solution is to transform method invocations into events. This is illustrated by the implementation of the `txPacket` method (line 46 in Figure 5.18), which simply emits the `txPacket` event.

```
1  reactive txLoop(ax88772 * drv)
2  {
3    ...
4    AWAIT(txPacket,txDisable,txUnplugged,
5          pipeXferComplete)
6    {
7      IF(txPacket){/*transition #10*/
8        /*start packet xfer over USB*/
9        ...
10
11       if(/*out of buffer space?*/) {
12         /*transition #9*/
13         eth->txStopQueue(eth);
14       };
15     }
16     ELIF(txDisable){/*transition #3*/
17       /*abort outstanding USB xfers*/
18       pipe->abort(pipe);
19
20       /*wait for the abort to complete*/
21       AWAIT(pipeAbortComplete);
22       EMIT(txDisableComplete);
23     }
24     ELIF(txUnplugged){/*transition #5*/
25       /*wait for the USB pipe to abort
26         all outstanding transfers*/
27       AWAIT(pipeAbortComplete);
28       EMIT(txUnpluggedComplete);
29     }
30     /*USB xfer complete*/
31     ELIF(pipeXferComplete){
32       if(/*transfer successful?*/)
33        /*transition #11*/
34        eth->txPacketDone(eth,
35          pipeXferComplete.pkt);
36       else
37        /*transition #12*/
38        eth->txPacketAbort(eth,
39          pipeXferComplete.pkt);
40     };
41   };
42   ...
43 };
44 /*Driver methods called by the OS*/
45
46 reactive txPacket(
47   PEthernetController * eth,
48   sk_buff * packet)
49 {
50   txPacket.packet = packet;
51   EMIT(txPacket);
52 };
53
54 reactive disable(
55   PEthernetController * eth)
56 {
57   EMIT (txDisable);
58   AWAIT (txDisableComplete);
59   EMIT (rxDisable);
60   AWAIT (rxDisableComplete);
61
62   /*disable the controller*/
63   ...
64
65   /*transition #4*/
66   eth->disableComplete (eth);
67 };
68
69 reactive unplugged(
70   PLifecycle * lc)
71 {
72   EMIT (txUnplugged);
73   AWAIT (txUnpluggedComplete);
74   EMIT (rxUnplugged);
75   AWAIT (rxUnpluggedComplete);
76
77   /*release all resources*/
78   ...
79
80   /*transition #6*/
81   lc->stopComplete (lc);
82 };
```

Figure 5.18: A fragment of the ax88772 driver.

The `txLoop` thread waits for this and other enabled requests using the `AWAIT` statement in line 4. Since the driver participates in several different protocols, it must be prepared to handle method invocations belonging to all its protocols. In this example, the last event (`pipeXferComplete`) in the `AWAIT` statement corresponds to a `USBPipeClient` protocol method, which is called when the USB data pipe completes transferring packet data to the controller.

The rest of the `txLoop` listing shows how the driver handles each input event and indicates the correspondence between methods exchanged by the driver through the `EthernetController` protocol and state transitions in Figure 5.17.

Consider, for example, how the driver handles the `disable` request. The implementation of the `disable` method emits the (line 54) `txDisable` event and then waits for the `txDisableComplete` event (line 58), which is generated by `txLoop` after all outstanding USB transfers have been aborted (line 22). Lines 59–60 perform a similar interaction with the receive thread. After both transmit and receive threads have been notified, it is safe to disable the controller (line 62). Finally, the `disable` method sends a completion notification to the OS in line 64. This illustrates the use of events to synchronise different concurrent activities inside the driver.

The `unplugged` method (line 69) implements similar logic for the hot-unplug notification: after notifying the receive and the transmit threads that the device has been unplugged, it releases all remaining resources held by the driver and sends a `stopComplete` notification to the OS.

The implementation of the `txLoop` method illustrates the effect of the Dingo architecture on the internal structure of device drivers. In a conventional driver, the logic implemented by this method would be scattered among multiple handlers, making it harder to understand and maintain. The improved structure in Figure 5.18 is enabled by three features of Dingo: the event-based architecture, which guarantees atomicity of driver invocations, the C-with-events preprocessor, which allows event-driven logic to be expressed sequentially, and the Tingu protocol specification language, which explicitly enumerated events that the driver must be prepared to handle or generated in every state.

5.4 Evaluation

This section evaluates the Dingo driver architecture with respect to its impact on driver reliability and performance.

The evaluation is based on two Dingo drivers that I implemented for Linux: the ax88772 100Mb/s USB-to-Ethernet adapter driver described in the previous sections and a Mellanox InfiniHost^{TM} III Ex 10Gb/s dual-port InfiniBand controller driver. The ax88772 adapter is representative, in terms of complexity and performance, of the majority of I/O devices found in general-purpose computer systems. In contrast, the InfiniHost controller is an example of a complex high-end device supporting extremely low-latency and high-bandwidth I/O

	Number of synchronisation objects		Number of critical sections	
	Linux	**Dingo**	**Linux**	**Dingo**
AX88772	8	2	19	2
InfiniHost	24	6	51	10

Table 5.2: The use of synchronisation primitives by Linux and Dingo drivers.

transfers.

Both Dingo drivers are based on the corresponding native Linux drivers, which allows direct comparison between the two implementations.

5.4.1 Code complexity

Side-by-side comparison of the Dingo drivers and their Linux counterparts shows that Dingo drivers implement the complete functionality of Linux drivers without increase in code size or complexity. In particular, the use of C with events effectively addresses the stack-ripping problem: whenever a Linux driver performs a blocking call to wait for an I/O completion or a timeout, the Dingo driver achieves the same effect using the `AWAIT` construct.

By enforcing atomicity of driver invocations, Dingo dramatically reduces the amount of synchronisation code in drivers. As shown in Section 5.3.4, event-based drivers need to synchronise tasks that span multiple method invocations, but situations where such synchronisation is necessary are uncommon, compared to preemptively multithreaded drivers. Table 5.2 summarises the use of synchronisation primitives in Linux and Dingo drivers. Synchronisation primitives used by Linux drivers include mutexes, semaphores, spinlocks, wait queues, completions, etc. In Dingo drivers synchronisation is based on events, as seen in Section 5.3.4. The table shows the total number of synchronisation objects used by the Linux and Dingo versions of the ax88772 and InfiniHost drivers, as well as the total number of critical code sections protected by these objects.

5.4.2 Reliability

At this stage it is difficult to directly measure the effect of the Dingo architecture on the rate of defects in drivers. Only a few Dingo drivers have been created, and they have not been used for a sufficiently long time to gather a statistically significant sample of defects.

Therefore, I took an indirect approach to measuring the impact of Dingo on driver reliability. I analysed the Dingo ax88772 and InfiniHost drivers against a sample of defects found in similar Linux drivers. For every defect studied, I determined whether an analogous defect could be reproduced in a Dingo driver. Some defects simply cannot occur in Dingo, for example, most race conditions cannot be reproduced due to the event-atomicity guarantee. Likewise, deadlocks caused by invoking a blocking operation in the interrupt context are not expressible in Dingo. Furthermore, Dingo protocols rule out some request sequences

	Eliminated by design	**Reduced likelihood**	**Unchanged likelihood**
Concurrency faults	27	2	0
S/W prot. violations	9	11	12
Total	36	13	12

Table 5.3: Categorisation of faults based on their potential occurrence in Dingo.

that can occur in Linux drivers along with defects introduced when handling them. This was illustrated using the in Section 5.3.1 when discussing protocol dependencies in the context of the `PowerManagement` protocol.

For those defects that can be reproduced in Dingo, I established whether the incorrect behaviour caused by the defect is explicitly forbidden by driver protocols. While Dingo does not eliminate these defects, the probability of introducing them is reduced compared to Linux drivers due to the presence of a clear and complete specification of the protocol. If, however, a protocol violation defect slips into the driver implementation, it can be detected using runtime verification, as discussed in Section 5.3.3.

I used defects from the four USB-to-Ethernet adapter drivers used in the study of Linux driver defects (Table 4.1) and analysed them against the Dingo implementation of the ax88772 driver. I also used the 123 bugs found in the Linux InfiniHost driver and analysed them against the Dingo version of the same driver. Of the 201 bugs found in these drivers, I selected the 61 that belonged to the types of defects that Dingo is targeting, namely concurrency defects (29) and defects caused by the complexity of OS protocols (32) and that were applicable to the ax88772 and InfiniHost drivers (some Ethernet driver bugs were not applicable to the ax88772 driver due to differences in the device interface).

The results of the evaluation are summarised in Table 5.3. Of the 61 selected defects, 36 fell in the category of defects not expressible in Dingo. Of the remaining possible defects 13 were OS protocol violations whose likelihood is reduced in Dingo. Being manually introduced in the corresponding Dingo driver, these defects could be identified by the runtime failure detector during testing.

Finally, 12 defects were deemed equally likely to occur in Dingo drivers and native Linux drivers. These defects violated OS protocol constraints that were not captured by the Tingu specifications. Three of these defects were violations of buffer management protocols. As discussed in Section 5.3.2, Tingu does not currently provide means to specify these protocols.

The remaining 9 defects were related to incorrect use of OS data structures. Most of these defects occurred in the InfiniHost driver. This driver implements several types of objects, such as request and response queues, protection domains, user contexts, etc. The InfiniBand driver protocol defines requests, which allow the OS to query the state of these objects. In response to a query, the driver must return a correctly initialised object descriptor. Formalising the requirement that each field of the descriptor is set to a valid value in

Tingu requires adding complex guard expressions to all appropriate protocol state transitions. Complex transition labels tend to clutter the specification and compromise its clarity, thus defeating the primary purpose of Tingu; therefore I chose to leave these constraints unspecified.

This example highlights the trade-off between clarity and scalability, which is inherent to visual formalisms.

5.4.3 Performance[2]

We evaluated the performance overhead of the Dingo driver model on the ax88772 and InfiniHost drivers using the Netperf [Net] benchmark suite. All benchmarks were run using a Linux 2.6.27 kernel on an 8-way (4 physical CPUs with 2 hardware threads each) Itanium 2 1.7GHz with 8GB of RAM.

In all experiments in this section, the Netperf server was run with default arguments on the machine with the Dingo driver under test. The Netperf client was running on another machine with the following arguments:

```
netperf -H<server-ip-addr> -p <server-port> -t <benchmark>
-c -C -l 60 - -m 32K
```

where `benchmark` is one of `UDP_RR`, `UDP_STREAM`, or `TCP_STREAM`. The `-l 60` argument sets the time of the run to 60 seconds. The `-c` and `-C` arguments enable CPU utilisation calculations on both the server and the client. CPU utilisation numbers returned by Netperf on the multiprocessor system turned out very imprecise; therefore we computed CPU utilisation during the netperf run using the OProfile tool [OPr]. Finally, the `-m 32K` option sets the transfer size to 32768 bytes (this option is only used in stream benchmarks).

For the ax88772 driver we measured latency and throughput for a varying number of concurrent network connections. The latency test measured the average round-trip latency of a 1-byte packet. The throughput test measured the throughput achieved by unidirectional transfers of 32-kilobyte data blocks.

Figure 5.19 shows results of the latency test. The Dingo driver achieved latency within 4% of its Linux counterpart, while introducing a small CPU overhead due to the protocol translation and request queuing inside the Dingo framework. Importantly, this overhead does not increase while going from 1 to 32 clients on a multiprocessor system. The throughput benchmark (Figure 5.20) showed no significantdifference in performance between the drivers.

The InfiniHost driver was used as the second example due to its extreme performance requirements. The InfiniBand interconnect architecture is designed for very high throughput and low latency. Despite the use of zero-copy techniques, it still puts substantial pressure on the CPU, especially for small transfers. Furthermore, InfiniBand supports traffic isolation among multiple concurrent connections; therefore the InfiniBand stack in Linux is designed

[2] The performance evaluation described in this section was carried out in collaboration with Dr. Peter Chubb.

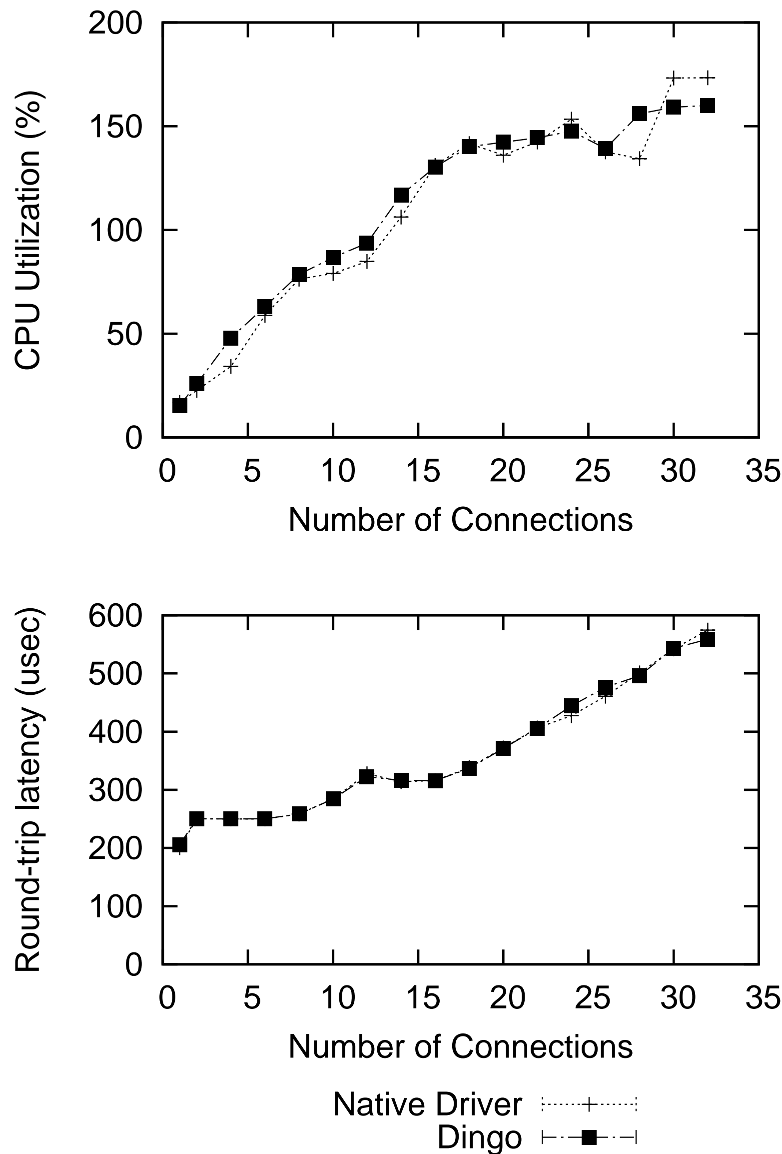


Figure 5.19: ax88772 UDP latency results. The top graph shows aggregate CPU utilisation over all connections (ranging from 0% to 800% on the 8-way system). The bottom graph shows average UDP echo latency across all connections.

to avoid synchronisation among data streams.

We compared the performance of the native Linux driver and the Dingo driver running in the fully serialised and hybrid modes. We used the IP-over-InfiniBand Linux module to send IP traffic through the InfiniBand link, and measured throughput and latency with Netperf. To achieve traffic isolation, we configured 32 independent network interfaces, one for each client, on top of the InfiniHost controller.

As shown in Figure 5.21, all three versions of the driver achieve the same latency. The serialised Dingo driver shows a small increase in CPU utilisation. In throughput benchmarks (Figure 5.22), the Dingo driver in the serialised mode showed 10% throughput degra-

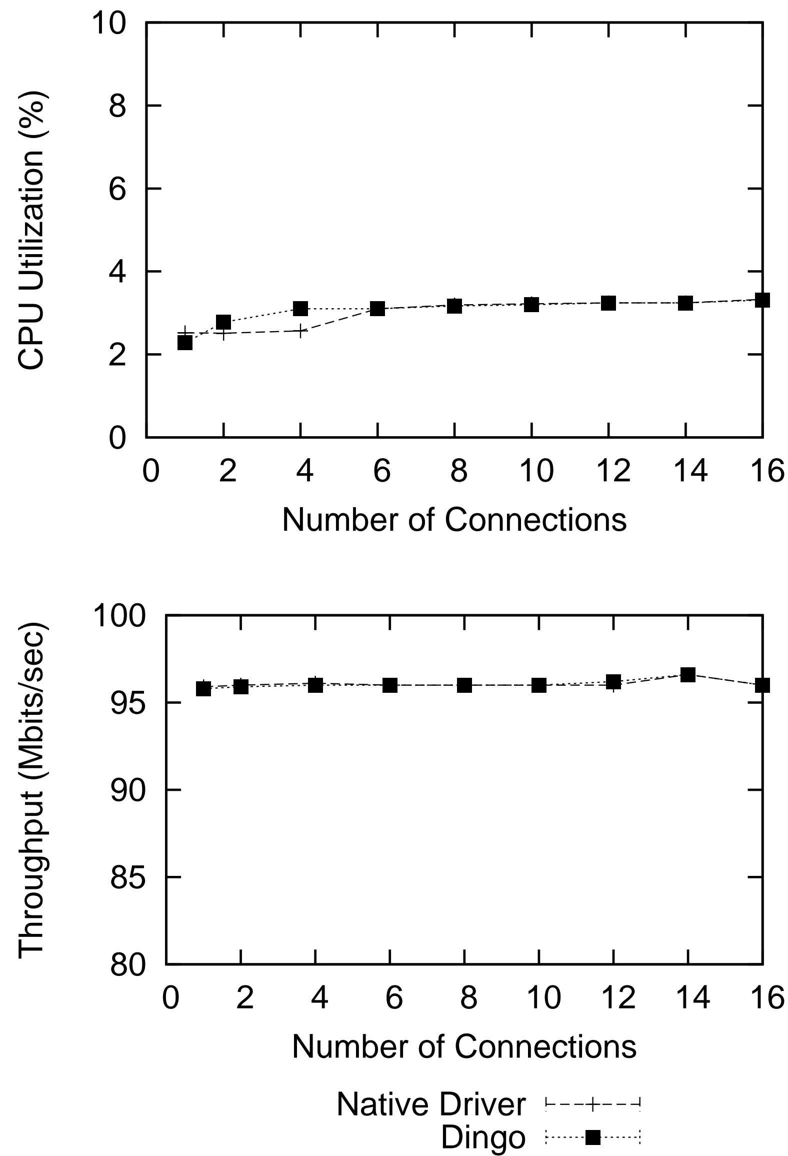


Figure 5.20: ax88772 UDP throughput results. The top graph shows aggregate CPU utilisation over all connections. The bottom graph shows aggregate UDP throughput.

dation in the worst case, and less than 3% throughput degradation and no CPU overhead in the hybrid mode (in the points where the hybrid driver consumes more CPU than the native one, it sustains proportionally higher throughput). In all cases the performance of Dingo drivers scaled as well as the native Linux driver. This shows that the Dingo hybrid mode allows drivers to take full advantage of multiprocessing capabilities.

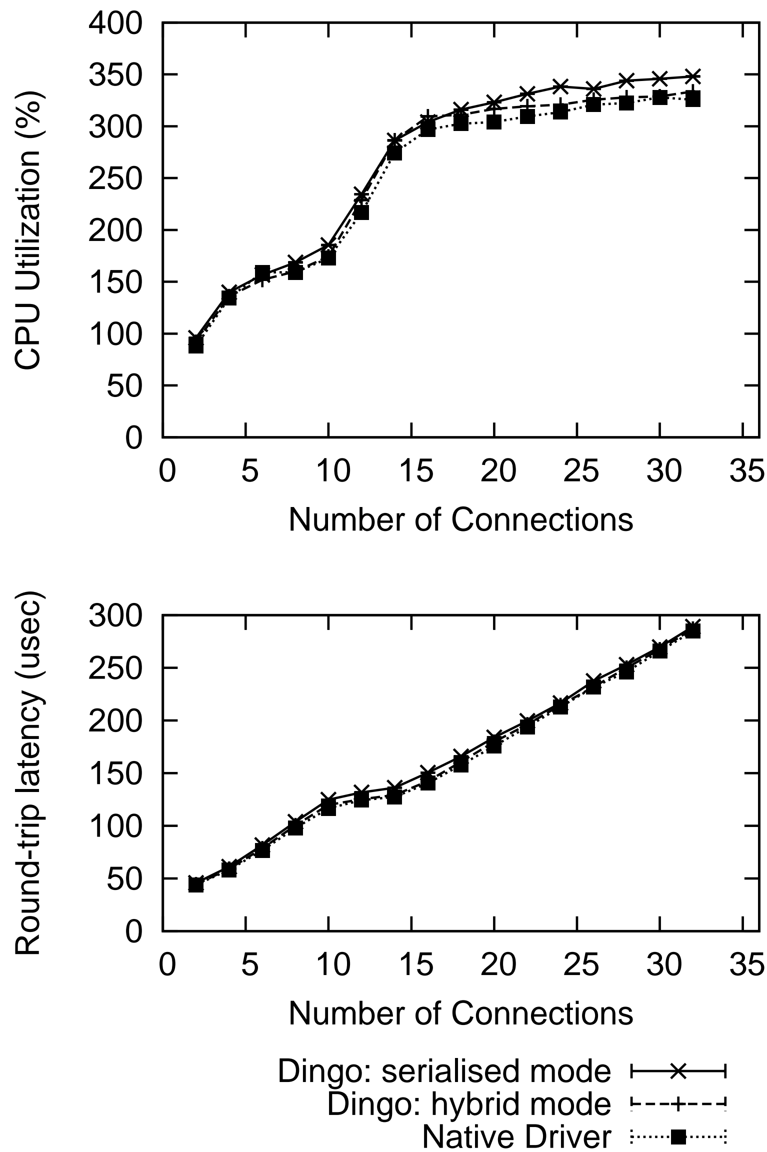


Figure 5.21: InfiniHost UDP latency benchmark results. The top graph shows aggregate CPU utilisation; the bottom graph shows average UDP echo latency.

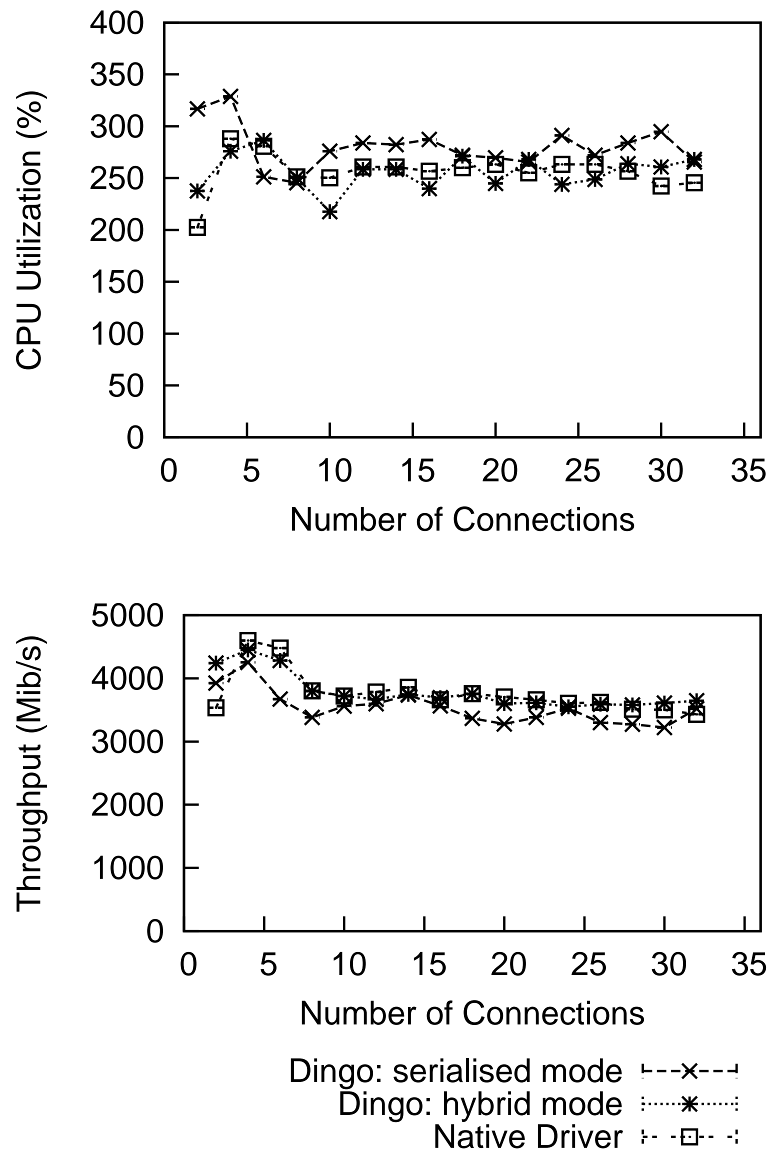


Figure 5.22: InfiniHost TCP throughput benchmark results. The top graph shows aggregate CPU utilisation; the bottom graph shows aggregate TCP throughput.

We used the OProfile [OPr] tool to identify the source of the CPU overhead in the serialised Dingo driver. We found that most of the extra CPU cycles were spent in acquiring the spinlock that protects the Dingo request queue, although the lock was not actually contended. This means that the overhead is caused by cache bouncing of the spinlock variable, as the lock is being repeatedly acquired by each of the 4 CPUs.

These experimental results indicate that for the ax88772 and Infinihost drivers, which are representative of a broad class of drivers with medium-to-high performance requirements, the reliability improvement offered by the Dingo architecture does not come at the cost of performance.

5.5 Conclusions

This chapter proposed several improvements to the device driver architecture and development methodology aimed to turn existing OSs into a more driver-friendly environment and reduce defects caused by the complexity of the driver-OS interface. First, it showed that the concurrency model based on events allows a reduction of the amount of non-determinism that the driver has to handle and therefore eliminates the majority of concurrency-related defects. Importantly, this is achieved with only a small impact on the performance. It further showed that the stack ripping problem arising from the use of the event-based model can be effectively addressed using a simple language extension.

Second, it presented a rigorous approach to defining interaction protocols between drivers and the OS that enables clear and precise specification of the required driver behaviour through the use of a visual formalism based on finite state machines. Examples presented throughout this chapter and in Appendix B show that Tingu is capable of capturing complex protocols in a compact specification that can be readily understood by software engineers.

While in this thesis Tingu is primarily used to formalise existing driver protocols, it is also suitable for designing new protocols. 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 OS design issues. The use of Tingu 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. In this way, Tingu helps close the communication gap between OS and driver developers.

Chapter 6

Automatic device driver synthesis with Termite

The Dingo architecture reduces the number of device driver defects by taking a formal approach to modelling the interface between the driver and the OS. In this chapter I extend this approach to formally specify both the OS and the device interfaces of the driver. The resulting specifications exhaustively describe the required driver behaviour and can be used to synthesise a complete driver implementation automatically. Driver synthesis has the potential to dramatically reduce the impact of human error on driver reliability and to cut down on development costs.

6.1 Motivation

The conventional, manual, driver development process is based on two sets of documentation: the device documentation that describes the software interface of the device and the OS documentation that describes OS services that must be implemented and used by the driver. Together these documents define the required correct driver behaviour. The job of the driver programmer is to map OS requests into sequences of device interactions—a straightforward, yet error-prone task.

In this chapter I show that this task can be automated. To this end, both the device and the OS interfaces must be specified formally. Each specification describes possible interactions across the respective interface and relates them to the third specification that defines common behaviours of all devices of the given class. These specifications are processed by a tool called Termite, which generates a complete driver implementation in C that satisfies both specifications.

This work builds on the work on formalising device driver protocols presented in Chapter 5. While Tingu OS protocol specifications do not in themselves provide sufficient information for driver synthesis, they are an important step towards a complete formalisation of driver behaviour as presented below.

Separating device description from OS-related details is a key aspect of the proposed approach. It allows the people with the most appropriate skills and knowledge to develop specifications: device protocol specifications can be developed by device manufacturers, and OS protocol specifications by the OS developers who have intimate knowledge of the OS and the driver support it provides.

In a hand-written device driver, interactions with the device and with the OS are intermingled, leading to drivers that are harder to write and harder to maintain. Termite specifications each deal with a single concern, and thus can be simpler to understand and debug than a full-blown driver.

Device protocol specifications are independent of any OS, so drivers for different OSs can be synthesised from a single specification developed by a device manufacturer, thus avoiding penalising less popular OSs with poor-quality drivers. A further benefit of device and OS separation is that any change in the OS need only be expressed in the OS-protocol specification in order to re-generate all drivers for that OS. This is particularly interesting for Linux, which frequently changes its device driver interfaces from release to release.

Generating code from formal specifications reduces the incidence of programming errors in drivers. Assuming that the synthesis tool is correct, synthesised code will be free of many types of defects, including memory management and concurrency-related defects, missing return value checks, etc. A defect in a driver can occur only as a result of an error in the specification. The likelihood of errors due to incorrect OS protocol specifications is reduced because these specifications are shared by many drivers and are therefore subject to extensive testing.

In contrast, a device specification is developed for a particular device and is only used in synthesising drivers for this device for a small number of OSs. Due to the low-level nature of device protocols, their specifications tend to be more complex than OS specifications and are therefore more likely to contain errors. One avenue for future research is to explore the use of model checking techniques to establish formal correspondence between the actual device behaviour, as defined in its register-transfer-level description, and the Termite specification. While model checking may not be able to guarantee equivalence between the two specifications, it may be useful in finding and eliminating many discrepancies between them, thus substantially increasing the level of confidence that the resulting device specification is correct. The feasibility of this approach is demonstrated by the success of hardware model checking tools like VCEGAR [CJK04] and UCLID [AS04].

Errors in device specifications can be reduced by using model checking techniques to establish formal correspondence between the actual device behaviour, as defined in its register-transfer-level description, and the Termite specification. However, this capability is not yet supported in Termite.

The separation of device and OS interface specifications sets Termite apart from the previous approaches to automatic driver synthesis surveyed in Section 3.6 [WMB03, ZZC03, KSF00, OOJ98]. These previous techniques rely on the driver developer to create a com-

plete model of the driver behaviour in the form of communicating state machines written in a high-level language. This model is then compiled into a low-level implementation language like C. While the use of a high-level domain-specific language offers some reliability benefits, it still requires the developer to construct the driver algorithm manually. In contrast, Termite derives the driver algorithm automatically based on the model of the device and the OS.

While the above discussion is concerned with the technical implications of automatic driver synthesis, the real-world success of this approach depends on device manufacturers and OS developers adopting it.

For device manufacturers, the proposed approach has the potential to reduce driver development effort while increasing driver quality. Furthermore, once developed, a driver specification will allow drivers to be synthesised for any supported OS, thus increasing the OS support for the device.

For OS developers, the quality and reputation of their OS depends greatly on the quality of its device drivers: major OS vendors suffer serious financial and image damage because of faulty drivers [GGP06]. Driver quality can be improved by providing and encouraging the use of tools for automatic driver synthesis as part of driver development toolkits. Since Termite drivers can co-exist with conventional hand-written drivers, migration to automatically-generated drivers can be implemented gradually.

Another concern for OS developers is that acceptance and success of their OS depends largely on compatibility with a wide range of devices. Since device protocol specifications are OS independent, providing support for driver synthesis allows the reuse of all existing Termite device protocol specifications, leading to potential increases in an operating system's base of compatible devices.

6.2 Overview of driver synthesis

Termite generates an implementation of a driver based on a formal specification of its device and OS protocols. The device protocol specification describes the programming model of the device, including its software-visible states and behaviours. The OS protocol specification defines services that the driver must provide to the rest of the system, as well as OS services available to the driver. Given these specifications, Termite produces a driver implementation that translates any valid sequence of OS requests into a sequence of device commands.

This is similar to the task accomplished by a driver developer when writing the driver by hand. In contrast to automatic driver synthesis, however, manual development relies on informal device and OS documentation rather than on formal specifications: The device protocol description is found in the device data sheet, whereas the OS protocol is documented in the driver developer's manual and in the form of comments in the OS source code.

In Termite, the device and the OS interfaces are specified independently and are comprised of different kinds of objects: the device protocol consists of hardware registers and interrupt lines, whereas the OS interface is a collection of software entrypoints and callbacks. How can Termite establish a mapping between the two interfaces, while keeping the associated protocol specifications independent?

The following solution is inspired by conventional driver development practices. Consider, for example, the task of writing a Linux driver for the RTL8139D Ethernet controller [Rea05]. Linux requires all Ethernet drivers to implement the `hard_start_xmit` entrypoint described in the Linux driver developer's manual [CRKH05]:

> `int (*hard_start_xmit) (...);`
> Method that initiates the *transmission of a packet*.

In order to implement this function, the driver developer consults with the RTL8139D device data sheet [Rea05], which describes the transmit operation of the controller as follows:

> Setting bit 13 of the TSD register triggers the *transmission of a packet*, whose address is contained in the TSAD register and whose size is given by bits 0-12 of the TSD register.

While the two documents were written independently by different authors, both of them refer to the act of *packet transmission*, which is the common behaviour of all Ethernet controller devices and is independent of the specific device architecture and OS personality. It allows the driver developer to relate the two specifications and to correctly implement the `hard_start_xmit` function by setting the appropriate device registers.

To generalise this example, both the device and the OS specifications refer to actions performed by the device in the external physical world, e.g., transmission of a network packet, writing a block of data to the disk, or drawing a pixel on the screen. The device specification uses these actions to describe how the device reacts to various software commands. Likewise, the OS specification mentions external device actions when describing the semantics of OS requests.

Together, the set of such external actions characterises a class of similar devices, such as Ethernet controllers or SCSI disks, and is both device and OS-independent. In Termite, these actions are formalised in a separate device-class specification, which is provided to the synthesis tool along with the device and OS specifications.

Figure 6.1 shows a high-level view of the synthesis process. The following subsections elaborate on each of the three specifications involved in driver synthesis.

6.2.1 Device-class specifications

An informal description of a device class can usually be found in the relevant I/O protocol standard. For example, the Ethernet LAN standard, maintained by the IEEE 802.3 working group [IEE], describes common behaviours of Ethernet controller devices, including packet

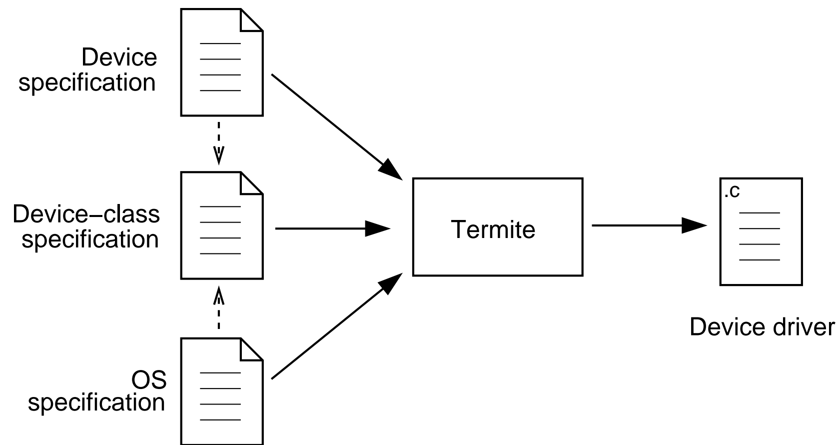


Figure 6.1: Driver synthesis with Termite. Solid arrows indicate inputs and outputs of the synthesis tool; dashed arrows indicate references from the device and OS specifications to the device-class specification.

transmission and reception, link status detection, speed autonegotiation, etc. Other I/O protocol standards include SCSI, USB, AC'97, IEEE 1394, SD, etc.

Such a standard can be used to derive a formal Termite device-class specification for the given type of devices. For interoperability reasons the standard must be agreed upon by all device and OS vendors; therefore a device-class specification based on it is guaranteed to be free of any device or OS dependencies. At the same time, standards are designed to allow freedom of implementation. In particular, they allow hardware optimisations, such as packet buffering and vectored I/O. Therefore, using the standard as the basis for the device-class specification ensures that the specification remains compatible with both basic and advanced implementations of the standard.

In Termite, device-class functionality is formalised as a set of events. The majority of events correspond to the various types of interactions between the device and its external physical environment, as described by the standard. The Ethernet controller device class, for example, includes such events as packet transmission, completion of autonegotiation, and link status change.

The remaining events describe changes to internal device configuration settings. Many of these settings are defined by widely adopted standards. For example, all Ethernet controllers are required to support retrieving and changing of the controller MAC address. Other settings are only supported by a subset of devices or even a single device. For instance, while most Ethernet controllers implement some form of multicast filters, the exact operation and format of these filters varies across devices. Device manufacturers implement such settings in order to differentiate their products from the competition; therefore it is important that Termite is capable of generating drivers that support these advanced capabilities.

Three options are available in Termite for including non-standard device capabilities in

device-class specifications. First, if the given capability is shared by several devices, it can be included in the common device-class specification as an optional feature that need not be supported by all implementations. Alternatively, if this capability is unique to the device, the device manufacturer has to develop an extended version of the device-class specification describing the given capability. The extension must be strictly incremental, so that it can be combined with an existing OS protocol specification that is not aware of the new function. The device manufacturer must also develop an extended OS protocol specification in order to enable access to this function in a specific OS. Finally, separate device-class and OS protocol specifications must be created in order to synthesise a driver for a one-of-a-kind device whose functionality cannot be described as a superset of any existing device class.

One important concern related to device-class specifications is determining who should develop and maintain these specifications. Since many of these specifications must be shared by all device and OS vendors, they should ideally be developed by an independent party that would ensure that the specification faithfully reflects the appropriate I/O protocol standard and is free of device or OS-specific features. One possibility is to delegate this task to the regulatory body responsible for maintaining the relevant I/O standard. Alternatively, the device-class specification can be developed by a consortium of OS and device manufacturers.

6.2.2 Device specifications

The device specification models the software view of the device behaviour. It describes device registers accessible to the driver and device reactions to writing or reading of the registers. A device's reaction depends on its current register values and state, e.g., whether the device has been initialised, is busy handling another request, etc. The device reaction may include actions such as updating register values, generating interrupts, and performing one or more external actions defined in the device-class specification.

A device specification can be constructed in several ways. First, it can be derived from informal device documentation. Hardware vendors often release detailed data sheets, describing the interface and operation of the device. Such a data sheet is intended to provide sufficient information to enable a third party to develop a driver for the device.

Figure 6.2 shows a specification of the transmit command of the RTL8139D device derived from its data sheet (for now, we write the specification in English, rather than in the formal Termite language, which will be introduced in Section 6.4). Here, steps 1, 3, and 4 represent actions of the device protocol, whereas the packet transfer performed in step 2 is a device-class event. The latter cannot be observed directly by the software, but can be controlled indirectly. Specifically, the driver can initiate packet transfer by setting bit 13 of the TSD register and is notified of the transfer completion by an interrupt and a flag in the status register.

The problem with this approach to obtaining device specifications is that informal de-

1. The TSD register is updated by the software.
2. If bit 13 of the TSD register changed from 0 to 1, the device performs a packet transfer. The physical address and size of the packet are determined by TSD and TSAD registers.
3. The device sets a flag in the interrupt status register to signal successful completion of the transfer.
4. An interrupt signal is generated.

Figure 6.2: Specification of the transmit operation of the RTL8139D controller derived from its data sheet.

To transmit a network packet:

1. Write the packet address to the TSAD register;
2. Write the TSD register, storing the packet size in bits 0 to 12 to and setting bit 13 to 1;
3. Wait for an interrupt from the controller;
4. Read the interrupt status register to make sure that the transfer was successful;
5. Packet transfer complete.

Figure 6.3: Specification of the transmit operation of the RTL8139D controller derived from a reference driver implementation.

vice documentation seldom undergoes adequate quality assurance. As a result, it tends to be incomplete and inaccurate. A specification derived from such a data sheet is likely to reproduce these defects, in addition to extra ones introduced in the process of formalisation.

Another approach to the construction of a device specification is to distil it from an existing driver implementation provided by the device vendor or a third party. The source code of the driver defines sequences of commands that must be issued to the driver in order to perform a specific operation. A Termite specification of the device is obtained by separating these device control sequences from OS-specific details.

Figure 6.3 shows a specification of the RTL8139D transmit operation extracted from the source code of the Linux driver for this device. While this specification is functionally equivalent to the one in Figure 6.2, it is substantially different in style. The specification obtained from the data sheet describes how the device reacts to software commands in different states, but does not explicitly define the order in which these commands should be issued to achieve a particular goal. In contrast, the specification derived from the existing driver source code specifies an explicit command sequence. The Termite synthesis tool, described here, can handle both types of specifications.

The main drawback of this approach to constructing device specifications is that it relies on someone to develop at least one driver for the device manually and thus contradicts our goal of eventually replacing manual driver development with automatic synthesis. Besides,

similarly to informal documentation, a device driver may contain errors, which are carried over to the resulting specification. However, the likelihood of such errors in a well-tested driver is much lower.

The third way to construct a device specification is to derive it from the *register-transfer level (RTL)* description of the device written in a *hardware description language (HDL)*. This requires abstracting away most of the internal logic and modelling only interface modules, responsible for interaction with the host CPU.

Since the RTL description is used as the source for generating the actual device circuit, it constitutes an accurate and complete model of the device operation. Therefore, this method of obtaining device specifications is the preferred one. Furthermore, since the RTL description has well-defined formal semantics, one could potentially use model checking techniques to verify that the resulting Termite specification constitutes a faithful abstraction of the device behaviour, thus eliminating errors introduced during manual abstraction. I have not implemented support for such model checking yet.

Many I/O devices contain a general-purpose processor or a simple microcontroller capable of executing programs stored in the device memory. Such programs are known as device firmware. The behavioural specification of such a device cannot be obtained solely from its RTL description, but needs to be extracted from the combination of RTL and the firmware code.

The main limitation of this approach to obtaining device specifications is that it requires access to the RTL description of the device and its firmware, which are usually part of the device manufacturer's intellectual property. Therefore, the device manufacturer is in the best position to produce such device specifications.

6.2.3 OS specifications

The OS protocol specification defines OS requests that must be handled by the driver, the ordering in which these request can occur and how the driver should respond to each type of request. To this end, it defines a state machine, where each transition corresponds to a driver invocation by the OS, an OS callback made by the driver, or a device-class event. Any of these operations is only allowed to happen if it triggers a valid state transition in the state machine.

This is similar to Tingu OS protocol specifications, the key difference being that Tingu specifications, as described in Chapter 5, specify the ordering of invocations exchanged between the driver and the OS without defining the semantics of these operations. In contrast, Termite OS protocol state machines describe the semantics of OS requests in terms of their external effect, i.e. in terms of device-class events that must be generated in response to the request.

Consider, for example, the fragment of the protocol between the Linux kernel and an Ethernet driver specified in Figure 6.4. This specification states that the driver must respond

1. The OS sends a `hard_start_xmit` request to the driver;
2. *Eventually*, the device completes the transfer of the packet, passed as an argument to `hard_start_xmit`;
3. The driver calls the `dev_kfree_skb_any` function to notify the OS of the packet completion.

Figure 6.4: A fragment of the Ethernet controller driver protocol specification.

to the `hard_start_xmit` request by completing the transfer of the network packet specified in the request. The transfer of a packet is a device-class event. The exact mechanism of generating this event is described in the device specification; the OS specification simply states that the event must occur for the `hard_start_xmit` request to be satisfied.

The specification in Figure 6.4 imposes both safety and liveness constraints on the driver. Safety ensures that the driver does not violate the prescribed ordering of operations, e.g., it is only allowed to send a packet after receiving an appropriate request. Liveness forces certain events to eventually happen, thus guaranteeing forward progress. In this example, after receiving the transmit request, the driver must eventually transfer the packet.

As discussed in Chapter 5, a driver interacts with the OS through several protocols—one for each service provided or used by the driver. Termite allows each OS protocol to be defined independently, in a separate specification. Multiple protocols can then be combined in a driver declaration given to Termite.

6.2.4 The synthesis process

The goal of the Termite synthesis tool is to generate a driver implementation that complies with all relevant protocol specifications. Such an implementation must satisfy the following requirements:

1. **Safety**. The driver must not violate the specified ordering of operations. If the driver issues a device command which raises a device-class event, this event must be allowed in the OS protocol specification in the current state, i.e. the driver should only perform external actions when allowed by the OS protocol. Likewise, every OS callback performed by the driver must correspond to a transition in the OS protocol state machine.

2. **Liveness**. The driver must be able to meet all its goals: whenever the OS protocol state machine is in a state where an event or one of a group of events is required to eventually happen, the driver must guarantee the occurrence of this event within a finite number of steps.

The driver synthesis problem can be formalised as a two-player game [Tho95] between the driver as one player and its environment, which is comprised of the device and the OS,

as the other. The players participate in the game by exchanging commands and responses across driver interfaces. Each player directly controls a subset of interactions: the OS controls requests sent to the driver, the driver controls commands sent to the device and OS callbacks, and the device controls responses to software commands. Rules of the game define legal sequences of interactions between the players and are given by the device and OS protocol specifications (safety). The driver's game objective is to complete any OS request in a finite number of steps (liveness).

The Termite synthesis algorithm computes a winning strategy on behalf of the driver. A winning strategy must guarantee that the driver will achieve its objectives, regardless of how the device and the OS behave, as long as their behaviour remains within rules. The formulation of the problem as a game enables us to employ existing game-theoretic techniques in computing the driver strategy. Details of the Termite synthesis algorithm are presented in Section 6.6.

The resulting strategy constitutes a state machine, where in every state the driver either performs an action, e.g., writes a device register or invokes an OS callback, or waits for an input from the environment, e.g., a completion interrupt from the device, or a call from the OS. This state machine is translated into C code, which can be compiled and loaded in the OS kernel just like a conventional, manually developed driver.

6.3 A trivial example

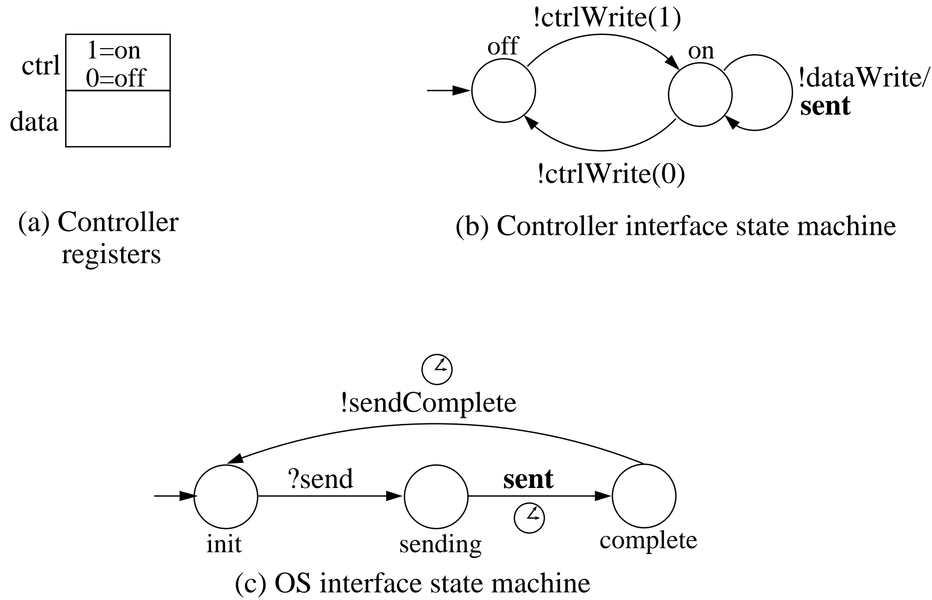
I illustrate the driver synthesis methodology using the example of a driver for a hypothetical trivial network controller device. This example serves to clarify the concepts introduced above. The specification language and the synthesis algorithm used to generate realistic device drivers are presented in the following sections.

The network controller is capable of transferring data over the network, one bit at a time. Its software interface consists of a control register, set to 1 or 0 to switch the device on and off respectively, and a data register, where the software can write a bit of data to be transmitted over the wire (Figure 6.5a).

For the sake of the example, assume that this device represents a class of similar trivial network controller devices. The device-class specification consists of a single event, `sent`. The event is generated every time the controller sends a bit of data. The semantics of this event do not depend on a specific register layout or OS personality, and can therefore be used in both the device and the OS protocols without violating the separation of concerns between the protocols.

Figure 6.5b and Figure 6.5c specify the device and the OS protocols of the driver to be synthesised respectively. State transitions in Figure 6.5 represent interactions between the driver, the device, and the OS and occurrences of device-class events.

According to the device specification in Figure 6.5b, the only command allowed in the initial state of the device protocol is writing 1 (`ctrlWrite(1)`) to the control register to



(a) Controller registers

(b) Controller interface state machine

(c) OS interface state machine

Figure 6.5: Specification of a trivial network controller driver.

switch the device on. Afterwards, the driver may either write a value to the data register (`dataWrite`), sending it on the network, as denoted by the occurrence of the `sent` event, or write 0 to the control register (`ctrlWrite(0)`), returning the device to the off state.[1] The OS protocol (Figure 6.5c) consists of a `send` command from the OS, in response to which the driver must send the data on the network and deliver a `sendComplete` notification to the OS.

Two transitions in Figure 6.5c have timeout labels. Timeout labels provide a simple way to define goals for the driver. If one of the driver protocols is in a state that has one or more outgoing transitions with a timeout label, the driver is not allowed to stay in this state indefinitely but must eventually leave it via a timeout transition.

The synthesis algorithm starts by computing the parallel product of the two state machines in Figure 6.5 resulting in a new state machine that describes all legal behaviours of the system consisting of the driver, the device, and the OS (Figure 6.6). A state of the product state machine corresponds to a pair of states in the original input state machines, e.g., the initial state of the product state machine corresponds to the pair of initial states `<off,init>`. In the product, actions that belong to different protocols occur independently, whereas the `sent` event shared by the two protocols only occurs when allowed by both input state machines. Timeout labels are transferred from the input FSMs to the corresponding transitions of the product state machine.

The product state machine defines the rules of the game between the driver, the device, and the OS. The next step is to find a winning strategy on behalf of the driver among all behaviours allowed by the product state machine. If such a strategy exists, it can be obtained from the product state machine by eliminating all transitions that do not lead to the goal.

[1]For clarity of presentation, the actual value written to the data register is ignored.

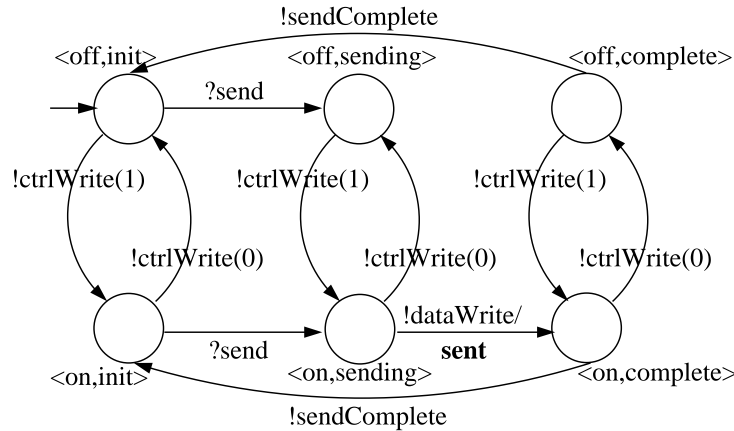


Figure 6.6: Product state machine representing combined constraints of the two driver protocols.

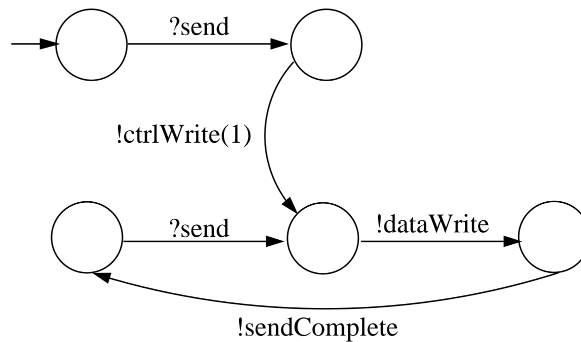


Figure 6.7: Synthesised driver algorithm.

The algorithm for computing such a strategy will be presented in Section 6.6.

A winning strategy for the state machine in Figure 6.6 is shown in Figure 6.7. It is easy to see that this state machine implements the expected correct driver behaviour: upon receiving a send request from the OS, it switches the controller on (`!ctrlWrite(1)`), transmits the data (`!dataWrite`), and sends a completion notification to the OS (`!sendComplete`), leaving the driver ready for the next request with the device powered on. The OS specification did not include powering down the device, so the `off` state becomes unreachable after the first time a message is sent.

The final step is to generate code to implement the state machine representing the calculated winning strategy. This step will be described in Section 6.6.

6.4 The Termite specification language

Device, OS, and device-class specifications in Termite are developed using the Termite specification language. This section outlines requirements that drove the design of the language and presents its syntax and semantics.

The Termite specification language must be suitable for modelling the behaviour of

complex I/O devices, containing multiple functional units. Such a system cannot be feasibly described by explicitly enumerating its states (as was done in the above example). A high-level language, providing constructs to express hierarchical composition of communicating state machines, is required.

The language should also provide flexible data definition and manipulation facilities. Examples of data in Termite specifications include device registers, DMA buffers, and operating system I/O request descriptors.

Some existing languages satisfy these requirements. In particular, hardware description languages are well-suited for describing device behaviour, and are sufficiently general to model arbitrary state-machine-based systems. At the moment, however, Termite does not provide an HDL frontend.

The previous chapter presented the Tingu language designed specifically for writing driver protocol specifications. Tingu provides facilities for dealing with data and concurrency and is thus a natural candidate for the Termite specification language. The main limitation of Tingu is that it relies on a visual formalism to describe the behavioural part of driver protocols. While Statecharts allow clear and concise specification of simple protocols, they quickly hit their limit when dealing with more complex behaviours. Even specifying driver-OS protocols using Statecharts required sacrificing completeness for readability in some cases (see Section 5.4). Device protocols tend to be more complex than OS protocols and cannot be practicably specified using Statecharts.

Therefore, device and OS protocol state machines in Termite are described using a textual algebraic notation rather than a visual language. Other elements of the Tingu syntax, including components, protocols, methods, variables, and dependencies, are reused unmodified, with one extension: Termite defines a new type of method to model device-class events, in addition to `in` and `out` methods. Device-class events represent internal device state transitions that are not observable at the driver interface, therefore these events are declared using the `internal` keyword. Termite currently does not allow synthesising drivers that spawn new ports at runtime; therefore subport declarations are not allowed in Termite protocol specifications. In summary, the Termite protocol specification language differs from Tingu in the following ways:

- In Termite, protocol state machines (i.e., the `transitions` section of the protocol specification) are expressed using the algebraic formalism presented below instead of Statecharts.
- Termite supports the `internal` method qualifier, in addition to `in` and `out` qualifiers.
- Termite does not support subport declarations.

The rest of this section introduces the formalism used to define protocol state machines in Termite. The formalism is based on the LOTOS process calculus [LOT89]. A protocol

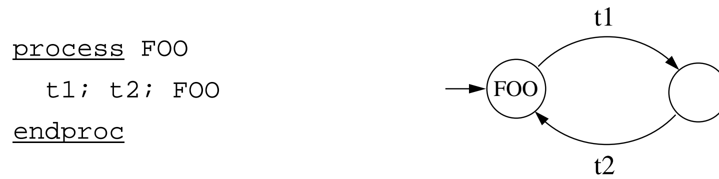


Figure 6.8: A simple Termite process and the corresponding state machine.

state machine consists of transitions combined into processes. A transition is triggered by a protocol method invocation or an occurrence of a device-class event. A process describes valid sequences of transitions.

A simple process is a sequence of transitions named after its initial state. For example, Figure 6.8 shows a process that allows transitions `t1` and `t2` to occur before returning to the initial state `FOO`. An individual transition in a process has the following syntax:

```
<transition> ::= <trigger> "[" <guard> "]" "/" <action> ":timed"
```

with optional `<guard>`, `<action>`, and `timed` components. Here, `<trigger>`, `<guard>`, and `<action>` follow the syntax defined for Tingu state transition labels (see Section 5.3 and Section A.2).

The trigger can be either a protocol method or one of two special triggers: `await` and `timeout`. The `await` trigger denotes a transition that is taken as soon as its guard evaluates to truth, without waiting for a particular method to be invoked. The `timeout` transition is triggered after the amount of time specified by its argument. It can be used to model time-dependent device and OS behaviours.

The guard specifies a predicate on protocol variables and method arguments that must hold for the transition to be enabled. The action defines how protocol variables are updated when the transition is taken.

The `timed` keyword is used to specify liveness requirements of the protocol: whenever the state machine is in a state with one or more enabled timed transitions, one of these transitions must eventually be taken. If the timed transition corresponds to an outbound or internal method, then it is the responsibility of the driver to invoke this method or force the corresponding device-class event to occur; otherwise, the other side of the protocol (the device or the OS) guarantees that it will invoke the corresponding method of the driver.

Processes are composed out of individual transitions using sequential and parallel composition operators which are listed in Table 6.1 and described in more detail below. The complete syntax of Termite processes in *Backus-Naur Form (BNF)* is presented in Section A.4.

Deadlock The deadlock process (denoted `stop`) cannot perform any transitions and never terminates.

Name	**Syntax**	**Semantics**	**Description**
Deadlock	`stop`		Inactive process
Termination	`exit`	e	Successful termination
Action prefixing	`t; P`	t, P	A process that performs transition t and then behaves as process P
Choice	`P1 [] P2`	P1, P2	A process that behaves either as process P1 or as process P2
Conditional	`if[cond]P1[]else P2`	true, cond, false, P1, P2	A process that behaves as P1 if cond holds or as P2 otherwise
Sequential composition	`P1 >> P2`	P1, exit, P2	Start process P2 after P1 terminates
Preemption	`P1 [> P2`	P1, t, P2	Execution of P1 is interrupted when the first transition of P2 occurs
Parallel composition	`P1|[m1...mn]|P2`	P1, m1, mn, P2	P1&P2 run concurrently, synchronising on methods m1..mn, i.e., one of these methods can only occur when it triggers a state transition in both processes
Interleaving	`P1|||P2`	P1, P2	P1&P2 run concurrently; transitions can arbitrarily interleave
Named process	`process P...endproc`		A process that can be instantiated by name

Table 6.1: Termite process syntax. Circles denote individual states. Squares denote entire state machines, aka processes.

Termination The purpose of the `exit` process is solely to perform successful termination, after which it behaves like the deadlock process `stop`. Formally,

$$\texttt{exit} \xrightarrow{\texttt{e}} \texttt{stop}$$

where `e` is the successful termination event. The above notation states that the left-hand side process (`exit`) can perform a transition labeled `e` and then behave as the right-hand side process (`stop`).

Similarly to the deadlock process, the termination process is not capable of performing any externally visible actions. The difference between `exit` and `stop` processes is in how they compose with other processes, as shown below.

Action prefixing The action prefixing operator defines a process that performs a specified transition and then behaves as another process P:

$$\texttt{t;P} \xrightarrow{\texttt{t}} \texttt{P}$$

Choice The choice operator P1[]P2 defines a process that can behave as either process P1 or process P2. The choice is resolved at the instance when the process performs its first transition. If this transition belongs to P1 then the choice process must continue behaving as P1; otherwise, if it is a transition defined in P2 then the choice process continues as P2. Termite does not allow non-deterministic choice when both P1 and P2 contain the same initial transition. Formally,

$$\frac{\mathtt{P1} \xrightarrow{\mathtt{t1}} \mathtt{P1'}}{\mathtt{P1[]P2} \xrightarrow{\mathtt{t1}} \mathtt{P1'}} \qquad \frac{\mathtt{P2} \xrightarrow{\mathtt{t2}} \mathtt{P2'}}{\mathtt{P1[]P2} \xrightarrow{\mathtt{t2}} \mathtt{P2'}}$$

The first of the above clauses states that if P1 is capable of performing transition t1 and transforming into process P1′ then process P1[]P2 is capable of the same behaviour. The second clause is analogous for P2.

Conditional The conditional operator alters the process's behaviour based on the protocol variable values:

$$\frac{(\mathtt{expr}) \wedge (\mathtt{P1} \xrightarrow{\mu_1} \mathtt{P1'})}{(\mathtt{if(expr)P1[]else\ P2}) \xrightarrow{\mu_1} \mathtt{P1'}} \qquad \frac{(\neg\mathtt{expr}) \wedge (\mathtt{P2} \xrightarrow{\mu_2} \mathtt{P2'})}{(\mathtt{if(expr)P1[]else\ P2}) \xrightarrow{\mu_2} \mathtt{P2'}}$$

where $\mu_\mathtt{i}$ is either a transition or a successful termination event.

Sequential composition The sequential composition operator executes two processes sequentially, with the second one starting execution as soon as the first one terminates successfully:

$$\frac{\mathtt{P1} \xrightarrow{\mathtt{t}} \mathtt{P1'}}{(\mathtt{P1} >> \mathtt{P2}) \xrightarrow{\mathtt{t}} (\mathtt{P1'} >> \mathtt{P2})} \qquad \frac{(\mathtt{P1} \xrightarrow{\mathtt{e}} \mathtt{stop}) \wedge (\mathtt{P2} \xrightarrow{\mu} \mathtt{P2'})}{(\mathtt{P1} >> \mathtt{P2}) \xrightarrow{\mu} \mathtt{P2'}}$$

Preemption The preemption operator aborts the execution of a process when another (preempting) process performs a transition:

$$\frac{\mathtt{P1} \xrightarrow{\mathtt{t}} \mathtt{P1'}}{(\mathtt{P1}[> \mathtt{P2}) \xrightarrow{\mathtt{t}} (\mathtt{P1'}[> \mathtt{P2})} \qquad \frac{\mathtt{P1} \xrightarrow{\mathtt{e}} \mathtt{stop}}{(\mathtt{P1}[> \mathtt{P2}) \xrightarrow{\mathtt{e}} \mathtt{stop}} \qquad \frac{\mathtt{P2} \xrightarrow{\mu} \mathtt{P2'}}{(\mathtt{P1}[> \mathtt{P2}) \xrightarrow{\mu} \mathtt{P2'}}$$

To avoid nondeterminism, Termite does not allow the preempting and the preempted processes to have transitions with the same trigger enabled simultaneously.

Parallel composition The parallel composition operator models concurrent execution of two processes. Concurrency here means that the parallel composition can perform an action that either component is ready to perform. The parallel processes synchronise on a subset of methods. One of these methods can only be invoked if both processes are ready for it. When the method is invoked, it triggers a state transition in both processes. The effect of the parallel composition operator is defined by the following rules:

$$\frac{\mathtt{P1} \xrightarrow{\mathtt{t1}} \mathtt{P1'},\ \mathtt{t1} = \mathtt{trig1[g1]/a1},\ \mathtt{trig1} \notin \{\mathtt{m1...mn}\}}{(\mathtt{P1|[m1...mn]|P2}) \xrightarrow{\mathtt{t1}} (\mathtt{P1'|[m1...m2]|P2})}$$

If the first process can perform transition `t1` whose label consists of trigger `trig1`, guard `g1`, and action `a1`, where `trig1` is not one of the methods in the synchronisation set {`m1`...`mn`}, then the composition can perform the same transition. The second process remains in its initial state.

Similar rule holds for the second process (parallel composition is commutative):

$$\frac{\mathtt{P2} \xrightarrow{\mathtt{t2}} \mathtt{P2}',\ \mathtt{t2} = \mathtt{trig2}[\mathtt{g2}]/\mathtt{a2},\ \mathtt{trig2} \notin \{\mathtt{m1}...\mathtt{mn}\}}{(\mathtt{P1}|[\mathtt{m1}...\mathtt{mn}]|\mathtt{P2}) \xrightarrow{\mathtt{t2}} (\mathtt{P1}|[\mathtt{m1}...\mathtt{m2}]|\mathtt{P2}')}$$

$$\frac{(\mathtt{P1} \xrightarrow{\mathtt{t1}} \mathtt{P1}') \wedge (\mathtt{P2} \xrightarrow{\mathtt{t2}} \mathtt{P2}'),\ \mathtt{t1} = \mathtt{trig}[\mathtt{g1}]/\mathtt{a1},\ \mathtt{t2} = \mathtt{trig}[\mathtt{g2}/\mathtt{a2}],\ \mathtt{trig} \in \{\mathtt{m1}...\mathtt{mn}\}}{(\mathtt{P1}|[\mathtt{m1}...\mathtt{mn}]|\mathtt{P2}) \xrightarrow{\mathtt{trig}[\mathtt{g1} \wedge \mathtt{g2}]/\{\mathtt{a1};\mathtt{a2};\}} (\mathtt{P1}'|[\mathtt{m1}...\mathtt{mn}]|\mathtt{P2}')}$$

If both processes are ready to perform transitions with the same trigger and this trigger belongs to the synchronisation set, then the parallel composition can perform a transition with the same trigger. This transition is guarded by the conjunction of the original guards and its effect on protocol variables is defined as the concatenation of the original actions: $\{\mathtt{a1};\mathtt{a2};\}$. Note that the commutativity of parallel composition requires that `a1` and `a2` are commutative.

Finally, the parallel composition successfully terminates when both constituent processes terminate:

$$\frac{(\mathtt{P1} \xrightarrow{\mathtt{e}} \mathtt{stop}) \wedge (\mathtt{P2} \xrightarrow{\mathtt{e}} \mathtt{stop})}{(\mathtt{P1}|[\mathtt{m1}...\mathtt{mn}]|\mathtt{P2}) \xrightarrow{\mathtt{e}} \mathtt{stop}}$$

Interleaving Interleaving is a special case of parallel composition where the synchronisation set is empty:

$$\mathtt{P1}|||\mathtt{P2} \equiv \mathtt{P1}|[]|\mathtt{P2}$$

Named process A named process is declared as follows:

```
process <process-name>
  <behavioural-expression>
endproc
```

where `behavioural-expression` is written using the constructs presented above. Once declared, the process can be instantiated by name and used with any of the above operators (in place of `P`, `P1`, or `P2`).

This feature facilitates modular specifications and behaviour reuse. Most importantly, it extends the expressive power of the language by allowing recursive behaviours, such as the one in Figure 6.8.

6.4.1 Restrictions on device-class specifications

As mentioned above, the Termite specification language is used to develop all three specifications involved in driver synthesis: the device protocols specification, the OS protocol

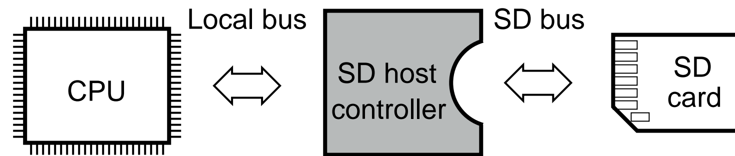


Figure 6.9: SD host controller device.

specification and the device-class specification. While the OS and the device protocol specifications can use the complete expressive power of the language, device-class specifications are subject to the following restrictions:

1. A device-class specification defines events that are either internal to the device or occur at the boundary between the device and its external physical environment. In the Termite specification language such events must be declared as methods with the `internal` specifier to denote the fact that they cannot be directly observed at the software interface of the device. Thus, device-class specifications only contain `internal` methods.

2. A device-class specification defines common behaviours of a class of devices but does not define the ordering in which these behaviours are allowed to occur. As such, it may not have `transitions`, `variables`, and `dependencies` sections.

The second restriction reflects a limitation of the present Termite implementation. In principle, associating ordering constraints with device-class events is a good idea, since it would enable separate validation of device and OS protocols: any correct device or OS protocol state machine must refine the behaviour defined by the device-class specification. However, such validation has not been implemented.

6.5 A realistic example

This section illustrates the various concepts introduced in the previous sections using a complete Termite specification of a driver for a *Secure Digital (SD)* host controller device. This device was chosen since it is simple enough to allow a concise description, yet represents a real device allowing us to show what Termite specifications for real hardware look like.

6.5.1 Overview

An SD host controller acts as a bridge between the host CPU and an SD card device connected to the SD bus (Figure 6.9). The SD bus architecture is host-centric with the host controller issuing commands on the bus and the SD card executing the commands and sending responses back to the host controller.

This example targets an open-source SD host controller implementation published by the OpenCores project [Edv]. The device protocol specification presented here has been

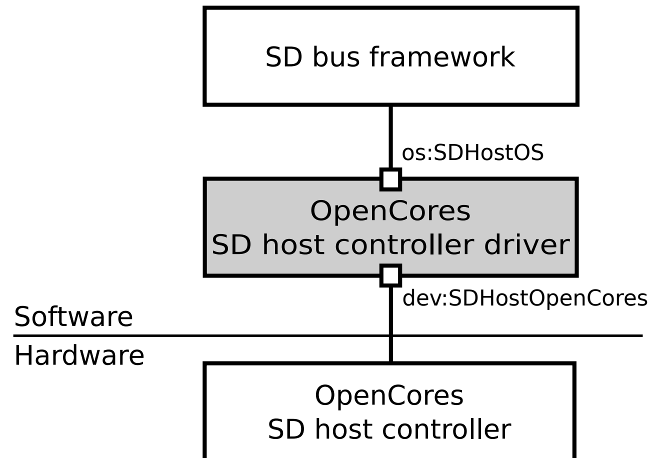


Figure 6.10: SD host controller driver and its ports.

```
component sdhc_opencores
{
ports:
  SDHostClass class;
  SDHostOS os<class/class>;
  SDHostOpenCores dev<class/class>;
};
```

Figure 6.11: The SD host controller driver component specification.

manually derived from the register-transfer level Verilog HDL design of the controller.

Figure 6.10 shows the architectural view of the driver that must be synthesised. The driver implements two ports: one for interaction with the OS and one for interaction with the device. This architecture assumes that the host CPU can read and write device registers directly, so the driver does not need to use a bus transport service to access the device.

The Termite declaration of the driver is shown in Figure 6.11. It lists the three specifications that the driver must comply with: the SD host controller device-class specification and the OS and the device protocol specifications. The device class is modelled as a separate protocol that only contains internal methods representing device-class events. The device and the OS protocols both declare dependencies on the device-class protocol, which allows establishing the relative ordering of interface method invocations and device-class events.

The following subsections consider each of the three protocols in detail.

In order to keep the example concise, I have chosen not to model all of the device and OS features. In particular, in modelling the controller and the SD bus behaviour I specify simplified SD command and response formats and abstract away the SD error recovery, power management, and hot plugging behaviours. Likewise, I define a simplified OS protocol, which is loosely based on the analogous protocol in Linux, but does not support advanced configuration options and multiple-block data transfers.

Note that results for synthesising drivers from unabridged device specifications are presented in Section 6.8.

6.5.2 The device-class specification

As mentioned in Section 6.2, a device-class specification must capture common external behaviour of a family of similar devices. For SD host controller devices, the common behaviour is defined in the SD bus specification [SD 06], maintained by the SD Association.

According to this specification, the controller operates by issuing SD commands, consisting of a 6-bit command index and a 32-bit argument, on the bus. Upon completion of the command, the card sends back a 32-bit response. Two commands involve an additional data transfer stage that follows the response: the block read command is followed by the transfer of a 512-byte block from the card; the block write command is followed by the transfer of a 512-byte block to the card. The argument of both commands is the block address in the card memory.

The controller also manages several bus configuration parameters, of which we model just one—the bus clock frequency. The frequency can be modified by applying a divisor to the basic clock.

Figure 6.12 shows the specification of the SD host controller device class. It is defined as an protocol with only internal methods (corresponding to device-class events). The first two events (lines 19 and 21) are generated when the device is turned on and ready for use and when it is inactive respectively. The remaining events describe command and data transfers and bus frequency change operations outlined above.

Since the device class only defines the set of events shared between the OS and the device specifications and does not impose constraints on the ordering of these events, a device-class specification does not define a state machine.

6.5.3 The OS protocol specification

The OS protocol specification (Figure 6.13) describes the service that an SD host controller driver must provide to the OS. It declares data types exchanged between the OS and the driver (lines 3–15), protocol methods, including OS requests and driver responses (lines 16–33), device-class events whose occurrence is restricted by the protocol (lines 33–42), variables used to describe the state of the protocol (lines 43–46), and the protocol state machine (starting in line 47).

The protocol state machine defines the driver's required reactions to requests in terms of device-class events that must occur before the driver sends a completion notification to the OS. This pattern is illustrated, for instance, in lines 48–50, which specify how the driver must handle a `probe` request from the OS. Before replying to this request in line 50, the driver must ensure that the `class.on` event in line 49 occurs. This event refers to the `on` event defined in the device-class specification (Section 6.5.2). In other words, the

```
1 protocol SDHostClass
2 {
3 types:
4   /*SD error conditions*/
5   enum sdh_status_t {
6     SDH_SUCCESS   = 0,  /* success */
7     SDH_ECRC      = 1,  /* CRC error */
8     SDH_ETIMEOUT  = 2   /* timeout */
9   };
10  /*SD command attributes*/
11  struct sdh_cmd_t {
12    unsigned<6> index; /*cmd index*/
13    unsigned<32> arg;  /*argument*/
14    bool data;         /*command with data?*/
15    bool response;     /*response expected?*/
16  };
17 methods:
18  /*Device initialized*/
19  internal on();
20  /*Device inactive*/
21  internal off();
22  /*Successful completion of a command stage*/
23  internal commandOK(sdh_cmd_t command, unsigned<32> response);
24  /*Command stage failed*/
25  internal commandError(sdh_cmd_t command, sdh_status_t status);
26  /*Successful completion of a data stage*/
27  internal blockTransferOK(
28      paddr_t mem_addr,        //host address of the block
29      unsigned<32> card_addr); //card address
30  /*Data transfer failed*/
31  internal blockTransferError(
32      paddr_t mem_addr,
33      unsigned<32> card_addr,
34      sdh_status_t status);
35  /*Bus frequency changed*/
36  internal busClockChange(u32 divisor);
37 };
```

Figure 6.12: The SD host controller device-class specification.

precondition for delivering the `probeComplete` notification to the OS is that the device is successfully initialised. Note that the state machine does not describe how this precondition is satisfied. This information is part of the device specification, considered below.

After completing the initialisation, the protocol state machine executes the `REQUESTS`

```
1 protocol SDHostOS
2 {
3 types:
4   struct sdhc_request_t {
5     unsigned<32> opcode; /*cmd index*/
6     unsigned<32> arg;    /*cmd argument*/
7     bool response;       /*response present*/
8     bool data_present;   /*data stage present*/
9     paddr_t block;       /*block address*/
10  };
11  struct sdhc_response_t {
12    int<32> cmd_status;   /*cmd stage status*/
13    unsigned<32> response;/*response from card*/
14    int<32> data_status;  /*data stage status*/
15  };
16 methods:
17  /*Probe and initialise the controller*/
18  in  probe ();
19  out probeComplete (int<32> status);
20
21  /*Shut down the device and terminate the driver*/
22  in  remove ();
23  out removeComplete ();
24
25  /*Issue a command on the bus, followed by a data
26    transfer stage (if the command involves one)*/
27  in  request (sdhc_request_t request);
28  out requestComplete (sdhc_response_t response);
29
30  /*Change the bus clock frequency*/
31  in  setClock (unsigned<32> divisor);
32  out setClockComplete ();
33 dependencies:
34  SDHostClass class {
35    restricts on;
36    restricts off;
37    restricts commandOK;
38    restricts commandError;
39    restricts blockTransferOK;
40    restricts blockTransferError;
41    restricts busClockChange;
42  };
```

Figure 6.13: The SD host controller driver OS protocol specification *(continued on the next page)*.

```
43 variables:
44   unsigned<32> m_reqDiv;/*requested divisor*/
45   sdhc_request_t m_request;
46   sdhc_response_t m_response;
47 transitions:
48   probe;
49   class.on:timed;
50   probeComplete[$status==0]:timed;
51   REQUESTS
52
53 where
54
55   process REQUESTS
56     /*A remove request*/
57     remove;
58     /*The driver must switch the device off
59       before calling the completion method*/
60     class.off:timed;
61     removeComplete:timed;
62     /*The protocol state machine terminates*/
63     exit
64   []
65     /*Command without a data transfer stage*/
66     request[$request.data_present==false]
67            /m_request=$request;
68     (
69       class.commandOK
70         [($command.index==m_request.opcode)&&
71          ($command.arg==m_request.arg)&&
72          ($command.response==m_request.response)&&
73          ($command.data==false)]
74         /{m_response.cmd_status=0;
75           m_response.response=$response;}:timed;
76       requestComplete[$response==m_response]:timed;
77       REQUESTS
78     []
79       class.commandError
80         /{m_response.cmd_status=$status;
81           m_response.response=0;}:timed;
82       requestComplete[$response==m_response]:timed;
83       REQUESTS
84     )
```

Figure 6.13: The SD host controller driver OS protocol specification *(continued)*.

```
85  []
86    /*Command 17 (block read request) and command 24
87      (block write request) are handled similarly*/
88    request[($request.data_present==true)&&
89            (($request.opcode==17)||($request.opcode==24))]
90           /m_request=$request;
91    (
92      /*Command stage completes successfully*/
93      class.commandOK[($command.index==m_request.opcode)&&
94                      ($command.arg==m_request.arg)&&
95                      ($command.response==m_request.response)&&
96                      ($command.data==true)]
97                     /{m_response.cmd_status=0;
98                       m_response.response=$response;}:timed;
99      (
100       /*Data transfer stage completes successfully*/
101       class.blockTransferOK[$mem_addr==m_request.block]
102                              /m_response.data_status=0 : timed;
103       requestComplete[$response==m_response]:timed;
104       REQUESTS
105     []
106       /*Data transfer fails*/
107       class.blockTransferError/m_response.data_status=$status;
108       requestComplete[$response==m_response]:timed;
109       REQUESTS
110     )
111   []
112     /*Command stage fails*/
113     class.commandError/{m_response.cmd_status=$status;
114                         m_response.response=0;
115                         m_response.data_status=0;};
116     requestComplete[$response==m_response]:timed;
117     REQUESTS
118   )
119 []
120   /*A setClock request*/
121   setClock/m_reqDiv=$divisor;
122   /*The driver must change the bus clock divisor to the
123     requested value before calling the completion method*/
124   class.busClockChange[$divisor==m_reqDiv]:timed;
125   setClockComplete:timed;
126   REQUESTS
127 endproc
128 };
```

Figure 6.13: The SD host controller driver OS protocol specification *(the end)*.

process (line 55). In its initial state, this process performs a choice between incoming requests defined in lines 57, 66, 88, and 121 using the choice operator `[]` (lines 64, 85, and 119). This means that the driver must wait for the OS to call one of these methods.

We consider one of the four requests in detail in order to illustrate the use of protocol variables and transition guards in OS specifications. Line 66 describes a request to issue an SD command without a data transfer stage. Upon receiving the request, the request structure is copied to the `m_request` variable. The state machine defines two possible outcomes of this request: either the device successfully completes the command (line 69) or the command completes with an error (line 79). The guard in lines 70–73 states that the command transferred on the bus must correspond to the one requested by the OS. In case of success, the response received from the SD card is saved in the `m_response` variable (line 75) and delivered to the OS by calling the `requestComplete` method (line 76). If the command fails, the driver stores the error code in the `m_response` variable (line 80) and reports the failure to the OS via the `requestComplete` method (line 82).

Handling of the other requests is explained using comments in Figure 6.13. Note that for a command with a data stage the driver must also wait for the data transfer to complete (lines 101–103) before signalling success.

6.5.4 The device protocol specification [2]

While the OS protocol specification determines the structure of the driver by defining requests that it must handle in every state, the device protocol specification reflects the structure and operation of the device hardware.

Figure 6.14 shows the internal architecture of the device in question, as defined in its HDL specification and Table 6.2 describes its registers. The device supports the bus mastering capability and uses DMA to transfer data blocks to and from the host memory. It is connected to an interrupt line, which is used to signal the completion of command and data stages to the driver.

The interface logic of the controller consists of the register file, the Command Master module, responsible for issuing commands without a data stage, the Data Master module, which handles block transfer commands, the BD module, which buffers block descriptors before passing them to the Data Master, and the Clock Divider module, which controls the SD bus clock.

The Termite specification of the device is shown in Figure 6.15. Ellipses are used throughout the specification to indicate omission of code fragments; the complete specification is given in Appendix C.

The types section describes the structure of device registers (only the command register is shown) and the data structure used to represent block descriptors inside the device

[2]The OpenCores SD controller device protocol specification presented in this section was completed in collaboration with Balachandra Mirla.

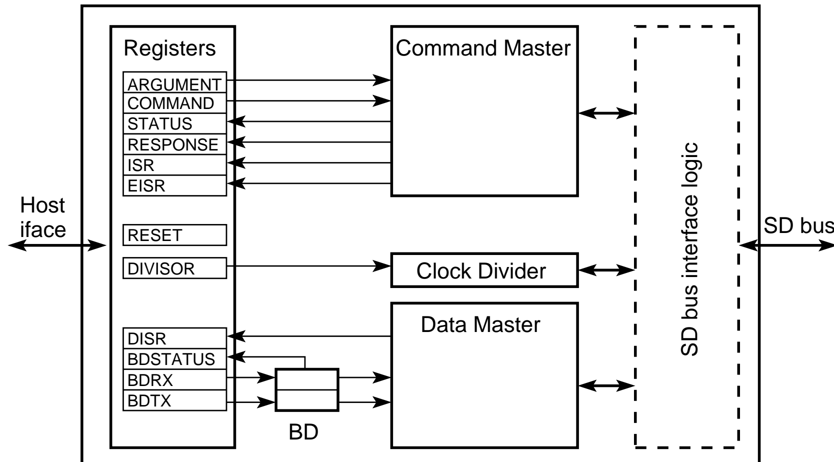


Figure 6.14: The OpenCores SD host controller device architecture.

(line 11). The methods section declares methods comprising the interface between the driver and the device. These include register read and write methods (lines 18–20) and the interrupt method (line 21). The `out` specifier of the method argument in line 19 indicates that the value of this argument is returned by the method.

Interface variables (lines 34–41) describe internal device registers and signals that influence the device's software-observable behaviour. The `m_command_reg` variable models the content of the command register of the device. The `m_new_command` variable models the internal signal that notifies the Command Master about a new command issued through the argument register, while the `m_data_command` variable indicates whether the new command will be followed by a data transfer stage. The `m_command` variable describes the command currently being handled by the Command Master. Finally, `m_tx_descr` and `m_rx_descr` represent block descriptors stored inside the `BD` module.

The device protocol state machine has been manually derived from the RTL design of the device in the Verilog HDL which is used to synthesise the device hardware. The structure of the state machine reflects the device architecture shown in Figure 6.14 and its behaviour models the device's reactions to software commands. The order in which these software commands are issued by the driver is determined automatically by the synthesis algorithm. In some cases, however, the device protocol state machine specifies an explicit sequence of commands that must be issued to the device in a certain state. For example, the state transitions in lines 43–44 force the driver to reset the device, by writing a 1 followed by a 0 to the reset register, before issuing any other commands.

This is necessary due to a limitation of the current Termite synthesis algorithm, namely, it requires the values of all protocol variables to be known in any state. This assumption requires the device to behave deterministically with respect to commands issued by the software, i.e., the state of the device must be completely determined by the sequence of commands issued by the driver. This assumption does not hold in the initial state of the device protocol, since at this point the device registers may have arbitrary values. The

Register: ARGUMENT (Command arg)		Size: 32	Access: RW
[31:0]	CMDA	Command argument value	

Register: COMMAND (Command)		Size: 16	Access: RW
[15:10]	CMDI	Command index	
[9:2]	RESERVED	Reserved	
[1:0]	RTS	Response type	
		00: No response	
		01: Response	

Register: STATUS (Card status)		Size: 16	Access: R
[15:1]	RESERVED	Reserved	
0	CICMD	Command inhibit	

Register: RESPONSE (Command response)		Size: 32	Access: R
[31:0]	CRSP	Response from the card	

Register: RESET (Software reset)		Size: 8	Access: RW
[31:0]	RESERVED	Reserved	
0	SRST	Software reset	

Register: ISR (Normal intr status)		Size: 16	Access: RW
15	EI	Error interrupt	
[14:1]	RESERVED	Reserved	
0	CC	Command complete	

Register: EISR (Error intr status)		Size: 16	Access: RW
[31:2]	RESERVED	Reserved	
1	CCRC	CRC error	
0	CTE	Command timeout	

Register: DIVISOR (Clock divisor)		Size: 8	Access: RW
[7:0]	CLKD	Clock divisor	

Register: BDSTATUS (Buffer descr status)		Size: 16	Access: R
[15:8]	FBRX	Free RX descriptors	
[7:0]	FBTX	Free TX descriptors	

Register: DISR (Data intr status)		Size: 16	Access: RW
[15:2]	RESERVED	Reserved	
1	TRE	Transmission error	
0	TRS	Transmission successful	

Register: BDRX (RX buffer descriptor)		Size: 32	Access: W
[31:0]	BDRX		

Register: BDTX (TX buffer descriptor)		Size: 32	Access: W
[31:0]	BDTX		

Table 6.2: SD host controller registers.

problem is overcome by issuing a sequence of commands that bring the device to a known state. In this example, writing 1 to the software reset register resets all device registers to

their known default values (lines 43).

Once the reset is complete, the device is ready to handle commands, as indicated by the `on` device-class event in line 45. Line 46 invokes the `SDHOST` process (line 49), which describes normal operation of the device. This process consists of four concurrent subprocesses, corresponding to four device modules in Figure 6.14: `REGISTERS`, `COMMAND_MASTER`, `DATA_MASTER`, and `CLOCK_DIVIDER` (lines 50–54). These subprocesses communicate via variables, which can be read and updated by any process. In addition, `COMMAND_MASTER` and `DATA_MASTER` synchronise on the `off` device-class event. This means that the event can only occur when it is enabled in both processes, i.e., the device becomes inactive when both the Command and the Data Master are inactive.

The preemption operator in line 55 specifies that writing 1 to the reset register (line 56) interrupts normal device operation and resets all registers to their default values. A subsequent writing of 0 to this register (line 58) resumes device operation from the clean state.

We illustrate how device registers are modelled using the argument register as an example. Reading the register (line 69) returns its current value. The effect of writing to this register depends on the state of the command inhibit flag (the `CICMD` field of the status register). If the flag is set, meaning that the Command Master is currently busy handling a command (line 72), a write to this register updates the register value, but does not trigger any signals. A write to the argument register when the flag is not set (line 75) triggers the `m_new_command` signal (line 77) and sets the command inhibit flag (line 79).

The `m_new_command` signal wakes up the Command Master waiting for this signal in line 85. It uses values in the `COMMAND` and `ARGUMENT` registers to form an SD command (lines 86–88) and sends it over the bus. Upon completion of the command, it raises an interrupt (line 91). The outcome of the command is reflected in the interrupt status registers (`ISR` and `EISR`) and the response register. This is another situation where the assumption of deterministic device behaviour is violated, since the exact device state is not known to the software until it reads the values of these registers. Therefore, the device protocol state machine specifies a sequence of register reads required to restore the determinism invariant (lines 92–94). Lines 95–96 acknowledge the interrupt by setting the interrupt status registers to zero. Lines 97–112 generate the `commandOK` or `commandError` device-class event, depending on whether the command was successful or not; they also reset the command inhibit flag, indicating that the Command Master is ready to handle another command.

```
1 protocol SDHostOpenCores
2 {
3 types:
4   /* device registers */
5   struct command_reg {
6     unsigned<2> RTS;
7     unsigned<8> RESERVED;
8     unsigned<6> CMDI;
9   };
10  ...
11  struct block_descr {
12    unsigned<32> mem_addr;  /*memory address*/
13    unsigned<32> card_addr; /*card address*/
14  };
15
16 methods:
17  /*register read/write methods */
18  out write_command_reg (command_reg v);
19  out read_command_reg (out command_reg v);
20  ...
21  in irq ();
22
23 dependencies:
24  SDHostClass class {
25    restricts on;
26    restricts off;
27    restricts commandOK;
28    restricts commandError;
29    restricts blockTransferOK;
30    restricts blockTransferError;
31    restricts busClockChange;
32  };
33
34 variables:
35  command_reg m_command_reg;
36  ...
37  unsigned<1> m_new_command;
38  unsigned<1> m_data_command;
39  sdhost_command_t m_command;
40  block_descr m_tx_descr;
41  block_descr m_rx_descr;
```

Figure 6.15: The OpenCores SD host controller device specification *(continued)*.

```
42 transitions:
43   write_reset_reg[$v.SRST==1]/{m_comand_reg=0;m_status_reg=0;...};
44   write_reset_reg[$v.SRST==0];
45   class.on;
46   SDHOST
47  where
48
49   process SDHOST
50   (REGISTERS
51   |||
52     (COMMAND_MASTER |[class.off]| DATA_MASTER)
53   |||
54     CLOCK_DIVIDER)
55   [>
56     write_reset_reg[$v.SRST == 1]/{m_comand_reg=0;
57                                    m_status_reg=0; ...};
58     write_reset_reg[$v.SRST==0];
59     SDHOST
60   endproc
61
62   process CLOCK_DIVIDER
63     write_clock_div_reg/m_clock_div_reg=$v;
64     class.busClockChange[$divisor==m_clock_div_reg.CLKD];
65     CLOCK_DIVIDER
66   endproc
67
68   process REGISTERS
69       read_argument_reg[$v==m_argument_reg];
70       REGISTERS
71     []
72       write_argument_reg[m_status_reg.CICMD==1]/m_argument_reg=$v;
73       REGISTERS
74     []
75       write_argument_reg[m_status_reg.CICMD==0]
76                         /{m_argument_reg=$v;
77                           m_new_command=1;
78                           m_data_command=0;
79                           m_status_reg.CICMD=1;};
80       REGISTERS
81     []
82       ...
83   endproc
```

Figure 6.15: The OpenCores SD host controller device specification *(continued on the next page)*.

```
84  process COMMAND_MASTER
85    await[m_new_command==1]
86         /{m_command.index=m_command_reg.CMDI;
87           m_command.arg=m_argument_reg.CMDA;
88           m_command.response=m_command_reg.RTS;
89           m_command.data=m_data_command;
90           m_new_command=0;};
91    irq : timed;
92    read_isr_reg/m_isr_reg=$v : timed;
93    read_eisr_reg/m_eisr_reg=$v : timed;
94    read_response_reg/m_response_reg=$v : timed;
95    write_isr_reg[$v==0] : timed;
96    write_eisr_reg[$v==0] : timed;
97    (
98    if[m_isr_reg.CC == 1]
99      class.commandOK
100       [($command==m_command) &&
101        ($response==m_response_reg.CRSP)]
102       /m_status_reg.CICMD=0 : timed;
103     COMMAND_MASTER
104   []
105   else
106     class.commandError
107       [($command==m_command) &&
108        ($status==(m_eisr_reg.CCRC ?
109                   SDH_CMD_ECRC : SDH_CMD_ETIMEOUT))]
110       /m_status_reg.CICMD=0 : timed;
111     COMMAND_MASTER
112   )
113 []
114   class.off;
115   exit
116 endproc
117
118 process DATA_MASTER
119   ...
120 endproc
```

Figure 6.15: The OpenCores SD host controller device specification *(continued)*.

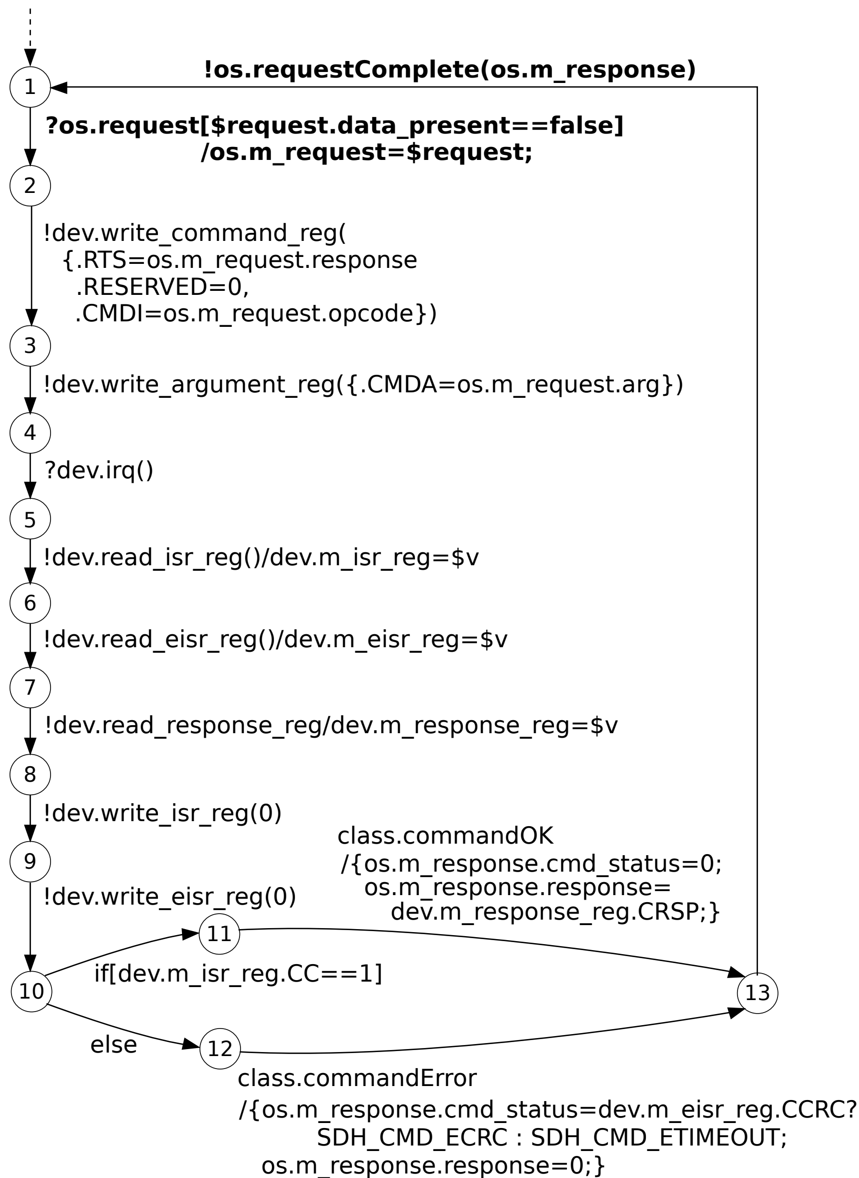


Figure 6.16: A fragment of the SD host controller driver state machine generated by Termite. Exclamation marks denote method invocations performed by the driver; question marks denote driver methods invoked from the environment. OS protocol methods are shown in bold font; device protocol methods are in normal font; device-class events are in italics.

6.5.5 The driver state machine

The Termite synthesis algorithm combines the device and the OS protocol specifications and produces a driver state machine which defines the driver's reactions to OS requests and device interrupts. The algorithm for constructing such a state machine will be presented in Section 6.6. Figure 6.16 shows a fragment of the resulting state machine responsible for handling SD commands without a data stage.

In the initial state (state 1), the driver waits for a request from the OS.

If this request satisfies the guard of the transition from state 1 to state 2 ([$request.data_present==false]), then this transition is taken.

According to the OS protocol specification, either the commandOK or the commandError device-class event must occur before the driver can send a completion notification to the OS (Figure 6.13, lines 69 and 79). To achieve this goal, the driver state machine performs the following sequence of device interactions: it issues the command specified in the request to the device through the command and argument registers (transitions $2 \rightarrow 3$ and $3 \rightarrow 4$), waits for an interrupt from the device (transition $4 \rightarrow 5$), reads the interrupt status registers and the response register (transitions $5 \rightarrow 6$, $6 \rightarrow 7$, and $7 \rightarrow 8$), and acknowledges the interrupt (transitions $8 \rightarrow 9$ and $9 \rightarrow 10$). If the command completed without an error (transition $10 \rightarrow 11$), the commandOK device-class event occurs (transition $11 \rightarrow 13$); otherwise (transition $10 \rightarrow 12$), the commandError event occurs. In either case, the fields of the m_response variable of the OS protocol are set to reflect the status of the command. Finally, the driver invokes the OS completion callback ($13 \rightarrow 1$).

6.6 The synthesis algorithm

As outlined in Section 6.3, the Termite synthesis algorithm proceeds in three steps. In the first step, it computes the parallel product of all driver protocol state machines, which represents all legal (safe) behaviours of the driver. In the second step, it finds a winning strategy among all legal behaviours, which guarantees liveness, i.e. ensures that the driver achieves its goals in any state. In the third step, the winning strategy is translated into a driver implementation in C.

In the trivial example presented in Section 6.3, states of individual protocol state machines and their product were enumerated explicitly. This is infeasible when dealing with realistic device and OS protocols. Therefore Termite relies on a compact symbolic representation of protocol state machines as described below.

6.6.1 Symbolic representation of protocol state machines

The symbolic representation of a protocol state machine defines the state space of the protocol in terms of state variables. Protocol state transitions are specified in terms of symbolic constraints on the values of protocol variables in the source and destination states.

Formally, a protocol state machine is a tuple

$$P = \langle S, i, M, G, T \rangle$$

where $S = V_1 \times ... \times V_n$ is the state space formed by the Cartesian product of the domains of state variables; $i \in S$ is the initial variables assignment; $M = M_{in} \cup M_{out} \cup M_{int}$ is the set of protocol methods, comprised of inbound, outbound and internal methods; $G \subset S \times S$ is the goal relation; $T \subset M^+ \times Guards \times Actions$ is the state transition relation.

M^+ represents the set of protocol methods extended with the nil method, which denotes a transition without a trigger: $M^+ = M \cup \{nil\}$. $Guards$ and $Actions$ are the sets of all Tingu guard and action expressions respectively.

For every method $m \in M^+$, we define two operators: $Dom()$ and $Range()$. $Dom(m)$ returns the domain of the method, i.e., the set of its input arguments assignments; $Range(m)$ returns the range of the method, i.e., the set of its output arguments (arguments declared with the `out` qualifier) assignments. If the method does not have any input or output arguments, then the corresponding operator returns a set consisting of a single element: $\{\varnothing\}$ (not an empty set $\varnothing$).

Each element $t \in T$ of the transition relation has the following form:

$$T = \langle m, g, a \rangle$$

where $m \in M^+$ is the trigger of the transition; $g : S \times Dom(m) \times Range(m) \mapsto \{true, false\}$ is the guard, which determines whether the transition is enabled based on its source state and method arguments; $a : S \times Dom(m) \times Range(m) \mapsto S$ is the action, which computes the new values of state variables based on their values in the source state and method arguments.

The transition relation T can be partitioned into uncontrollable and controllable transitions: $T = T_u \cup T_c$. Uncontrollable transitions are triggered by incoming method invocations from the environment ($m \in M_{in}$). The driver may wait for such a transition to occur but it cannot force a specific method to be invoked at a specific instance. Uncontrollable transitions are further subdivided in timed and untimed ones: $T_u = T_{u_timed} \cup T_{u_untimed}$. Controllable transitions include outgoing method invocations, device-class events and nil-transitions ($m \in M_{out} \cup M_{int} \cup \{nil\}$). When the driver is in a state where such a transition is enabled, it can force the transition to be taken.

The goal relation maps each state $s \in S$ into a set of states that must be reached from s to satisfy the protocol liveness requirements. The goal set can be empty, meaning that the protocol does not define any goals in the given state.

Guard, action, and goal relations are defined in terms of symbolic expressions written using the Termite syntax (Section A.2).

A Termite protocol specification is converted into the symbolic representation as follows. First, the protocol state machine in the Termite specification language is expanded into a flat state machine by applying the rules given in Section 6.4 to each Termite operator. Second, each state of the resulting flat state machine is assigned a unique integer number from the range $\{1..num_states\}$. A new protocol variable $state$ is introduced to model the current state of the protocol. Thus, the state space S of the resulting state machine is comprised of the original protocol variable domains and the domain of the $state$ variable: $S = V_1 \times ... \times V_{num_vars} \times \{1..num_states\}$.

The initial state i is computed by assigning the $state$ variable to the initial protocol state and setting all other state variables to zero.

The set M of protocol methods consists of all methods listed in the Termite protocol specification, including dependencies.

The transitions and goals relations are computed as follows. For each state transition $\langle s_1, m[g]/a, s_2\rangle$ of the flat state machine (where s_1 and s_2 are the source and the target states of the transition and $m[g]/a$ is the Termite transition label), the corresponding transition is added to the transition relation: $T := T \cup \{t\}$. The transition t has the same trigger as the original transition (m); its guard is obtained from the original guard by adding a predicate on the value of the $state$ variable ($state == s_1$), and its action is obtained from the original action by adding a statement that updates the value of the $state$ variable ($state = s_2$):

$$t = \langle m, g \wedge (state == s_1), \{a; state = s_2;\}\rangle$$

If the original transition is timed (i.e., the transition label contains the `:timed` keyword) then the goal relation is extended with a new goal:

$$G := G \cup \{\langle \sigma_1, \sigma_2\rangle |\ \sigma_1 \vDash (state == s_1),\ \sigma_2 \vDash (state == s_2)\}$$

where $\vDash$ denotes the "satisfies" relation. This means that in a state that satisfies the symbolic constraint $state == s_1$, the goal set includes all states that satisfy the constraint $state == s_2$.

6.6.2 Computing the product state machine

Given the set $\{P_1, ..., P_k\}$ of driver protocols, Termite computes their parallel product $P = P_1 \parallel ... \parallel P_k$. The parallel product of two protocol state machines, $P_1 = \langle S_1, i_1, M_1, G_1, T_1\rangle$ and $P_2 = \langle S_2, i_2, M_2, G_2, T_2\rangle$ is computed as

$$P_1 \parallel P_2 = \langle S_1 \times S_2\ ,\ \langle i_1, i_2\rangle\ ,\ M_1 \cup M_2\ ,\ G\ ,\ T\rangle$$

where G and T are defined by the following rules.

Goals are carried over from the multipliers to the product:

$$\frac{\langle \sigma_1, \sigma_2\rangle\ \in\ G_1}{\forall \sigma_1', \sigma_2' \in S_2:\ \langle\langle \sigma_1, \sigma_1'\rangle, \langle \sigma_2, \sigma_2'\rangle\rangle \in G}$$

$$\frac{\langle \sigma_1, \sigma_2\rangle\ \in\ G_2}{\forall \sigma_1', \sigma_2' \in S_1:\ \langle\langle \sigma_1', \sigma_1\rangle, \langle \sigma_2', \sigma_2\rangle\rangle \in G}$$

P_1 transitions that are not synchronised with P_2 are carried over to the product state machine:

$$\frac{t = \langle m, g, a\rangle \in T_1,\ m \notin M_2}{\langle m,\ g_{S_2},\ a_{S_2}\rangle \in T}$$

where g_{S_2} and a_{S_2} are extensions of the transition guard and action to the state space of protocol P_2 defined as:

$$g_{S_2}(\sigma_1, \sigma_2, in, out) = g(\sigma_1, in, out)$$

$$a_{S_2}(\sigma_1, \sigma_2, in, out) = \langle a(\sigma_1, in, out), \sigma_2\rangle$$

where $\sigma_1 \in S_1,\ \sigma_2 \in S_2,\ in \in Dom(m),\ out \in Range(m)$.

Similarly, for P_2 transitions:

$$\frac{t = \langle m, g, a\rangle \in T_2,\ m \notin M_1}{\langle m,\ g_{S_1},\ a_{S_1}\rangle \in T}$$

$$g_{S_1}(\sigma_1, \sigma_2, in, out) = g(\sigma_2, in, out)$$

$$a_{S_1}(\sigma_1, \sigma_2, in, out) = \langle \sigma_1, a(\sigma_2, in, out)\rangle$$

Finally, transitions with common triggers are synchronised:

$$\frac{t_1 = \langle m, g_1, a_1\rangle \in T_1,\ \ t_2 = \langle m, g_2, a_2\rangle \in T_2,\ \ m \in M_1 \cap M_2}{\langle m,\ g1_{S_2} \wedge g2_{S_1},\ a\rangle \in T}$$

$$a(\sigma_1, \sigma_2, in, out) = \langle a_1(\sigma_1, in, out),\ a_2(\sigma_2, in, out)\rangle$$

6.6.3 Computing the strategy

The core part of the synthesis algorithm is the procedure that, given the symbolic product of driver protocol state machines $P = \langle S,\ i,\ M,\ G,\ T\rangle$, computes a driver strategy. In a nutshell, for every reachable state of the product, this procedure computes the behaviour that guarantees that the driver will achieve one of the goals defined in this state, in any environment that conforms with protocol specifications.

Since the product state machine contains transitions controlled by the driver (outgoing method invocations) as well as transitions controlled by the environment (incoming driver method invocations by the OS or the device), the problem of computing a driver strategy can be formulated as a two-player reachability game problem [Tho95]. A basic algorithm for solving such games is described by Thomas [Tho95]. Given an origin state and a set of goal states, the algorithm recursively computes the *controllable predecessor* set, i.e. the set of all states from which the goal can be reached in one step. This includes states where the driver can trigger a transition (by invoking the corresponding OS or device method) that will take it to the goal, as well as states where any legal method invocation from the environment takes the driver to the goal. This results in a reachability graph containing all states and transitions of the product state machine from which there exists a strategy leading to the goal. The algorithm terminates when the origin state is added to the graph or when a fix point is reached. In the former case, the reachability graph contains the strategy that the driver must follow to achieve the goal from the origin state. For every state between the origin and the goal, this strategy prescribes that the driver must either perform a specific method invocation or wait for a method invocation from the environment. In the latter case (a fix point is reached), a winning strategy does not exist and the algorithm returns a failure.

Termite implements a variation of this algorithm based on the following heuristic: any path in the product state machine leading from the origin state to the goal is likely to be extensible to a complete winning strategy. This means that if there exists a sequence of driver commands and device responses leading to the goal then this sequence is likely to

belong to a complete strategy that incorporates any valid device responses, not just those included in the sequence. If this is the case then such strategy can be found by extending the initial sequence with all valid device responses.

If this heuristic holds, it allows constructing a winning strategy faster, since finding a single path to the goal is computationally easier than building the reachability graph.

If the heuristic fails and the initial sequence cannot be extended to a complete strategy, the algorithm should backtrack and find another initial sequence. Such a backtrack was not required in any of the drivers that have been synthesised so far; therefore it is currently not implemented, meaning that theoretically Termite may fail to find a winning strategy even if one exists.

6.6.3.1 Formal representation of the driver strategy

The driver strategy computed by Termite constitutes a directed graph where every node corresponds to a subset of the product state space S. These subsets only cover states that are reachable by the driver and may overlap. Edges of the graph represent transitions of the product state machine. There are two types of edges: controllable edges labeled with a controllable transition and uncontrollable edges labeled with an uncontrollable transition.

The graph is constructed in such a way that every node has either only uncontrollable or only controllable outgoing edges, or no edges at all. When the driver reaches a node with only uncontrollable actions (called an uncontrollable node), it stops, waiting for a method invocation from the environment. In a node with only controllable edges (controllable node), the driver performs a method invocation corresponding to one of the edges. All controllable edges in a node are guaranteed to have transitions with mutually exclusive guards, so that exactly one transition is enabled in the node, hence the driver behaviour is deterministic. Upon reaching an edge without outgoing edges, the driver terminates.

The strategy graph has a designated initial node, which corresponds to the initial state of the product state machine.

Formally, the driver strategy graph is represented as a tuple:

$$\langle \mathbb{N},\ \mathrm{i},\ \mathbb{E} \rangle$$

$\mathbb{N} \subset 2^S$ is the set of nodes, where each node corresponds to a subset of states of the product state machine, $\mathrm{i} \in \mathbb{N}$ is the initial node, $\mathbb{E} = \mathbb{E}_u \cup \mathbb{E}_c$ is the set of edges, which consists of uncontrollable edges $\mathbb{E}_u$ and controllable edges $\mathbb{E}_c$.

An uncontrollable edge is defined by its source and target nodes and the corresponding transition in the product state machine. $\mathbb{E}_u \subset \mathbb{N} \times T_u \times \mathbb{N}$.

A controllable edge is defined by its source and target nodes, the corresponding transition in the product state machine, and a function that computes input arguments of the transition trigger given the current values of state variables: $\mathbb{E}_c \subset \mathbb{N} \times T_c \times (S \mapsto \bullet) \times \mathbb{N}$, where $(S \mapsto \bullet)$ denotes the set of all functions over S.

The strategy computed by the Termite synthesis algorithm guarantees that the driver satisfies both safety and liveness requirements in a well-behaved environment.

Safety means that behaviours generated by the strategy comply with driver protocol specifications, namely: (1) any path in the strategy graph corresponds to a valid trace of the product state machine and (2) given a node in the strategy graph with only uncontrollable outgoing edges and a corresponding state of the product state machine, for every uncontrollable transition enabled in the state there exists an edge in the node labeled with this transition.

Liveness means that, whenever in a state with a goal, the driver eventually reaches one of the states in the goal set by following the behaviour prescribed by the graph.

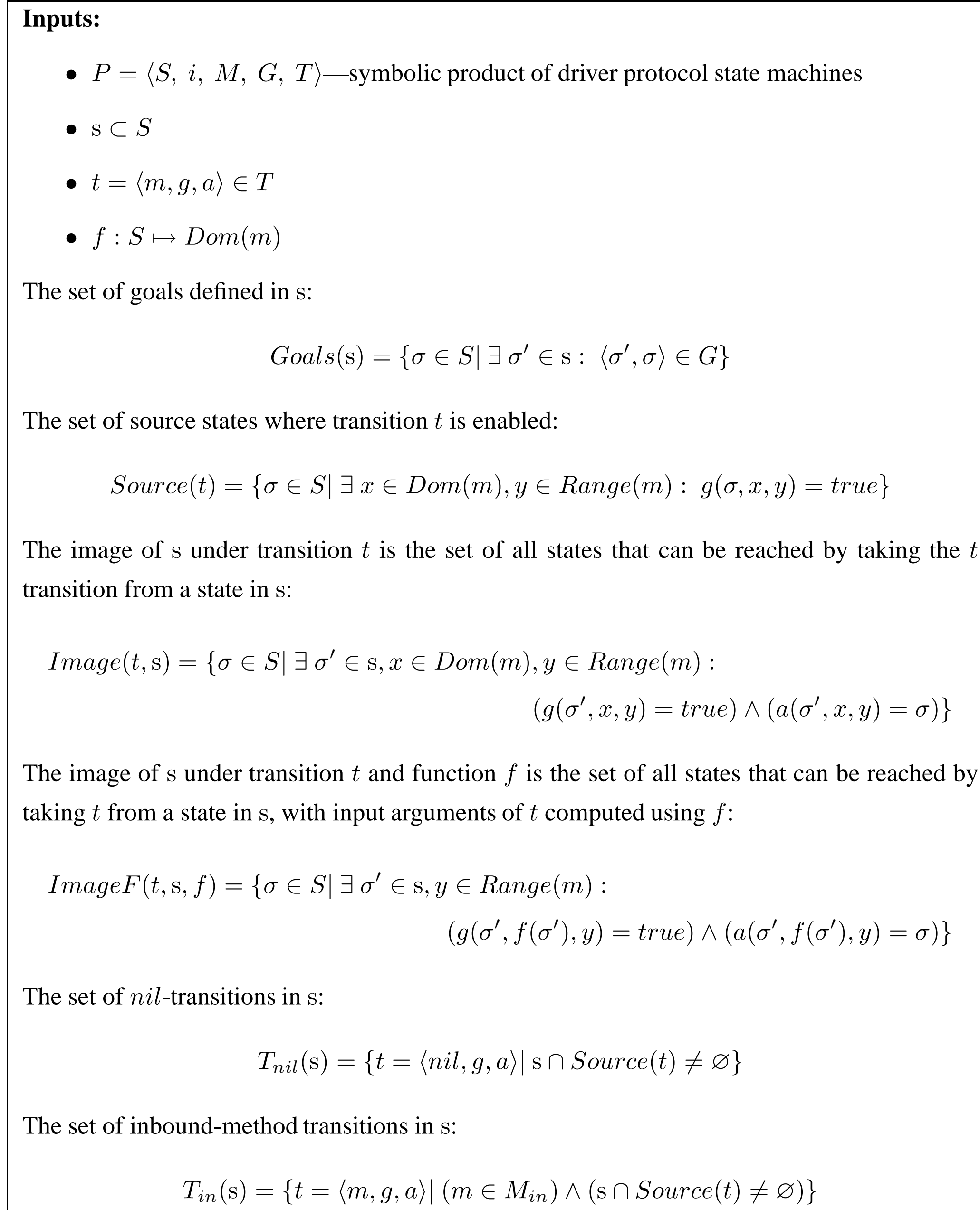
Finally, a well-behaved environment must satisfy the following requirements: (1) any method invoked by the environment triggers a valid state transition in the product state machine (safety), (2) whenever the driver is in a state with one or more timed uncontrollable transitions, the environment will eventually trigger one of the uncontrollable transitions enabled in this state (liveness), (3) if a timed uncontrolled transition becomes enabled infinitely often, it will eventually be taken (fairness).

6.6.3.2 Algorithm for computing the driver strategy

Figure 6.17 defines some helper functions used in computing the strategy. Figure 6.18 shows the main body of the algorithm that computes the driver strategy.

The `Strategy` procedure The procedure explores the state space of the driver incrementally. It maintains the set of reachable strategy nodes that have been added to the graph but the behaviour in which has not been determined yet. On every iteration it selects one of these nodes and computes the driver behaviour in this node. This may result in one or more edges being added to the node and one or more new nodes being added to the graph. The algorithm terminates either when the set of unexplored nodes becomes empty (success) or when the algorithm is unable to determine behaviour in a node.

The initial node of the strategy graph consists of a single state i (line 2). This initial node is added to the set of graph nodes $\mathbb{N}$ and the set of unexplored nodes $\mathbb{U}$. Lines 5–21 describe the main loop of the algorithm. Line 6 removes a node from the set of unexplored nodes. If there are nil-transitions enabled in the corresponding states of the product state machine (line 7), then one of them must be taken (such transitions correspond to `if/else` and `await` Termite operators, which must be executed as soon as they become enabled). Lines 8–10 add edges for all such nil-transitions to the graph. The $\mathbf{AddControllableTransition}$ procedure, described below, adds an edge labeled with transition t to node s. The last argument of the procedure is the function used to compute transition arguments. Since nil-transitions have no arguments, function $f_{\varnothing}$, which maps any input to $\{\varnothing\}$, is used here.



Inputs:

- $P = \langle S,\ i,\ M,\ G,\ T\rangle$—symbolic product of driver protocol state machines
- $\mathrm{s} \subset S$
- $t = \langle m, g, a\rangle \in T$
- $f : S \mapsto Dom(m)$

The set of goals defined in s:

$$Goals(\mathrm{s}) = \{\sigma \in S |\ \exists\, \sigma' \in \mathrm{s} :\ \langle \sigma', \sigma\rangle \in G\}$$

The set of source states where transition t is enabled:

$$Source(t) = \{\sigma \in S |\ \exists\, x \in Dom(m), y \in Range(m) :\ g(\sigma, x, y) = true\}$$

The image of s under transition t is the set of all states that can be reached by taking the t transition from a state in s:

$$Image(t, \mathrm{s}) = \{\sigma \in S |\ \exists\, \sigma' \in \mathrm{s}, x \in Dom(m), y \in Range(m) : \\ (g(\sigma', x, y) = true) \wedge (a(\sigma', x, y) = \sigma)\}$$

The image of s under transition t and function f is the set of all states that can be reached by taking t from a state in s, with input arguments of t computed using f:

$$ImageF(t, \mathrm{s}, f) = \{\sigma \in S |\ \exists\, \sigma' \in \mathrm{s}, y \in Range(m) : \\ (g(\sigma', f(\sigma'), y) = true) \wedge (a(\sigma', f(\sigma'), y) = \sigma)\}$$

The set of nil-transitions in s:

$$T_{nil}(\mathrm{s}) = \{t = \langle nil, g, a\rangle |\ \mathrm{s} \cap Source(t) \neq \varnothing\}$$

The set of inbound-method transitions in s:

$$T_{in}(\mathrm{s}) = \{t = \langle m, g, a\rangle |\ (m \in M_{in}) \wedge (\mathrm{s} \cap Source(t) \neq \varnothing)\}$$

Figure 6.17: Helper functions used in computing the driver strategy.

Line 11 checks whether there are any goals defined in s. If not then in this node the driver simply waits for invocations from the environment (lines 12–14). Otherwise, the **MoveToGoal** procedure is used to determine behaviour in the node. If it finds a path to the goal, it adds the corresponding edges in node s and returns $true$. Otherwise, it returns $false$ and the synthesis algorithm fails (line 17).

The AddControllableTransition procedure Figure 6.19 presents the **AddControllableTransition** procedure. Line 2 checks whether the set of states obtained by taking transition t with input arguments computed using f from s has already been added to the graph. If so, then an edge from s to this existing node is added to the

```
1:  procedure Strategy(P)
    Input : P = ⟨S, i, M, G, T⟩—the product of driver protocol state machines
    Output : ⟨ℕ, i, 𝔼⟩—computed driver strategy
    Returns : true—strategy computed successfully; false—failed to compute strategy
2:      i := {i}
3:      ℕ := {i}
4:      𝕌 := {i}
5:      while 𝕌 ≠ ∅ do
6:          s := Remove(𝕌) // Remove an element from 𝕌
7:          if T_nil(s) ≠ ∅ then
8:              for all t ∈ T_nil(s) do
9:                  AddControllableTransition(s, t, f_∅)
10:             end for
11:         else if Goals(s) = ∅ then
12:             for all t ∈ T_in(s) do
13:                 AddUncontrollableTransition(s, t)
14:             end for
15:         else
16:             res := MoveToGoal(s, Goals(s))
17:             if res = false then
18:                 return false
19:             end if
20:         end if
21:     end while
22:     return true
23: end procedure
```

Figure 6.18: The main Termite algorithm for computing the driver strategy.

graph. Otherwise (line 4), the new node is first added to the set of graph nodes (line 6) and to the set of unexplored nodes (line 7) and then a transition from s to the new nodes is added to the graph (line 8).

The AddUncontrollableTransition procedure This procedure (Figure 6.20) is similar, except the target of the transition is computed using the $Image$ function, which returns the set of states obtained by taking transition t from s with all possible argument values.

The MoveToGoal procedure The **MoveToGoal** procedure finds a path from a source set of states to a goal set. The sought path must consist of only controllable transitions and timed uncontrollable transitions (i.e., transitions that the driver can trigger itself or that

1: **procedure** $\mathbf{AddControllableTransition}((\mathrm{s},\ t,\ f))$
Inputs : $\mathrm{s} \subset S,\ t = \langle m, g, a\rangle \in T_c,\ f : S \mapsto Dom(m)$
2: **if** $\exists\, \mathrm{s}_2 \in \mathbb{N} : ImageF(t, \mathrm{s}, f) = \mathrm{s}_2$ **then**
3: $\mathbb{E}_c := \mathbb{E}_c \cup \{\langle \mathrm{s}, t, f, \mathrm{s}_2\rangle\}$
4: **else**
5: $\mathrm{s}_2 := ImageF(t, \mathrm{s}, f)$
6: $\mathbb{N} := \mathbb{N} \cup \{\mathrm{s}_2\}$
7: $\mathbb{U} := \mathbb{U} \cup \{\mathrm{s}_2\}$
8: $\mathbb{E}_c := \mathbb{E}_c \cup \{\langle \mathrm{s}, t, f, \mathrm{s}_2\rangle\}$
9: **end if**
10: **end procedure**

Figure 6.19: The $\mathbf{AddControllableTransition}$ procedure.

1: **procedure** $\mathbf{AddUncontrollableTransition}(\mathrm{s},\ t)$
Inputs : $\mathrm{s} \subset S,\ t = \langle m, g, a\rangle \in T_u$
2: **if** $\exists\, \mathrm{s}_2 \in \mathbb{N} : Image(t, \mathrm{s}) = \mathrm{s}_2$ **then**
3: $\mathbb{E}_u := \mathbb{E}_u \cup \{\langle \mathrm{s}, t, \mathrm{s}_2\rangle\}$
4: **else**
5: $\mathrm{s}_2 := Image(t, \mathrm{s})$
6: $\mathbb{N} := \mathbb{N} \cup \{\mathrm{s}_2\}$
7: $\mathbb{U} := \mathbb{U} \cup \{\mathrm{s}_2\}$
8: $\mathbb{E}_u := \mathbb{E}_u \cup \{\langle \mathrm{s}, t, \mathrm{s}_2\rangle\}$
9: **end if**
10: **end procedure**

Figure 6.20: The $\mathbf{AddUncontrollableTransition}$ procedure.

are guaranteed to be eventually triggered by the environment). To this end, it constructs a reachability tree to the goal (Figure 6.21). The root of the tree is the goal set. Each node in the tree represents a set of states from which there exists a transition to a node one step closer to the root. The transition is represented by the edge that connects the two nodes.

In constructing the reachability tree, the $\mathbf{MoveToGoal}$ algorithm relies on two auxiliary procedures: $\mathbf{UPre}$ and $\mathbf{CPre}$. Given a set of states s and a timed uncontrollable transition t, the $\mathbf{UPre}$ procedure computes the predecessor $\hat{\mathrm{s}}$ of s under t (Figure 6.21). The predecessor is the set of all states from which transition t triggered with any arguments leads to a state in s:

$$UPre(\mathrm{s}, t) = \{\sigma \in S |\ (\sigma \in Source(t)) \wedge (Image(t, \{\sigma\}) \subset \mathrm{s})\}$$

The $\mathbf{CPre}$ procedure takes a set of states s' and a controllable transition t' and computes the predecessor $\hat{\mathrm{s}}'$ of s' under t', i.e., a set of states where there exists a function f such

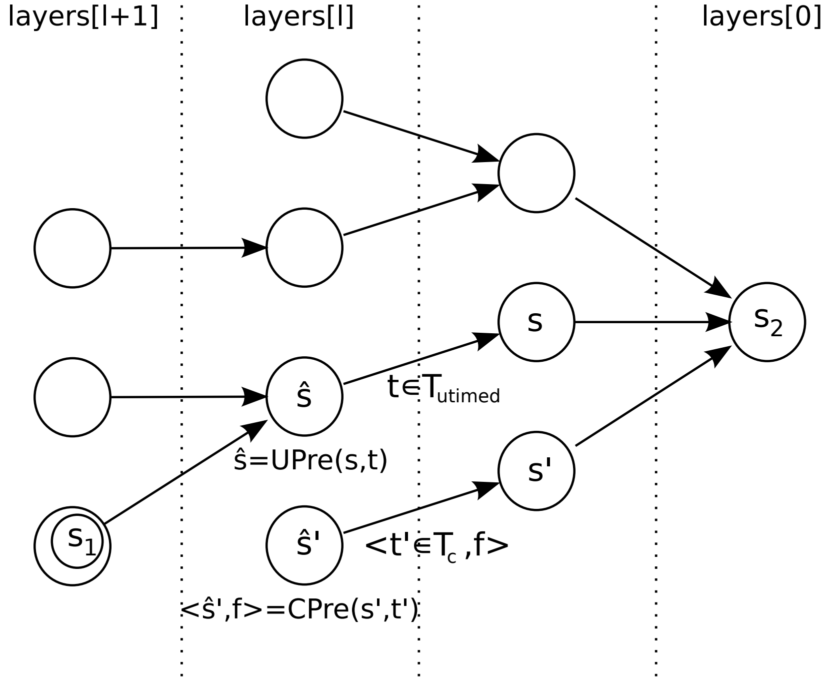


Figure 6.21: Example of a reachability tree from the source set s_1 to the goal set s_2 constructed by the **MoveToGoal** procedure.

that transition t' triggered with arguments computed with f leads to s'. The procedure returns both the computed set $\hat{\mathrm{s}}'$ and the function f. Formally, the tuple $\langle \hat{\mathrm{s}}', f \rangle$ returned by $CPre(\mathrm{s}', t')$ satisfies the condition:

$$ImageF(t', \hat{\mathrm{s}}', f) \subset \mathrm{s}'$$

The implementation of **CPre** and **UPre** is described in Section 6.6.4.

The **MoveToGoal** procedure (Figure 6.22) starts by creating the root node of the tree (line 2). The main iteration of the algorithm (lines 4–33) adds a new tree layer and checks whether the source set s_1 belongs to one of the new nodes. To this end it iterates through all nodes in the previous layer (line 5). For each node, it first enumerates all timed uncontrollable transitions (line 6) and computes the predecessor of the node under each transition (line 7). If the predecessor is not empty and is not contained in one of existing nodes (line 8), then it is added to the tree. Line 10 checks whether the source set s_1 is contained within the predecessor set. If so, a path from the source to the goal has been found, which determines the behaviour of the driver in s_1. Since the first transition in the path is an uncontrollable transition, the driver should wait for an input from the environment. Hence all enabled uncontrolled transitions must be added to the strategy graph node corresponding to the source set. This is accomplished by lines 11–13 before returning from **MoveToGoal** in line 14.

If a solution has not been found in the previous step, the algorithm enumerates or controllable transitions and computes the predecessor of the node under each transition (line 19). If the predecessor is not empty and is not contained in one of existing nodes

```
1:  procedure MoveToGoal(s_1, s_2)
    Inputs : s_1 ⊂ S, s_2 ⊂ S
2:      layers[0] := {s_2}
3:      l := 0
4:      loop
5:          for all s ∈ layers[l] do
6:              for all t ∈ T_u_timed do
7:                  pre := UPre(s, t)
8:                  if (pre ≠ ∅) ∧ (Find(layers, pre) = false) then
9:                      layers[l + 1] := layers[l + 1] ∪ {pre}
10:                     if s_1 ∈ pre then
11:                         for all t' ∈ T_in(s_1) do
12:                             AddUncontrollableTransition(s_1, t')
13:                         end for
14:                         return true
15:                     end if
16:                 end if
17:             end for
18:             for all t ∈ T_c do
19:                 ⟨pre, f⟩ := CPre(s, t)
20:                 if (pre ≠ ∅) ∧ (Find(layers, pre) = ∅) then
21:                     layers[l + 1] := layers[l + 1] ∪ {pre}
22:                     if s_1 ∈ pre then
23:                         AddControllableTransition(s, t, f)
24:                         return true
25:                     end if
26:                 end if
27:             end for
28:         end for
29:         if layers[l + 1] = ∅ then
30:             return false
31:         end if
32:         l := l + 1
33:     end loop
34: end procedure
```

Figure 6.22: The **MoveToGoal** procedure.

(line 20), then it is added to the tree. If the new node contains the source set (line 22), then a controllable edge corresponding to transition t and function f is added to the strategy graph.

The algorithm terminates when either a solution is found or the reachability tree is complete (no new nodes have been added at the last iteration) (line 29). In the latter case, the algorithm returns a failure.

6.6.4 Computing the strategy symbolically

All operations involved in computing the driver strategy are performed symbolically. To this end, driver states and state transitions are represented and manipulated in the form of symbolic constraints on state variables and method arguments. For instance, given the following transition of the product state machine:

$$t = m[(v_1 == 1)\&\&(a_1 == v_2)]/v_3 = a_1$$

where v_1, v_2, and v_3 are state variables and a_1 is an argument of method m, the corresponding symbolic constraint is:

$$(v_1 == 1) \wedge (a_1 == v_2) \wedge (v_3' == a_1) \wedge (v_1' == v_1) \wedge (v_2' == v_2)$$

Here, v_1', v_2', and v_3' represent values of state variables in the target state of the transition. The first three clauses in the above expression are derived directly from transition guard and action; the fourth and fifth clauses $(v_1' == v_1) \wedge (v_2' == v_2)$ are derived from the fact that the transition does not modify variables v_1 and v_2, hence their values are the same in the source and the target state. In general, any state transition can be represented as a boolean formula over predicates, which are written as Termite expressions. Termite stores these formulae in *Disjunctive Normal Form (DNF)*.

All operations involved in computing the driver strategy, including functions in Figure 6.17 and **CPre** and **UPre** procedures are performed symbolically. For instance, the image of set s described by constraint $v_2 == 0$ under transition t can be computed as follows:

$$\begin{aligned} &Image(t, \mathrm{s}) = \{< v_1', v_2', v_3' > | \exists v_1, v_2, v_3, a_1 : \\ &(v_2 == 0) \wedge ((v_1 == 1) \wedge (a_1 == v_2) \wedge (v_3' == a_1) \wedge (v_1' == v_1) \wedge (v_2' == v_2))\} = \\ &\qquad \{< v_1', v_2', v_3' > | \ (v_1' == 1) \wedge (v_2' == 0) \wedge (v_3' == 0)\} \end{aligned}$$

In general, computing the $Image$ function and other above-listed operations requires solving first-order logic equations over Termite expressions. In principle, any *satisfiability modulo theories (SMT)* solver [BT07, BPT07] can be used for this purpose.

Currently, Termite implements its own simple solver. The current implementation of the solver is limited to dealing with Termite expression of the form $(x_1 == x_2)$, $(x_1! = x_2)$, $(x_1 == C)$, and $(x_1! = C)$ (where x_1 and x_2 are state variables or method arguments and `C` is a constant) and their boolean combinations. If it encounters an assignment that is not expressible via such constraints, e.g., $x_1 = x_2 + x_3$, it assumes that the value of x_1 is undefined. This assumption is conservative: if a driver strategy can be found under this

assumption, this strategy is correct. It may, however, prevent Termite from finding a strategy if one exists. For example, a transition whose guard depends on the value of x_1 can never be added to the strategy, which may lead to a failure to find a winning strategy. In practice, this problem is overcome by structuring Termite specifications to avoid the use of such variables in transition guards, which requires extra effort on the part of the specification developer. This limitation is not intrinsic to the Termite approach and can be addressed by the use of a more powerful solver.

6.6.5 Generating code

The last step of the synthesis process is translating the driver strategy computed in the previous step into C. The resulting driver implementation consists of a C structure that describes the state of the driver, a constructor function that creates and initialises an instance of this structure, and a number of entry points, one for each incoming interface method. The state structure contains a field representing every driver state variable, including variables declared in each driver protocol, extra variables introduced to model the state of each protocol state machine, and a $node$ variable that identifies the current node of the strategy graph.

The implementation of a driver entrypoint is generated as follows. It first checks the value of the $node$ variable to identify the uncontrollable edge that corresponds to the given incoming method invocation. It then updates state variables as defined by the action associated with the edge. If the edge leads to a controllable node, the driver chooses an edge in that node whose guard evaluates to truth and takes the corresponding transition by invoking the associated method. Otherwise, the driver returns from the entry point to wait for the next incoming method call.

The resulting driver follows the Dingo architecture and is designed to run within the Dingo runtime framework. In particular, it expects all method invocations to be serialised and never blocks inside a method. The Termite driver synthesis methodology and algorithm are not, however, inherently dependent on the Dingo architecture and can be adapted to generate drivers for other driver frameworks, e.g., native Linux or Windows drivers. Such an adaptation would involve extending the protocol specification language to model concurrent driver invocations and modifying the synthesis algorithm to generate synchronisation code to ensure safe execution of the driver in a multithreaded environment.

6.7 Debugging synthesised drivers[3]

Automatic driver synthesis allows reducing the number of software defects in drivers but does not eliminate them completely. Defects in a synthesised driver can be caused by errors in one of the input specifications or in the synthesis tool. An error in the specification means that the actual device or OS behaviour deviates from the specification in one of the following

[3]The Termite debugger tool presented in this section was developed in collaboration with John Keys.

ways:

- The device or the OS invokes a driver method in a state where this operation is not allowed by the specification.
- The device or the OS invokes a driver method with arguments that do not satisfy the guard condition associated with the corresponding state transition in the specification.
- The device or the OS does not perform a driver invocation that corresponds to a timed transition in the specification (i.e., must occur eventually).
- The device or the OS does not correctly handle an invocation from the driver that is permitted by the specification, e.g., crashes or misbehaves in some other way.

An error in the synthesis tool can result in a driver implementation that violates one of its protocol specifications in one of the following ways:

- The driver performs illegal invocations of device or OS methods, i.e., invokes a method that is not allowed by the appropriate protocol specification in the current state or provides method arguments that violate the guard condition of the corresponding transition.
- The driver does not handle legal method invocations from the device or the OS, i.e., the synthesised driver state machine does not define a transition for a method even though it is permitted by the appropriate protocol specification in the given state.
- The driver fails to achieve some of its goals.

The practical utility of automatic driver synthesis depends on the availability of debugging tools that will help detect and eliminate such errors. One possibility is to use conventional debugging techniques on the synthesised C driver; however debugging automatically generated code is notoriously difficult.

Fortunately, source-level debugging is rarely necessary for Termite drivers. Along with the implementation of the driver in C, Termite also outputs the state machine of the driver, similar to the one shown in Figure 6.16, which can be viewed as the implementation of the driver in a high-level language. By observing the execution of the driver at the state-machine level, one can easily spot situations where either the driver or its environment violates protocol specifications.

We have implemented such a state-machine-level debugger for Termite. In order to enable debugging, the driver must be synthesised with an option that forces the synthesis tool to inject debugger callouts in the driver code. At runtime, these callouts are activated by passing a special argument to the driver kernel module.

The debugger front-end runs on a separate machine and communicates with the debugged driver via a serial port. It provides a graphical user interface that visualises the

driver state machine and the current state of the driver. Similarly to conventional debuggers, the Termite debugger interacts with the driver in three situations. First, the user may single-step the driver, in which case the driver stops after each state transition. Second, the user may set breakpoints on one or more states of the driver state machine and let the driver run until it hits one of the breakpoints. Third, the driver stops and notifies the debugger whenever it receives an illegal method invocation from the environment.

In each case, the driver sends a message containing information about the last transition and values of driver state variables to the debugger and waits for a response from the debugger before continuing execution.

The debugger visualises the data received from the driver and allows the user to set additional breakpoints before issuing a step or run command. Figure 6.23 shows a screenshot of the Termite debugger stopped at a breakpoint. The debugger window contains a fragment of the driver state machine. The breakpoint state and the last transition taken by the driver before entering this state are highlighted. The status line at the top of the window shows identifiers of the current and the previous states, the name of the method that triggered the transition and the values of method arguments.

The current implementation of the debugger is missing some useful features that would further simplify error diagnostics. In particular, it does not relate the current state of the driver state machine to the corresponding states of the device and OS protocol state machines. Such functionality can be easily added to the debugger, since the mapping between driver states and protocol states is maintained during driver synthesis and can be stored along with the driver state machine.

6.8 Evaluation[4]

In this section I report on the experience in applying Termite to synthesise drivers for real, non-trivial devices, measure the performance of the synthesised drivers, and evaluate the reusability of Termite device specifications across different OSs.

6.8.1 Synthesising drivers for real devices

We have used Termite to synthesise two device drivers for Linux: a driver for the Ricoh r5c822 SD host controller (a full-featured analogue of the SD host controller described in Section 6.5) and a driver for the ASIX ax88772 100Mb/s USB-to-Ethernet adapter described in Chapter 5. These drivers occupy the middle range of the driver complexity spectrum. In particular, they support most features found in modern devices, including power management, request queueing, and DMA (with the ax88772 driver using DMA indirectly via the USB host controller). Unlike more complex devices, they use simple DMA descriptor formats and support a limited range of configuration options. Since the two devices belong to

[4]The evaluation presented in this section was completed in collaboration with Dr. Peter Chubb.

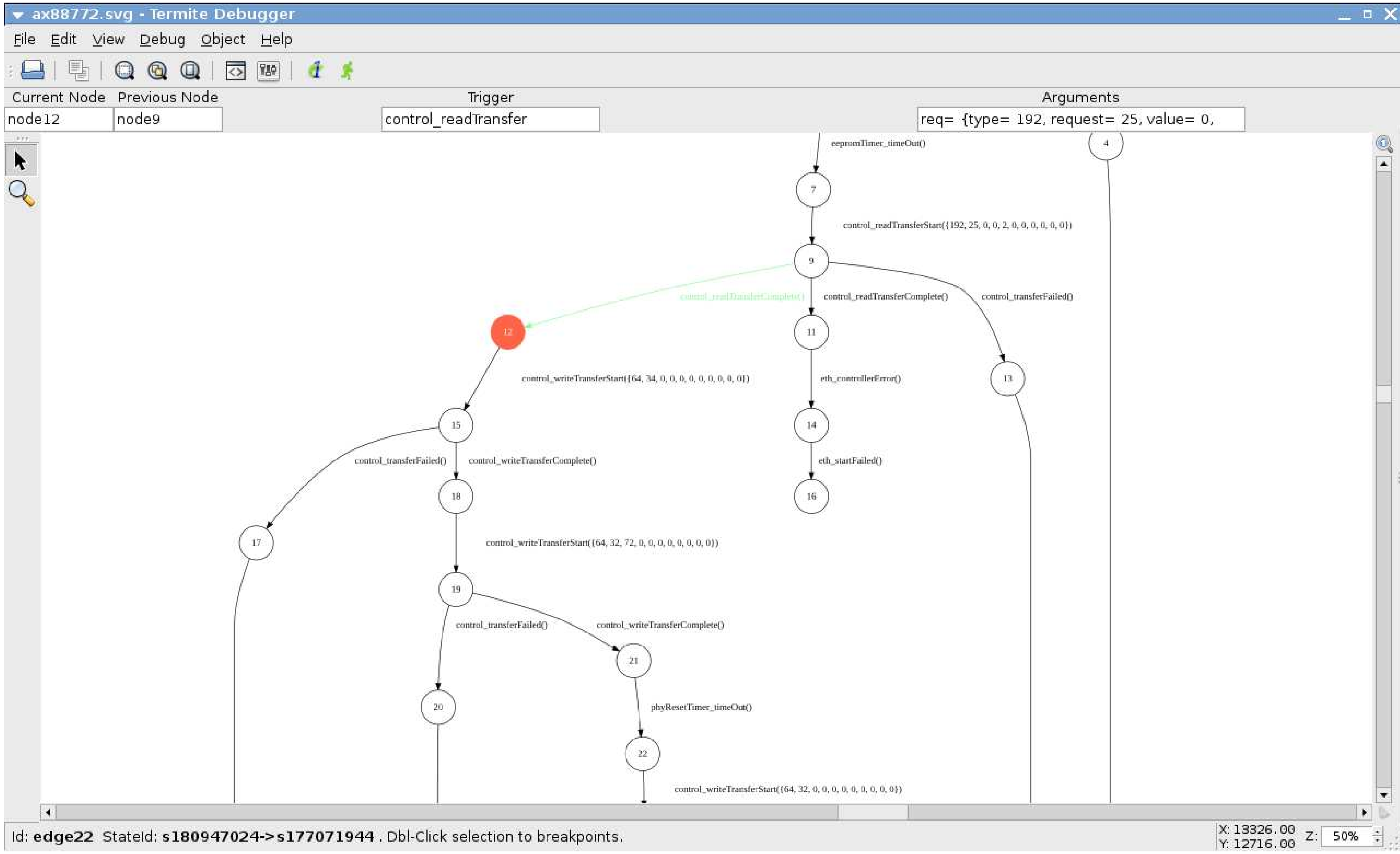


Figure 6.23: Termite debugger screenshot.

	SD	Ethernet
Native Linux driver	1174	1200
Device protocol	653	463
OS protocol (SD/Ethernet)	378	213
Bus protocol (PCI/USB)	263	96
Synthesised driver	4667	2620

Table 6.3: Size in lines of code, excluding comments, of the r5c822 and ax88772 driver implementations in Linux, their Termite specifications, and the synthesised drivers.

different device classes and attach to different buses (PCI and USB), these examples cover a broad spectrum of issues involved in driver synthesis.

Both devices are based on proprietary designs, so we did not have access to their RTL descriptions. The r5c822 controller implements a standardised SD host controller architecture whose detailed informal description is publicly available [SD 07]. This description provided sufficient information to derive a Termite model of the controller interface.

The ax88772 data sheet [ASI08] did not contain sufficient information to derive a Termite model of the device from it. In particular, it did not provide a complete description of device initialisation and configuration. For instance, when studying the Linux driver for this device we discovered a sequence of register reads and writes that the driver performed during startup. Many of these operations could not be explained based on the information in the data sheet. However, removing or even reordering any of them resulted in a misconfigured device. Therefore, we used the Linux driver for this device as the primary source of information.

As a result, the two specifications are substantially different in style. As explained in Section 6.2.2, specifications derived from device documentation tend to be declarative in nature: they describe how the device responds to various software commands, but do not enforce a particular ordering of these commands, which must be computed during driver synthesis based on the goals that the driver must achieve in different states. In contrast, specifications based on existing driver code are more constructive: they define sequences of commands and device reactions that must be issued to generate a specific device-class event (e.g., to complete and SD command or transfer a network packet).

Table 6.3 compares the size of Termite specifications to the manual implementation of equivalent drivers in the Linux kernel tree. Although line counts are not a reliable complexity measure, especially when comparing code written in different languages, note that for both drivers the device specification, which is the only part that needs to be developed per device, is significantly smaller than the native Linux driver. The last line of the table shows that the synthesised drivers are several times larger than the manual implementations. This is one area for future improvement.

In one case we were unable to completely specify the device in Termite: the ax88772

driver must implement pre- and post-processing of network buffers exchanged with the device in order to append and remove an extra header that the device expects in each packet. Termite currently does not provide facilities to specify constraints on the content of memory buffers. DMA buffers are currently represented using their virtual and physical addresses and size, which allows passing unmodified buffers between the device and the OS, but not performing any transformations on them. We therefore implemented this functionality in C and made it available to the device protocol via two methods: `rxFixup` and `txFixup`. The total size of these functions is 110 lines of C code, or less than 10% of the size of the native Linux implementation of this driver.

Synthesis took 2 minutes for the ax88772 driver and 2 hours 26 minutes for the r5c822 driver on a 2GHz AMD Opteron machine with 8GB of RAM. This difference is due to the different styles of the two device specifications. The ax88772 specification, derived from an existing driver, only contains useful execution paths that lead to the occurrence of device-class events. In contrast, the more declarative r5c822 specification allows a large number of possible command sequences, which the synthesis algorithm must explore to find the meaningful ones that lead to the goal.

6.8.2 Performance

We compared the performance of the synthesised drivers against that of equivalent native Linux drivers. Benchmarks described in this section were run on a 2GHz Centrino Duo machine. We disabled one of the cores in order to allow precise CPU accounting. For the SD bus controller driver we ran a locally developed benchmark that performs a sequence of raw (unbuffered) reads from an SD card connected to the controller. We measured CPU usage and achieved bandwidth for different block sizes. In all cases, the throughput and CPU usage of the synthesised driver differed from that of the native Linux driver by less than 1%.

The USB-to-Ethernet controller is more interesting from the performance perspective, as it is capable of generating high CPU loads, especially when handling small transfers. Figure 6.24 compares throughput and CPU utilisation achieved by the synthesised and native drivers under the Netperf [Net] TCP_STREAM benchmark. In these experiments, the Netperf server was run with default arguments on the machine with the synthesised driver under test. The Netperf client was running on another machine with the following arguments:

```
netperf -H<server-ip-addr> -p <server-port> -t TCP_STREAM -c
-C -l 60 - -m <transfer-size>
```

Where the `-l 60` argument sets the time of the run to 60 seconds. The `-c` and `-C` arguments enable CPU utilisation calculations on both the server and the client. The `-m` option sets the transfer size.

According to Figure 6.24, both drivers showed virtually identical performance even

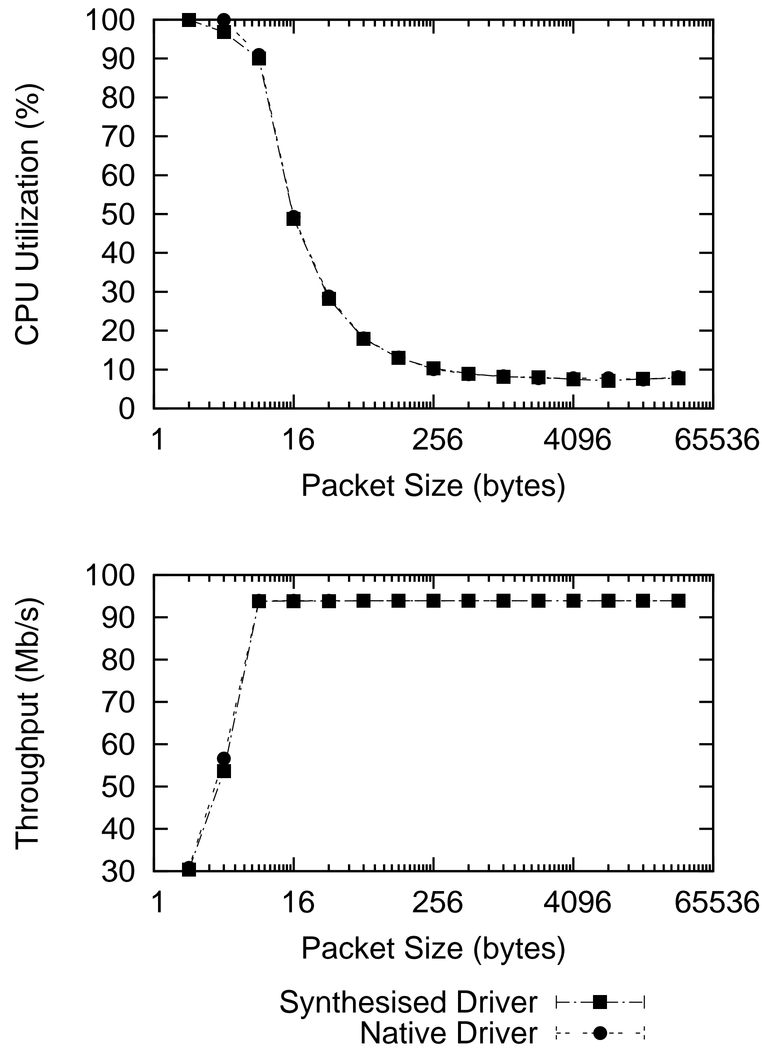


Figure 6.24: ax88772 TCP throughput benchmark results. The top graph shows CPU utilisation; the bottom graph shows achieved throughput.

under the heaviest loads induced by a large number of small packets.

These results are reassuring, as they indicate that automatically synthesized drivers can achieve performance comparable to manually developed ones.

6.8.3 Reusing device specifications[5]

In order to validate the claim that device specifications can be reused across different OSs, we synthesised a FreeBSD r5c822 driver from the same device specification that was used to generate the Linux version of the driver. To this end we developed specifications for the FreeBSD versions of the SD host controller driver protocol and the PCI bus transport protocol. These protocols differ from their Linux counterparts in a number of aspects, including SD command format, driver initialisation, PCI resource allocation, bus power

[5]The work described in this section was completed in collaboration with Etienne Le Sueur.

management, and DMA descriptor allocation. Once these protocols were specified (this took approximately 6 person-hours, an effort that only needs to be undertaken once for the given OS), a driver for FreeBSD was generated automatically using the unmodified device and device-class specifications.

6.9 Limitations and future work

Our experience with Termite has demonstrated the feasibility of driver synthesis for real devices. This section summarises limitations of the current implementation and improvements required to turn it into a practical solution capable of generating a broad class of drivers.

The front-end Termite currently relies on the device manufacturer or the driver developer to write a formal specification of the device protocol. While offering substantial advantages over conventional driver development in terms of code structure, size, reuse, and quality assurance, this approach still requires substantial manual effort. This effort can be avoided by automatically distilling a Termite model of the device from its RTL description. Implementing support for this is the key area of the continuing research on driver synthesis.

The synthesis algorithm Several limitations of the current synthesis algorithm complicate modelling. These include the assumption of deterministic device behaviour described in Section 6.5.4, and the lack of support for complex constraints on variables in the symbolic execution engine (Section 6.6.4).

In addition, Termite currently only allows the manipulation of memory buffers via calls to C functions (Section 6.8.1). In order to reduce the reliance on manually written code, support for the specification of constraints on the memory buffer layout (fragmentation, alignment, etc.) and content (headers, paddings, etc.) needs to be added to Termite. This way, one will only have to use C to implement more complex data transformations, such as hashing or encryption.

Termite does not support drivers that require dynamic resource allocation. In some cases, resource allocation is performed by the Dingo runtime framework. For example, when a USB device driver sends a request to the device, the framework allocates a new USB request structure. Most of the remaining allocation operations performed by drivers are related to the management of DMA buffers. Support for these operations must be added as part of the buffer management extension described in the previous paragraph.

Finally, the Termite synthesis algorithm needs further improvement to reduce the synthesis time and support more complex devices with larger state spaces. One promising approach is to use counterexample-guided abstraction refinement, which allows a reduction of the size of the state space to be explored by dynamically identifying relevant state information. This technique has been successfully applied in model checking and has also been shown to work for two-player games [HJM03].

The overall approach The Termite approach to driver synthesis relies on the distinctive state-machine-like structure of device drivers and its relation to the structure of the I/O device. Some new devices, such as high-end GPUs and network processors do not fit into this model. These devices are built around a general-purpose CPU, often running a separate instance of a general-purpose OS. They are controlled by uploading programs that execute on the device's CPU and communicate with the host CPU via the I/O bus. Modelling such devices and generating software for them is beyond the reach of Termite.

6.10 Conclusions

Device driver synthesis is a promising approach to solving the driver reliability problem. In this chapter I have demonstrated the feasibility of this approach by describing a driver synthesis methodology and its implementation. The ultimate goal of this work is to create a viable alternative to current manual driver development practices, leading to better quality drivers. The key factor in achieving this is to make driver synthesis attractive to device vendors by providing easy-to-use and efficient languages and tools for it.

Chapter 7

Conclusions

Software defects in device drivers are the leading source of instability in current OSs. This dissertation has demonstrated that many driver defects can be avoided with the help of an improved device driver architecture and development process. This improvement is achieved by taking an approach based on identifying and eliminating the root causes of the driver reliability problems, as opposed to focusing on their symptoms.

In particular, I showed that many driver defects are provoked by the complex and poorly defined interfaces between the driver and the OS. This problem is addressed by abandoning the conventional multithreaded model of computation enforced by most OSs on device drivers in favour of a more disciplined event-based model and by documenting OS protocols using a formal visual language. In addition to helping driver developers avoid defects in drivers, this approach enables automatic verification of driver behaviour against protocol specifications. In particular, I demonstrated an implementation that performs such verification at runtime using automatically generated protocol observers. Static verification of protocol compliance is part of the future work.

The formal approach to modelling the device driver behaviour leads to a radically new method of driver construction, which consists of automatically generating the implementation of the driver based on a formal model of its device and OS protocols. This approach has the potential to dramatically reduce driver development effort while increasing driver quality.

In this dissertation I have demonstrated the feasibility of automatic driver synthesis for real non-trivial devices. Further research is necessary in order to turn the results of this work into a practical driver synthesis tool. Open problems include automatically deriving device protocol specifications from the RTL description of the device, specifying and synthesising behaviours that involve memory buffer manipulation, and improving the synthesis algorithm to deal with more complex devices.

Appendix A

The syntax of the Tingu and Termite protocol specification languages

This appendix describes the complete syntax of the Tingu and Termite protocol specification languages. The two languages use common syntax for declaring components, protocols, types, methods, variables, and dependencies. The main difference is that Tingu specifies behavioural constraints of the protocol using Statecharts, while Termite uses a textual formalism based on the LOTOS process calculus.

Section A.1 provides a *Backus-Naur Form (BNF)* specification of the common part of the Tingu and Termite syntax. Section A.2 describes the syntax of state transition labels used in both Tingu and Termite specifications. Section A.3 describes the subset of the Statecharts visual syntax used in Tingu protocol state machines specifications. Finally, Section A.4 describes the Termite process syntax.

A.1 Component, protocol, and type declarations

A tingu specification consists of zero or more specification items, where every item is a type, protocol, or component declaration.

```
<tingu-spec> ::=
               | <tingu-spec> <spec-item>
<spec-item> ::= <type-decl>
              | <protocol-decl>
              | <component-decl>
```

A.1.1 Common definitions

A Tingu identifier is a string of alphanumeric characters and underscore, beginning with a letter or an underscore.

```
<identifier> ::= [a-zA-Z_][a-zA-Z0-9_]*
```

A quoted string:

```
<quoted-string> ::= \"([^\"\t\n]*)\"
```

Identifiers are used as component, protocol, method, argument, type, port names, etc.

```
<component-name> ::= <identifier>
<protocol-name> ::= <identifier>
<port-name> ::= <identifier>
<type-name> ::= <identifier>
<method-name> ::= <identifier>
<arg-name> ::= <identifier>
<var-name> ::= <identifier>
<enumerator-name> ::= <identifier>
<struct-field-name> ::= <identifier>
<function-name> ::= <identifier>
<process-name> ::= <identifier>
```

A type specifier is the name of an existing type, a pointer to an existing type, or a fixed-width integer type specifier. Type specifiers are used in method argument declarations, variable declarations, and type declarations.

```
<type-specifier> ::= <type-name>
                   | <type-name> "*"
                   | <int-type-specifier>
```

A fixed-width integer type specifier consists of the “int” or “unsigned” keyword followed by the type width (in bits) in angle brackets.

```
<int-type-specifier> ::= "int" "<" <int-constant> ">"
                       | "unsigned" "<" <int-constant> ">"
```

Port substitution lists are used in component port declarations and protocol subport declarations.

```
<port-substitution-list> ::= "<" <port-subst>
                             <port-subst-list-tail> ">"
<port-subst-list-tail> ::=
                         | <port-subst-list-tail> "," <port-subst>
```

A port substitution consists of the path to the substituting port and the substituted port name, separated by a “/”. A path is a sequence of “.”-separated port names or the “self” keyword.

```
<port-subst> ::= <port-path> "/" <port-name>
<port-path> ::= <port-name> <port-path-tail>
              | "self"
<port-path-tail> ::=
                   | <port-path-tail> "." <port-name>
```

A.1.2 Types

A type declaration is an enumeration declaration, a structure declaration, and integer type declaration, an opaque type declaration, or a pointer type declaration.

```
<type-decl> ::= <enum-decl>
              | <struct-decl>
              | <int-decl>
              | <opaque-decl>
              | <ptr-decl>
```

Enumeration declarations follow the C syntax.

```
<enum-decl> ::= "enum" <type-name> "{" <enum-list> "}"
<enum-list> ::=
              | <enumerator> <enum-list-tail>
<enum-list-tail> ::=
                   | <enum-list-tail> "," <enumerator>
<enumerator> ::= <enumerator-name> "=" <int-constant>
```

Structure declarations follow the C syntax.

```
<struct-decl> ::= "struct" <type-name> "{"<struct-field-list>"}"
<struct-field-list> ::=
                      | <struct-field-spec> <struct-field-list>
<struct-field-spec> ::= <type-specifier> <struct-field-name> ";"
```

An integer type declaration consists of an integer type specifier (Section A.1.1) followed by the type name

```
<int-decl> : <int-type-specifier> <type-name>
```

An opaque type declaration consists of the “opaque” keyword followed by the type name.

```
<opaque-decl> ::= "opaque" <type-name>
                | "opaque" "struct" <type-name>
                | "opaque" "union" <type-name>
                | "opaque" "enum" <type-name>
```

A pointer type declaration consists of the base type name, a “*”, and the pointer type name.

```
<ptr-decl> ::= <type-name> "*" <type-name>
```

A.1.3 Protocols

A protocol declaration consists of a “protocol” keyword, protocol name and a list of protocol sections in curly braces.

```
<protocol-decl> ::= "protocol" <protocol-name>
                    "{" <protocol-sections> "}" ";"
<protocol-sections> ::=
                      | <protocol-sections> <protocol-section>
```

A protocol section is a types section, a methods section, a dependencies section, a variables section, a ports section, or a transitions section.

```
<protocol-section> ::= <type-section>
                     | <method-section>
                     | <dependency-section>
                     | <variable-section>
                     | <port-section>
                     | <transition-section>
```

A types section consists of a "types:" keyword followed by type declarations (Section A.1.2).

```
<type-section> ::= "types" ":" <type-decls>
<type-decls> ::=
               | <type-decls> <type-decl> ";"
```

A methods section consists of a "methods:" keyword followed by method declarations.

```
<method-section> ::= "methods" ":" <method-decls>
<method-decls> ::=
                 | <method-decls> <method-decl> ";"
```

A method declaration consists of a direction specifier ("input", "output", or "internal"), method name, argument list, an optional method attribute, and an optional spawn clause, which specifies sub-ports that are spawned by this method (the "internal" keyword and method attributes are only used in the Termite version of the protocol specification language).

```
<method-decl> ::= "in" <method-signature>
                | "out" <method-signature>
                | "internal" <method-signature>
<method-signature> ::= <method-name> "(" <argument-list> ')'
                       <optional-method-attr> <optional-spawn>

<optional-method-attr> ::=
                         | "timed"
                         | "fallback"

<optional-spawn> ::=
                   | "spawns" <spawn-list>
<spawn-list> ::= <port-name>
```

```
                | <spawn-list> "," <port-name>

<argument-list> ::=
                  | <argument-decl> <argument-list-tail>
<argument-list-tail> ::=
                       | <argument-list-tail> "," <argument-decl>
<argument-decl> ::= <optional-arg-attr> <type-specifier>
                    <arg-name>
```

A method argument can have an optional “out” qualifier, meaning that the argument value must be passed by reference and is modified by the method.

```
<optional-arg-attr> ::=
                      | "out"
```

A dependencies section consists of the “dependencies:” keyword followed followed by dependency declarations.

```
<dependency-section> ::= "dependencies" ":" <dependency-decls>
<dependency-decls> ::=
                     | <dependency-decls> <dependency-decl> ";"
```

A dependency consists of a protocol name, a port identifier and a list of method dependencies in curly braces. A method dependency consists of the “restricts” or “listens” keyword followed by the method name followed by the optional “timed” or “fallback” attribute (these attributes are used in Termite specifications only).

```
<dependency-decl> ::= <protocol-name> <port-name>
                      <dependency-body>
<dependency-body> ::= "{" <method-dependency-list> "}"
<method-dependency-list> ::=
                           | <method-dependency-list>
                             <method-dependency> ';'
<method-dependency> ::= "restricts" <method-name>
                        <optional-method-attr>
                      | "listens" <method-name>
                        <optional-method-attr>
```

A variables section consists of the "variables:" keyword followed by variable declarations.

```
<variable-section> ::= "variables" ":" <variable-decls>
<variable-decls> ::=
                   | <variable-decls> <variable-decl> ";"
```

A variable declaration consists of a type specifier or an ADT specifier and variable name.

```
<variable-decl> ::= <type-specifier> <var-name>
                  | <adt-specifier> <var-name>
```

An ADT specifier consists of a container name (currently, “list” and “set” are the only supported containers) followed by a type specifier in angle brackets.

```
<adt-specifier> ::= <adt-name> "<" <type-specifier> ">"
<adt-name> ::= "list"
<adt-name> ::= "set"
```

A port section consists of the “ports:” keyword followed by a list of port declarations.

```
<port-section> ::= "ports" ":" <port-decls>
<port-decls> ::=
               | <port-decls> <port-decl>
```

A port declaration consists of a protocol name, a port name, an optional index type specifier in square brackets and an optional list of port substitutions (Section A.1.1).

```
<port-decl> ::= <protocol-name> <port-name>
                "[" <optional-type-specifier> "]" ";"
              | <protocol-name> <port-name>
                "[" <optional-type-specifier> "]"
                <port-substitution-list> ";"
<optional-type-specifier> ::=
                            | <type-specifier>
```

A transitions section consists of the "transitions:" keyword followed by a state machine specification. The state machine specification can be either an import statement, containing a reference to an external specification or a program in the Termite specification language (see Section A.4).

```
<transition-section> ::= "transitions" ':' <import-statement>
                       | "transitions" ':' <termite-state-machine>
<import-statement> :: "import" "(" "format" "=" <identifier> ","
                      "location" "=" <quoted-string> ")" ";"
```

A.1.4 Components

A component declaration consists of the `component` keyword followed by the component name and a port list in curly braces.

```
<component-decl> ::= "component" <component-name>
                     "{" "ports" ":" <component-ports> "}" ";"
<component-ports> ::=
                    | <component-ports> <component-port>
```

A component port declaration consists of a protocol name, a port name, and an optional port substitution list.

```
<component-port> ::= <protocol-name> <port-name> ';'
                   | <protocol-name> <port-name>
                     <port-substitution-list> ';'
```

A.2 Protocol state transition labels

This section describes the syntax of state transition labels used in both Tingu and Termite specifications.

A transition label consists of a trigger, a guard, an action, and an optional "timed" keyword, which is only used in Termite specifications.

```
<transition> ::= <trigger> <guard> <action> <optional-timed>
optional-timed ::=
                 | ":" "timed"
```

A trigger consists of a method identifier with an optional direction qualifier. The method identifier is either a local method name that refers to a method of the current protocol or a dependency method name.

```
<trigger> ::= "!" <method>
            | "?" <method>
            | <method>
<method> ::= <method-name>
           | <port-name> "." <method-name>
```

A guard can be empty (transition is always allowed) or consist of an expression in square brackets.

```
<guard> ::=
          | "[" <expression> "]"
```

An action can be empty (transition does not modify protocol variable) or consists of a "/" followed by a statement.

```
<action> ::=
           | "/" <statement>
```

The simplest expression consists of a single operand, which can be a variable name, a method argument name, an enum, integer, or boolean constant.

```
<expression> ::= <operand>
<operand> ::= <variable-name>
            | "$" <arg-name>
            | <enumerator-name>
            | <int-constant>
            | "true"
            | "false"
```

Unary, binary, and ternary arithmetic expressions:

```
<expression> ::= "~" <expression>
               | "!" <expression>
               | "-" <expression>
               | <expression> "||" <expression>
               | <expression> "&&" <expression>
               | <expression> "==" <expression>
               | <expression> "!=" <expression>
               | <expression> "+" <expression>
               | <expression> "-" <expression>
               | <expression> "*" <expression>
               | <expression> "/" <expression>
               | <expression> "%" <expression>
               | <expression> "<<" <expression>
               | <expression> ">>" <expression>
               | <expression> "|" <expression>
               | <expression> "&" <expression>
               | <expression> "^" <expression>
               | <expression> ">" <expression>
               | <expression> ">=" <expression>
               | <expression> "<" <expression>
               | <expression> "<=" <expression>
               | <expression> "?" <expression> ":" <expression>
```

Parenthesis are used to control the ordering of subexpression evaluation.

```
<expression> ::= "(" <expression> ")"
```

Structure field access expression:

```
<expression> ::= <expression> "." <struct-field-name>
```

ADT function invocation expression.

```
<expression> ::= <expression>"."<function-name> "("<arguments>")"
<arguments> ::=
              | <argument-list>
<argument-list> ::= <argument>
                  | <argument-list> "," <argument>
<argument> ::= <expression>
```

A statement can consist of a single simple statement or a semicolon-separated list of statements.

```
<statement> ::= <simple-statement>
<statement> ::= "{" <statement> "}"
<statement> ::= <statement-list>
<statement-list> ::= <statement> ";"
```

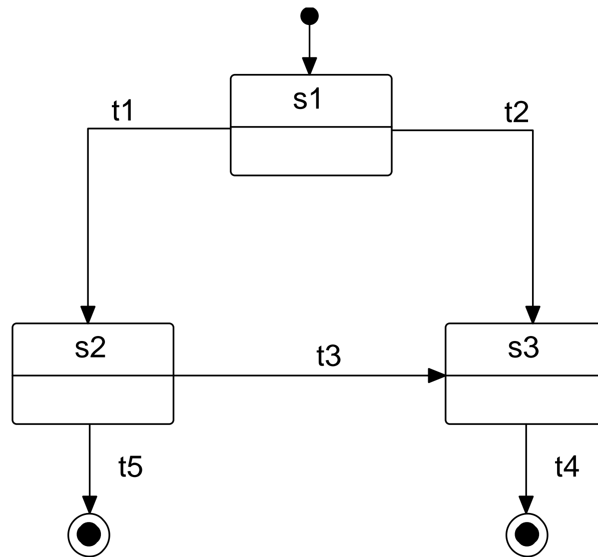


Figure A.1: Example of a simple statechart.

```
<statement-list> ::= <statement-list> <statement> ";"
```

A simple statement is a single expression, an assignment statement, and increment or decrement statement, or a port spawn statement.

```
<simple-statement> ::= <expression>
                     | <expression> "=" <expression>
                     | "++" <expression>
                     | "--" <expression>
                     | "new" <port-name> "(" <expression> ")"
```

A.3 Protocol state machines

The complete Statecharts language defined by Harel [Har87] proved difficult to assign unambiguous formal semantics [vdB94]. Fortunately, most of the problematic Statechart features are either irrelevant or non-essential to modelling driver protocols. The subset of the Statecharts syntax described here can be assigned formal semantics in a simple and natural way.

Statecharts are state machine diagrams extended with features to model hierarchy and concurrency. A basic statechart that contains neither hierarchy nor concurrency is just a finite state machine consisting of a set of states and transitions, with exactly one default initial state and any number (including zero) of final states (Figure A.1). Execution of a statechart is triggered by events. In Tingu, events correspond to driver interface method invocations.

Compact representation of complex behaviours is achieved by organising states into a hierarchy: several simple states can be placed inside a superstate, which can, in turn, belong

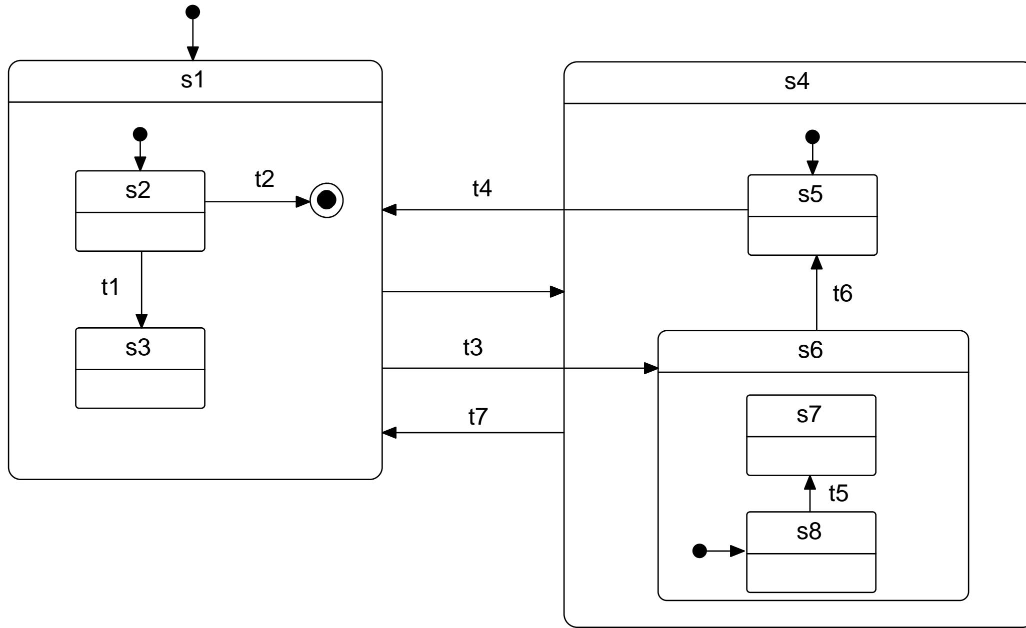


Figure A.2: Example of a statechart with OR-superstates.

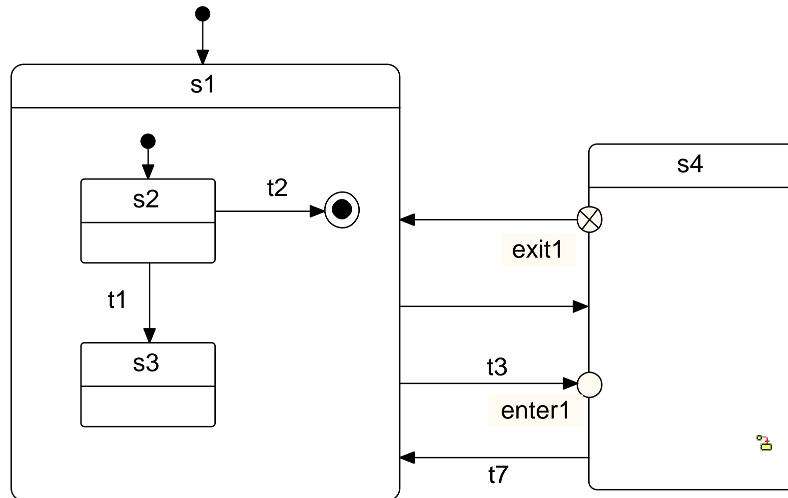
to a higher-level superstate, etc. Two types of superstates are supported: OR-superstates and AND-superstates.

OR-superstates are used to cluster states with similar behaviours. Two or more states that have identical outgoing transitions (i.e., transitions with the same label and target states) can be placed inside an OR-superstate and the identical transitions can be replaced with a single transition that originates from the superstate. For example, transitions `t3` and `t7` in Figure A.2 define common behaviours for substates of OR-superstates `s1` and `s4`, respectively. The semantics of an OR-superstate is the exclusive or of its substates: when the statechart is in an OR-superstate, it must be in exactly one of its substates.

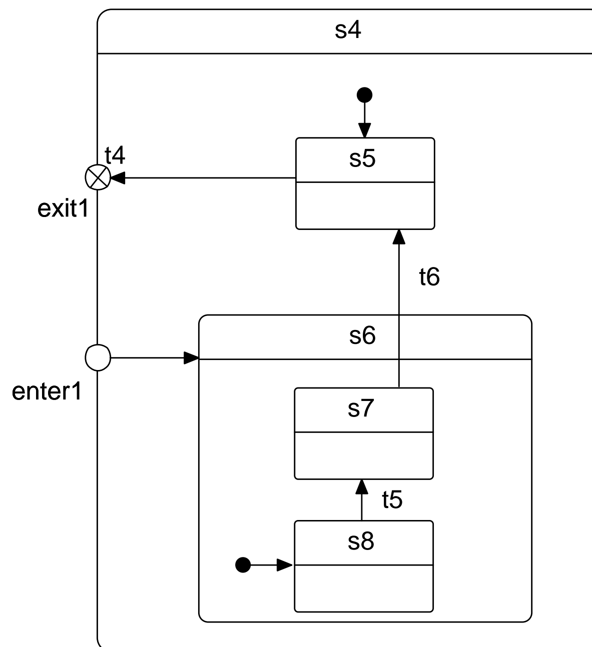
An OR-superstate can have a default state (e.g., state `s2` in Figure A.2). A state transition that terminates at the superstate boundary enters the superstate via its default state. An OR-superstate can also contain final states. When a final state is reached, the superstate is exited through the default exit transition, which is an unlabelled transition originating at the superstate boundary. There can be at most one default exit transition from a superstate, and a superstate that has at least one final state is required to have a default exit transition. Multilevel state transitions that cut through several levels of the state hierarchy are allowed (e.g., `t3` and `t4` in Figure A.2).

If the statechart is too large to fit in a single diagram, a superstate can be collapsed into a simple state and its content can be moved to a separate diagram. This is illustrated in Figure A.3, which splits the statechart in Figure A.2 into two statecharts. Enter and exit connectors are introduced in the points where multilevel state transitions cross the boundary of the collapsed superstate.

An OR-superstate can contain one or more history pseudo-states. When the superstate is entered through a history pseudo-state, the last configuration of the superstate and all its



(a) The main statechart.



(b) Superstate `s4` expanded.

Figure A.3: Example of state collapsing.

substates is restored recursively. If this is the first time the superstate has been activated, the default state is entered. Figure A.4 shows a modified version of the previous example involving a history pseudostate.

AND-superstates represent concurrent activities. An AND-superstate consists of two or more regions separated with dashed lines (Figure A.5). All regions of an AND-superstate are active simultaneously. An individual region behaves like an OR-superstate. A single event can trigger state transitions in one or more regions inside an AND-superstate, in which case all state transitions occur simultaneously. Any transition that leaves an AND-superstate preempts all its internal regions and exits the superstate. For example, both transitions `t4` and `t5` in Figure A.5 exit superstate `s1`.

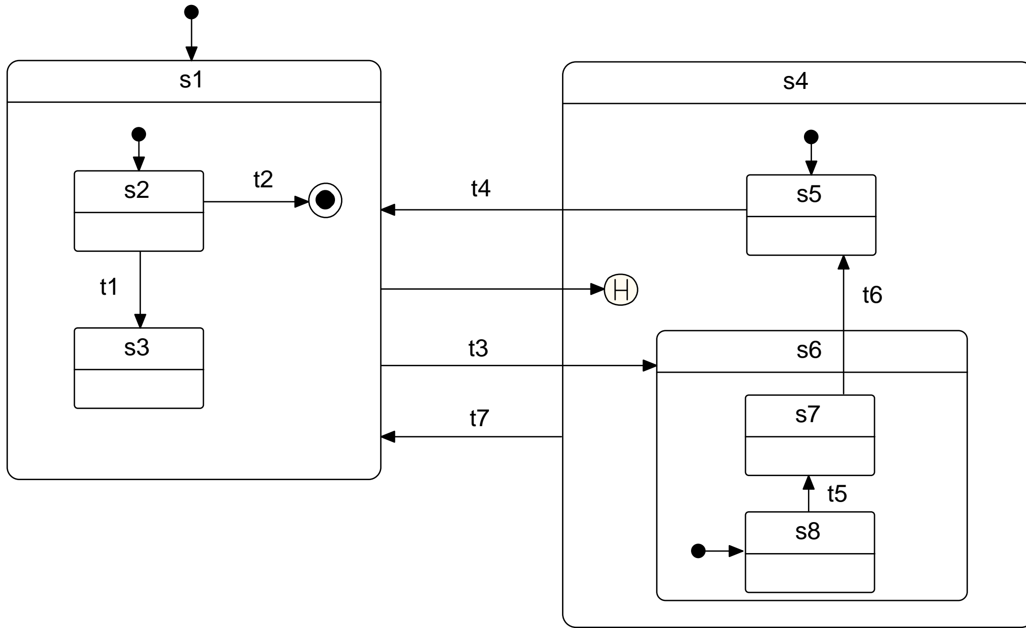


Figure A.4: Example of a statechart with a history pseudo-state.

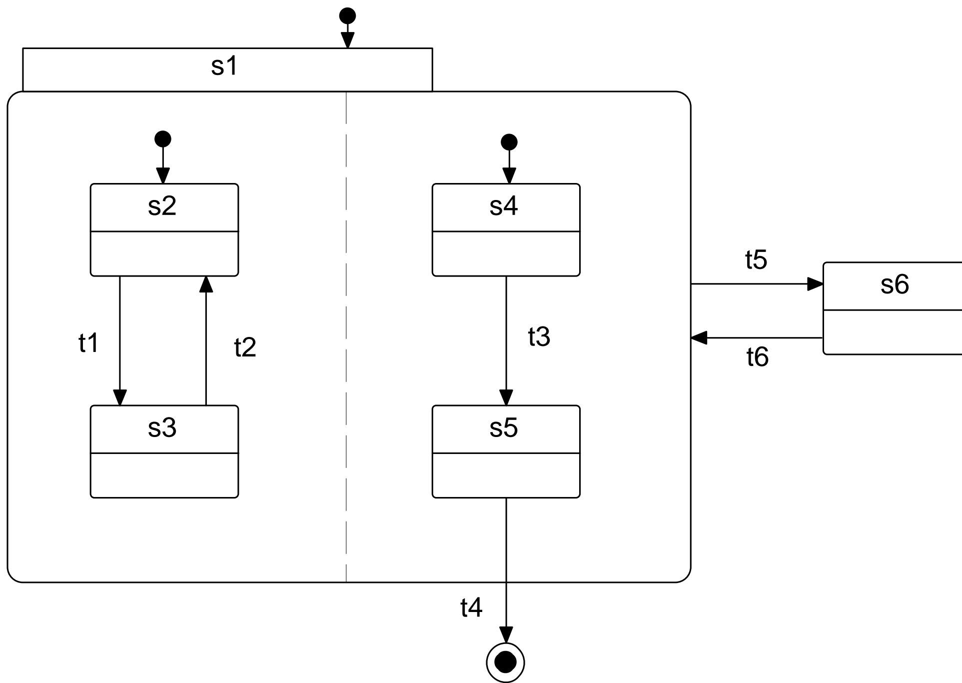


Figure A.5: Example of a statechart with an AND-superstate.

A single event can enable several conflicting transitions, i.e., transitions that lead to different states. In such a case, the transition with the highest scope is taken (the scope of a transition is the lowest ancestor of both the source and the target state of the transition). A well-formed Tingu state machine is not allowed to contain conflicting transitions with the same scope.

The following Statecharts features are not supported in Tingu: internal events, instantaneous states, join and fork connectors, and conditional pseudo-states.

A.4 Termite processes

A Termite specification of a protocol state machine consists of a Termite behavioural expression followed by an optional `"where"`-clause, which declares named processes referenced by the behavioural expression.

```
<termite-state-machine> ::= <behavioural-expression>
                          | <behavioural-expression>
                            "where"
                            <process-decls>
<process-decls> ::=
                  | <process-decls> <process-decl>
```

A process declaration consists of the `"process"` keyword followed by the name of the process and the specification of its behaviour.

```
<process-decl> ::= "process" <process-name>
                   <behavioural-expression>
                   "endproc"
```

A behavioural expression is one of the expression types described in Section 6.4. A behavioural expression can be taken in parenthesis to control the ordering of operators.

```
<behvaoural-expression> ::= "stop"
                          | "exit"
                          | <process-name>
                          | <prefixing>
                          | <choice>
                          | <conditional>
                          | <sequential>
                          | <preemption>
                          | <parallel>
                          | <interleaving>
                          | (<behavioural-expression>)
<prefixing> ::= <transition> ";" <behavioural-expression>
<choice> ::= <behavioural-expression> "[]"
             <behavioural-expression>
<conditional> ::= "if" "[" <expression> "]"
                  <behavioural-expression>
                  "else" <behavioural-expression>
<sequential> ::= <behavioural-expression>
                 ">>"
                 <behavioural-expression>
<preemption> ::= <behavioural-expression>
                 "[>"
                 <behavioural-expression>
<parallel> ::= <behavioural-expression>
```

```
                "|[" <synchronisation-list>  "]|"
                 <behavioural-expression>
<synchronisation-list> ::= <method-name> <sync-list-tail>
<sync-list-tail> ::=
                    | "," <synchronisation-list>
<interleaving> ::= <behavioural-expression> "|||"
                    <behavioural-expression>
```

Appendix B

Tingu protocol specification examples

This appendix illustrates the use of the Tingu language using several examples of Tingu protocol specifications.

B.1 The `Lifecycle` protocol

The `Lifecycle` protocol defines initialisation and shutdown requests that must be implemented by all Dingo drivers. The protocol specification is shown in Figures B.1 and B.2.

B.1.1 `Lifecycle` methods

`probe` - Request to probe and initialise the device. This request is delivered to the driver when a new device is discovered on the bus during system startup or at runtime. Upon receiving this message, the driver can start issuing bus transactions to access the (as defined by bus-specific protocols).

`probeComplete` - Signals successful completion of device initialisation.

`probeFailed` - Unsupported device ot device initialisation failed.

`stop` - Stop the device and resease all resources held by the driver.

`stopComplete` - Driver deinitialisation complete. After receiving this notification, the OS may release bus resources associated with the driver; therefore no device accesses are allowed after `stopComplete` (as defined by bus-specific protocols).

`unplugged` - The OS notifies the driver that the device has been disconnected from the bus. The OS guarantees that no further I/O requests will be delivered to the device after an `unplugged` event. The driver is not allowed to issue new bus transactions, but may have to wait for outstanding transactions to terminate, as defined by the appropriate bus protocol.

```
protocol Lifecycle
{
methods:
  in  probe();
  out probeComplete();
  out probeFailed(error_t error);

  in  stop();
  out stopComplete();

  in  unplugged();

transitions:
  import(format=rhapsody, location="LifecycleSM@ioprotocols.sbs");
}
```

Figure B.1: The `Lifecycle` protocol declaration.

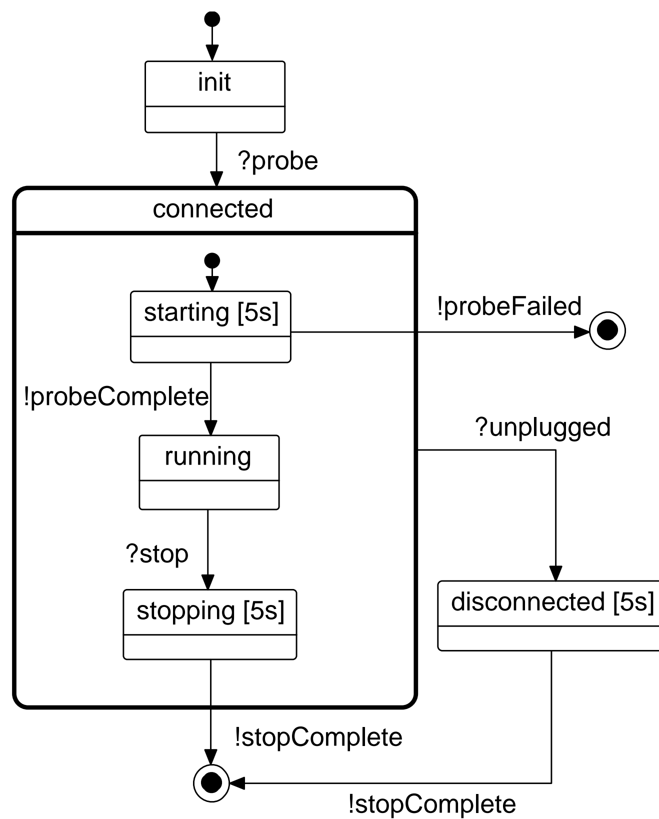


Figure B.2: The `Lifecycle` protocol state machine.

B.2 The `PowerManagement` protocol

The `PowerManagement` protocol defines device suspend and resume requests that must be implemented by all Linux drivers that support bus-directed power management. Specific devices can implement their own power saving schemes that are not covered by this protocol. The protocol specification is shown in Figures B.3 and B.4.

B.2.1 `PowerManagement` methods

`suspend` - The OS is about to put the entire bus that the device is connected to or just a subset of devices on the bus, including the current device, in a low-power mode. The driver must prepare the device for the transfer into the new state, which includes completing any outstanding requests and saving device context information that will be lost in the low-power mode. The argument of the request is the target power state, which must correspond to a lower state than the current one (D0 corresponds to running at full power, D3 corresponds to switching the device power off). The OS guarantees that no further I/O requests will be delivered to the device after a `suspend` request and until the `resumeComplete` notification from the driver (see below), as defined by the appropriate driver protocol, e.g., `EthernetController` or `Infiniband`.

`suspendComplete` - The device is ready to be suspended. The OS may switch the bus to the new power state after the driver calls this method, making certain bus operations unavailable, as defined by the bus protocol.

`resume` - The bus is running at full power again. The driver must restore the device state and prepare for handling OS requests.

`resumeComplete` - Resume complete; the driver is ready to handle new I/O requests from the OS.

```
protocol PowerManagement
{
  types:
    enum power_level_t {
      D0 = 0,
      D1 = 1,
      D2 = 2,
      D3 = 3
    };

  methods:
    in  suspend (power_level_t level);
    out suspendComplete ();

    in  resume ();
    out resumeComplete ();

  variables:
    /* Current device power mode */
    power_level_t power_level;

  dependencies:
    Lifecycle lc {
      listens   probeComplete;
      listens   probeFailed;
      listens   unplugged;
      restricts stop;
  };

  transitions:
    import(format=rhapsody,
           location="PowerManagementSM@ioprotocols.sbs");
};
```

Figure B.3: The `PowerManagement` protocol declaration.

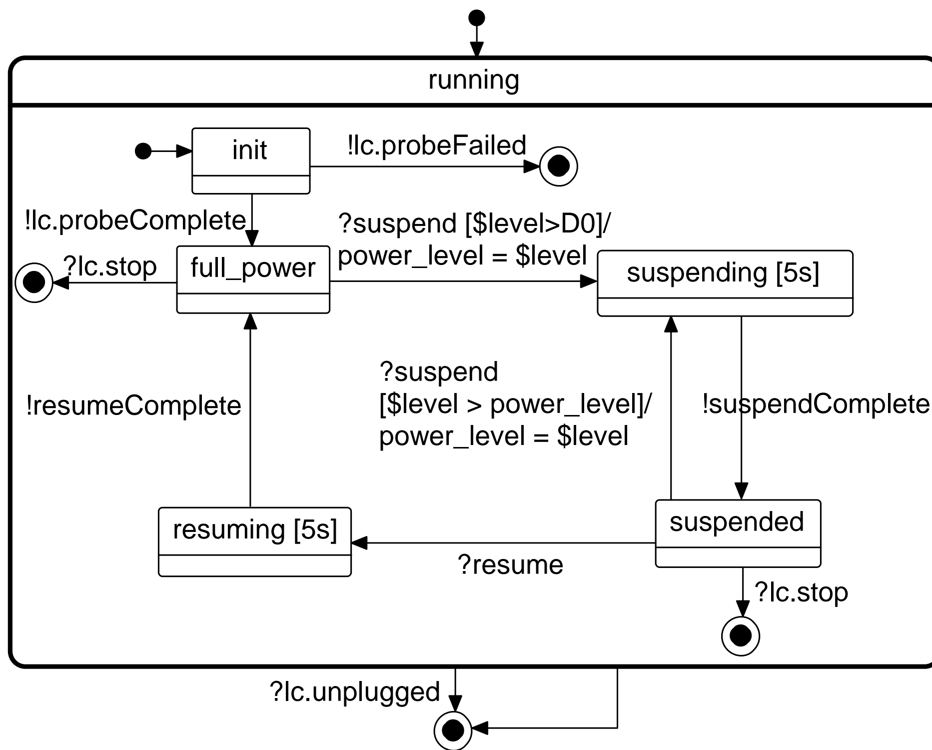


Figure B.4: The `PowerManagement` protocol state machine.

B.3 The `EthernetController` protocol

The `EthernetController` protocol describes the service that an Ethernet controller driver must provide to the OS. The service allows the OS to send and receive packets via the network interface implemented by the controller. Drivers that manage multiple network interfaces must implement an instance of this protocol for every supported interface. The protocol specification is shown in Figures B.5, B.6, B.7, B.8, and B.9.

B.3.1 `EthernetController` methods

`enable` - Request to enable receive and transmit circuits inside the controller and allocate any resources required to handle incoming and outgoing network packets. The OS invokes this method the first time a user process opens a network connection via the given interface.

`enableComplete` - The controller is enabled. Following this notification, the driver may start delivering incoming packets to the OS and the OS may start sending packets via the driver.

`disable` - Disable the device transmit and receive circuits. In response to this request, the driver must abort all outstanding packet transfers and disable device receive and transmit circuits. No new packets can be sent and received after `disable`

`disableComplete` - Disable complete.

`txStartQueue` - Notifies the OS that more space is available in the device transmit packet queue, so that the OS can send more packets to the drivers.

`txStopQueue` - Notifies the OS that the device transmit queue is full. The OS will not attempt to send new packets until a `txStartQueue` notification.

`txTimeout` - Called by the OS networking code when it detects a transmission timeout, to hint the driver that the device hardware may have locked up and needs a reset.

`txPacket` - Request to transmit a packet.

`txPacketDone` - Packet transfer complete.

`txPacketAbort` - Packet transfer failed.

`rxPacketInput` - Delivers a packet received from the network to the OS.

`linkUp` - Network link is up.

`linkDown` - Notifies the OS about a lack of carrier signal on the wire.

`setMacAddress` - Request to change the controller MAC address.

```
protocol EthernetController
{
  types:
    opaque sk_buff; // Linux structure describing a network packet
  methods: in  enable ();
    in  enable ();
    out enableComplete ();
    in  disable ();
    out disableComplete ();
    out txStartQueue ();
    out txStopQueue ();
    in  txTimeout ();
    in  txPacket (sk_buff * packet);
    out txPacketDone (sk_buff * packet);
    out txPacketAbort (sk_buff * packet);
    out rxPacketInput (sk_buff * packet);
    out linkUp ();
    out linkDown ();
    in  setMacAddress (size_t size, void * buf);
    out setMacAddressComplete ();
    in  getMacAddress (size_t size, void * buf);
    out getMacAddressComplete (size_t size);
    in  setMulticast (u32 mc_flags, size_t mc_count, void * mc_list);
    out setMulticastComplete ();
  variables:
    size_t txCount;
  dependencies:
    Lifecycle lc {
      listens   probe;
      listens   probeComplete;
      listens   probeFailed;
      listens   unplugged;
      restricts stop;
      restricts stopComplete;
    };
    PowerManagement pm {
      restricts suspend;
      restricts suspendComplete;
      listens   resumeComplete;
    };
  transitions:
      import(format=rhapsody,
             location="EthernetControllerSM@ioprotocols.sbs");
};
```

Figure B.5: The EthernetController protocol declaration.

`setMacAddressComplete` - MAC-address change complete.

`getMacAddress` - Read controller MAC address.

`getMacAddressComplete` - MAC-address read complete.

`setMulticast` - Set the list of multicast addresses that the controller should listen to.

`setMulticastComplete` - Multicast list stored in the device.

B.3.2 The `EthernetController` protocol state machine

After completing device initialisation (transition from state `starting` to `running` in Figure B.6), the protocol participates in three concurrent activities described by `link_status`, `properties`, and `tx_rx` superstates. These activities are interrupted when the device is unplugged, suspended, or stopped.

The `link_status` state (Figure B.7) describes how the driver reports link status to the OS. In the initial state, the driver must determine the current link status and report it to the OS within 10 seconds. Afterwards, the driver only reports the link status when it changes.

The `properties` state (Figure B.8) describes the configuration interface of the driver, which consists of three operations: reading the controller MAC address, modifying the controller MAC address, and setting the list of multicast addresses that the controller should listen to.

The `tx_rx` state (Figure B.9) describes how the driver exchanges network packets with the OS. The OS must enable the driver before sending or receving packets via the network interface. Upon receiving an `enable` request, the driver has 5 seconds to enable the transmit and receive circuits of the device (the `enable` state in Figure B.9) and respond by invoking the `enableComplete` callback.

The `enabled` state is split into two parallel regions: the top region describes the packet transmission protocol; the bottom region describes packet reception. The driver signals when it's ready to transmit a packet by calling the `txStartQueue` method, which switches the transmit protocol state machine to the `txq_running` state. If the OS enqueues new packets faster than the controller can transmit them, the hardware buffers inside the controller will eventually become full. In this case, the driver sends a `txStopQueue` notification to the OS to prevent it from sending new packets.

The receive part of the protocol does not support OS-driven flow control and consists of a single `rxPacketInput` method, which delivers an incoming packet to the OS.

The `disable` command interrupts the transmit and receive operations of the driver. In response to this command, the driver must abort all outstanding packets (the `disable` state in Figure B.9), disable the device and notify the OS via the `disableComple` callback.

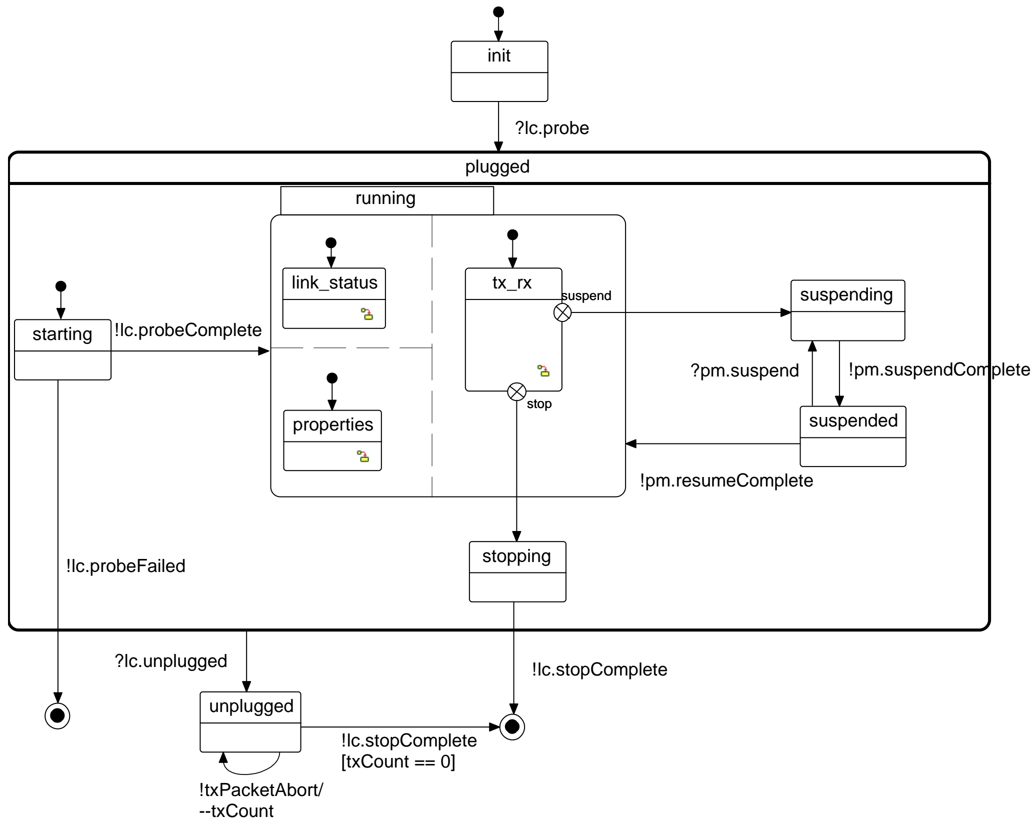


Figure B.6: The top-level `EthernetController` protocol state machine.

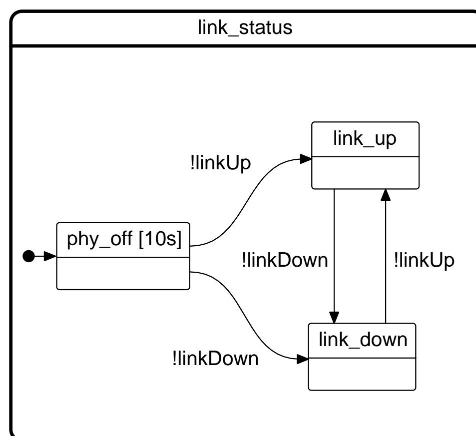


Figure B.7: The `link_status` state of the `EthernetController` protocol state machine expanded.

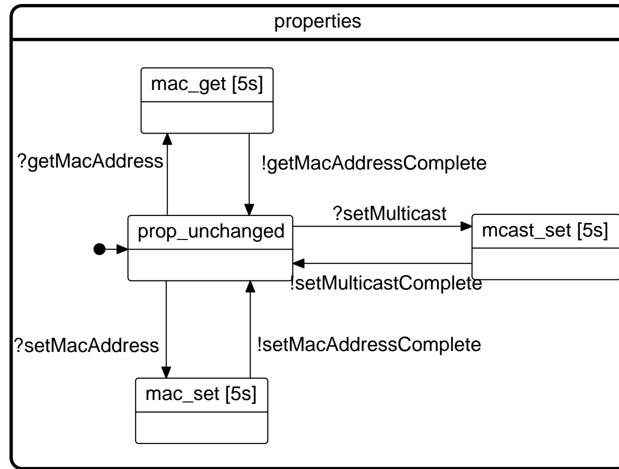


Figure B.8: The `properties` state of the `EthernetController` protocol state machine expanded.

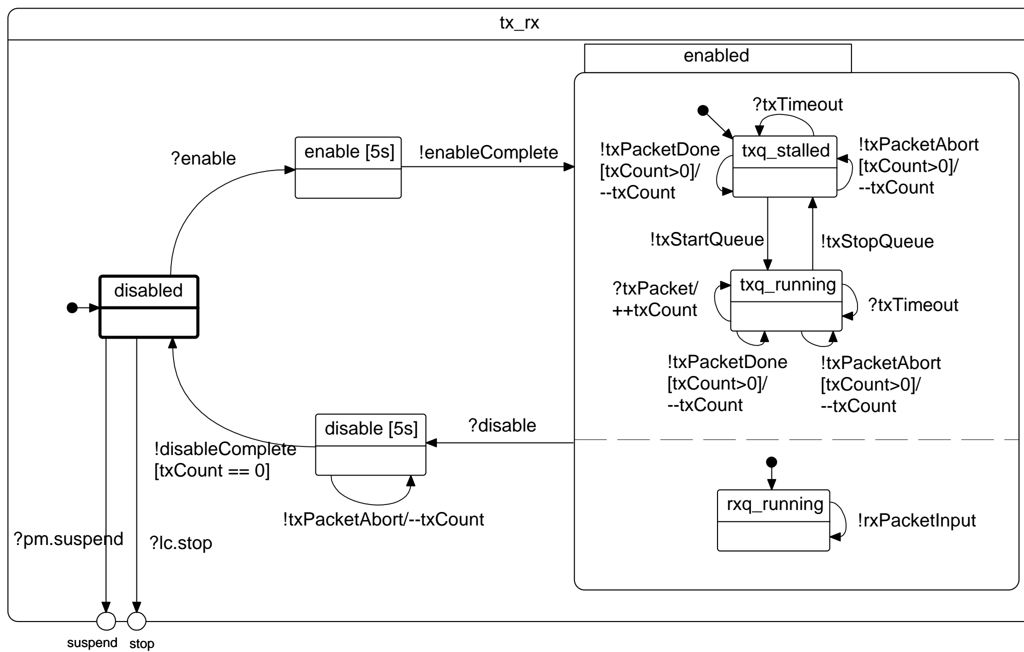


Figure B.9: The `tx_rx` state of the `EthernetController` protocol state machine expanded.

B.4 The `USBInterfaceClient` protocol

The `USBInterfaceClient` protocol defines the service provided by the USB bus framework to a USB device driver. This protocol allows the driver to access a single interface of the device. Multiple-interface devices are typically managed by multiple drivers, one for each interface of the device. However, a single driver can control multiple USB interfaces by implementing several ports of type `USBInterfaceClient`. The protocol specification is shown in Figures B.10 and B.11.

The `USBInterfaceClient` protocol is concerned with selecting the interface configuration and opening USB pipes. The actual USB data transfers occur via pipes whose behaviour is described by the `USBPipeClient` protocol shown in Figures B.12 and B.13.

B.4.1 `USBInterfaceClient` methods

`altsettingSelect` - Choose a different alternate setting for the interface. A USB interface can support different modes of operation that require different interface configurations. The driver selects the desired configuration by choosing the corresponding alternate setting. The list of available configurations and their parameters is specified in the device descriptor accessible through the metadata interface not covered here. Each alternate setting supports a different set of USB endpoints. When a different setting is selected, all pipes connecting the driver to currently used endpoints are closed (see the description of the `USBPipeClient` protocol below).

`altsettingSelectComplete` - Alternate setting successfully selected.

`altsettingSelectFailed` - Alternate setting select failed.

`pipeOpen` - Open a USB pipe to the specified device endpoint. This method spawns a new `pipe` port, which implements the `USBPipeClient` protocol for transferring data over the pipe. When the driver invokes this method, the USB framework allocates a USB pipe and binds it to the provided port, so that the driver can immediately start using the pipe through this port.

B.4.2 `USBPipeClient` methods

`transferStart` - Start a USB transfer over the pipe.

`transferStalled` - USB transfer failed and the pipe was stalled. Outstanding transfers will not complete and must be aborted. No new transfers can be started until the pipe is resumed.

`transferFailed` - USB transfer completed with an error; the pay remains operational and will keep processing outstanding requests.

```
protocol USBInterfaceClient
{
  types:
    unsigned<8> usb_altsetting_num_t;
    unsigned<32> usb_endpoint_addr_t;
    enum usb_xfer_type_t {
      USB_CONTROL = 0,
      USB_BULK = 1,
      USB_INTERRUPT = 2,
      USB_ISOCHRONOUS = 3
    };
  methods:
    out altsettingSelect (usb_altsetting_num_t alternate);
    in  altsettingSelectComplete ();
    in  altsettingSelectFailed ();
    out pipeOpen (usb_endpoint_addr_t address,
                  usb_xfer_type_t type) spawns pipe;
  dependencies:
    Lifecycle lc {
      restricts probeFailed;
      listens   probeComplete;
      listens   unplugged;
    };
    PowerManagement pm {
      restricts suspendComplete;
      listens   resume;
    };
  ports:
    USBPipeClient pipe [usb_endpoint_addr_t] <self/iface, lc/lc, pm/pm>;
  transitions:
    import(format=rhapsody,
           location="USBInterfaceClientSM@ioprotocols.sbs");
};
```

Figure B.10: The `USBInterfaceClient` protocol declaration.

`transferComplete` - USB transfer completed successfully.

`abort` - Abort all transfers in the pipe. Used to clear a pipe stall before resuming the pipe or to flush the pipe without waiting for all transfers to complete before shutting down the device or switching to a different interface setting.

`transferAborted` - Transfer aborted as a result of an `abort` request.

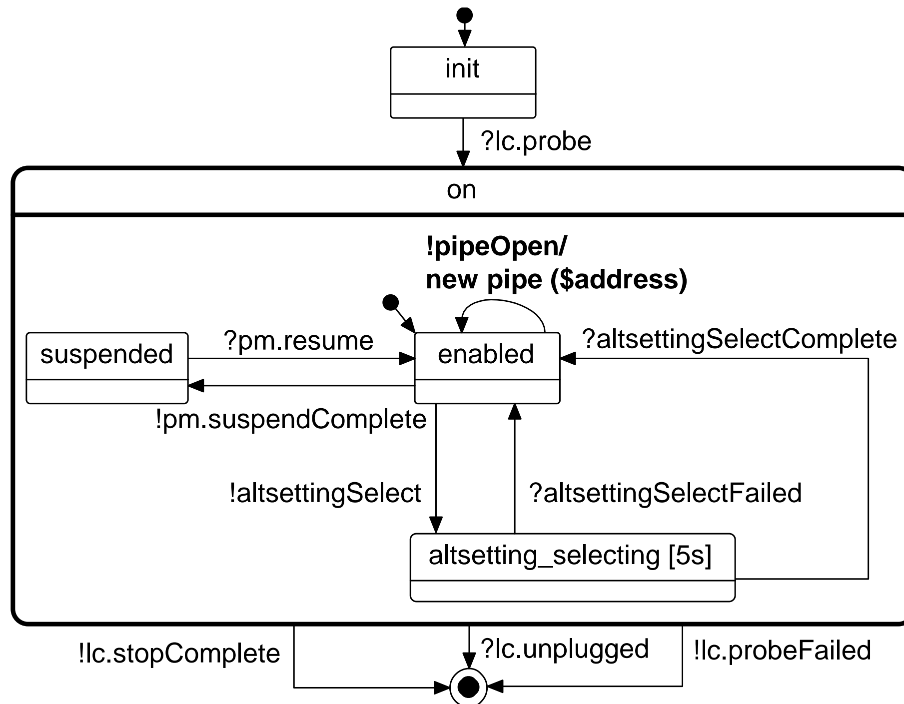


Figure B.11: The `USBInterfaceClient` protocol state machine.

`abortComplete` - Notifies the driver of the completion of an `abort` operation. Called after all transfers in the pipe have been aborted.

`resume` - Resume the pipe after a stall. Can only be called if there are no transfers remaining in the pipe.

`resumeComplete` - Notifies the driver of the completion of a `resume` request.

B.4.3 The `USBPipeClient` protocol state machine

The `USBPipeClient` protocol state machine shown in Figure B.13 has two features that have not appeared in protocols covered in the previous sections.

The first one is the use of the list *abstract data type (ADT)* to model the list of outstanding USB transfers. The `transfers` variable is declares in Figure B.12 as follows: `list<dingo_urb*> transfers;`. When a new transfer is started using the `transferStart` method, it is added to the back of the list using the `push_back` function. Whenever a transfer is completed using the `transferComplete`, `transferFailed`, `transferStalled`, or `transferAborted` callback, the protocol state machine asserts that the completed transfer must be the same as the one currently at the head of the transfer list (i.e., transfers must complete in the first-in-first-out order), and removes the transfer from the list.

Another feature of interest in this protocol is the use of protocol dependencies to synchronise with the parent `USBInterfaceClient` protocol. `USBInterfaceClient` defines the `altsettingSelect` method, which selects an alternate interface configu-

```
protocol USBPipeClient
{
  types:
    opaque dingo_urb; // USB request block

  methods:
    out transferStart (dingo_urb * request);
    in  transferStalled (dingo_urb * request);
    in  transferFailed (dingo_urb * request);
    in  transferComplete (dingo_urb * request);
    out abort ();
    in  transferAborted (dingo_urb * request);
    out resume ();
    in  resumeComplete ();
    in  abortComplete ();

  variables:
    // transfers queued in the pipe
    list<dingo_urb*> transfers;

  dependencies:
    USBInterfaceClient iface {
      restricts altsettingSelect;
    };

    Lifecycle lc {
      listens   unplugged;
      restricts probeFailed;
      restricts probeComplete;
    };

    PowerManagement pm {
      restricts suspendComplete;
      listens   resume;
    };

  transitions:
    import(format=rhapsody,
           location="USBPipeClientSM@ioprotocols.sbs");
};
```

Figure B.12: The `USBPipeClient` protocol declaration.

ration. Calling this method terminates all active USB pipes. To prevent resource leaks, the driver is only allowed to invoke this method when there are no outstanding transfers in any of its pipes. In order to express this constraint, the `USBPipeClient` protocol declares a `restricts` dependency on the `altsettingSelect` method of the `USBInterfaceClient` protocol (see the dependencies section in Figure B.12). The `iface.altsettingSelect` transitions in Figure B.13 are guarded by the `transfers.size()==0` expression, which ensures that the pipe is empty when the corresponding method of the parent protocol is invoked.

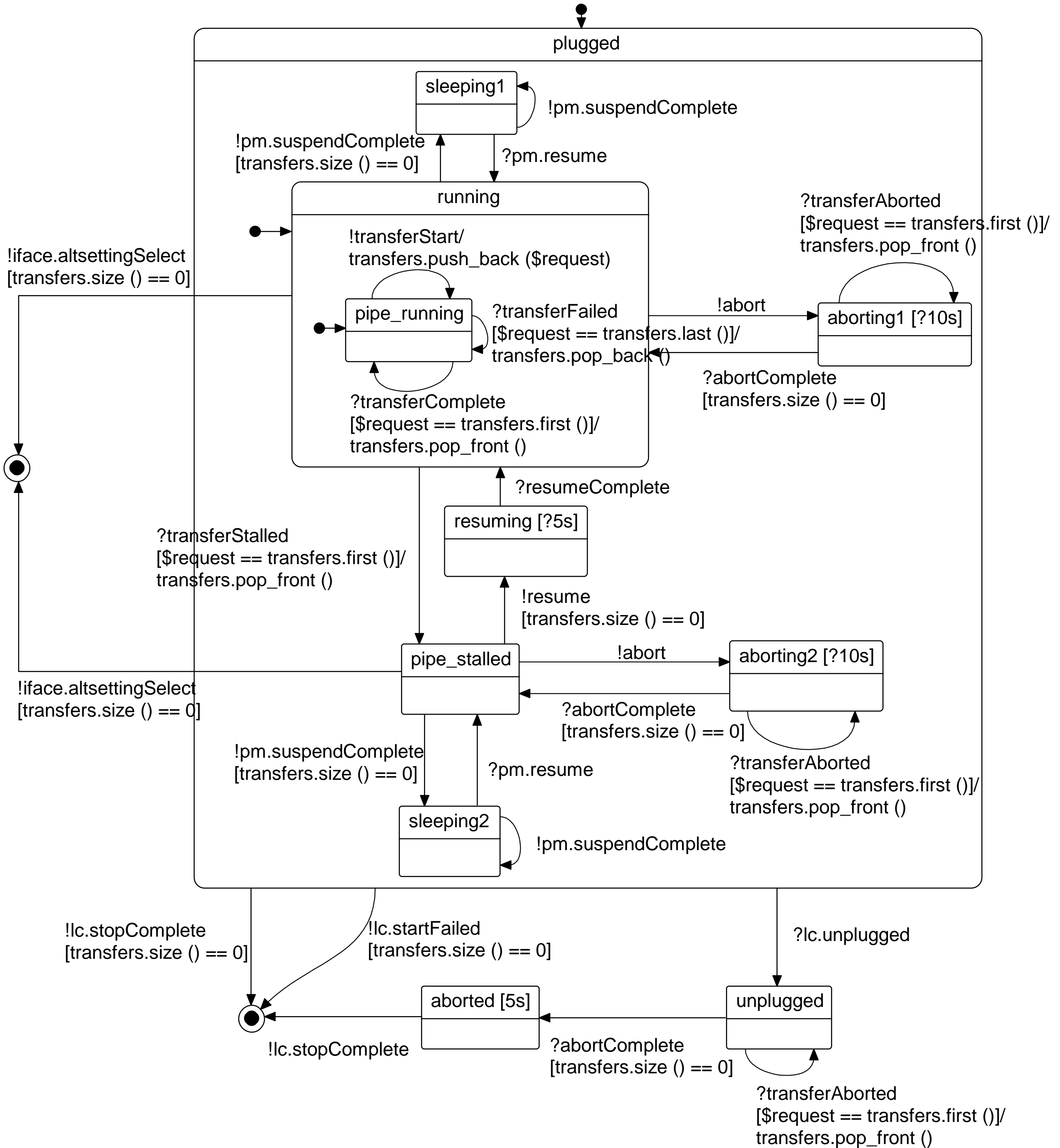


Figure B.13: The `USBPipeClient` protocol state machine.

B.5 The `InfiniBandController` protocol

The `InfiniBandController` protocol describes the service that an InfiniBand controller driver must provide to the OS. The InfiniBand architecture is designed to enable high-throughput, low-latency, and low-CPU-overhead communication between computer systems [Inf08]. This is achieved using high-speed interconnect technology and sophisticated host-side architecture, which supports *remote direct memory access (RDMA)*, traffic isolation, and other performance enhancement features. RDMA allows one of the communicating hosts to issue a command to read or write a block of data in the remote host memory. The command is completed by the local and remote InfiniBand controllers without interrupting the execution of the remote host and does not require an extra memory copy operation on either side. Traffic isolation is achieved by providing support for multiple prioritised communication endpoints in the hardware.

In order to allow applications to leverage these mechanisms, the InfiniBand controller driver must export an interface to a number of hardware objects to the OS. These objects and relations between them are illustrated by the UML class diagram in Figure B.14.

Queue pairs A queue pair represents an InfiniBand communication endpoint, similar to a network socket. A queue pair consists of a request queue and response queue. The client writes a command to a request queue in order to initiate communication with the remote host (e.g., send a message or perform an RDMA write to the remote memory). The response queue contains memory buffers, which the controller fills with messages received from the remote endpoint. A queue pair can have a private response queue or share a response queue with several other queue pairs. In the latter case, a special object type, shared response queue, is used.

Completion queues Completion queues store results of completed request and response operations. Every queue pair is assigned a request and a response completion queues. Depending on application-level needs, this can be the same or different queues. Moreover, completion queues can be shared among multiple queue pairs.

Protection domains Protection domains is a security mechanism that allows the user to control which memory regions can be accessed via RDMA operations through a particular queue pair. A domain consists of a set of host memory regions. Every queue pair is assigned to a protection domain at the time of creation. The host controller ensures that the remote endpoint can only access memory regions inside the protection domain of the queue pair.

User contexts InfiniBand controllers are typically designed to allow user-level applications direct access to the controller, avoiding costly system calls. To this end, an application can register a user context consisting of memory regions that are mapped to the user address space, providing direct access to queue pairs and completion queues.

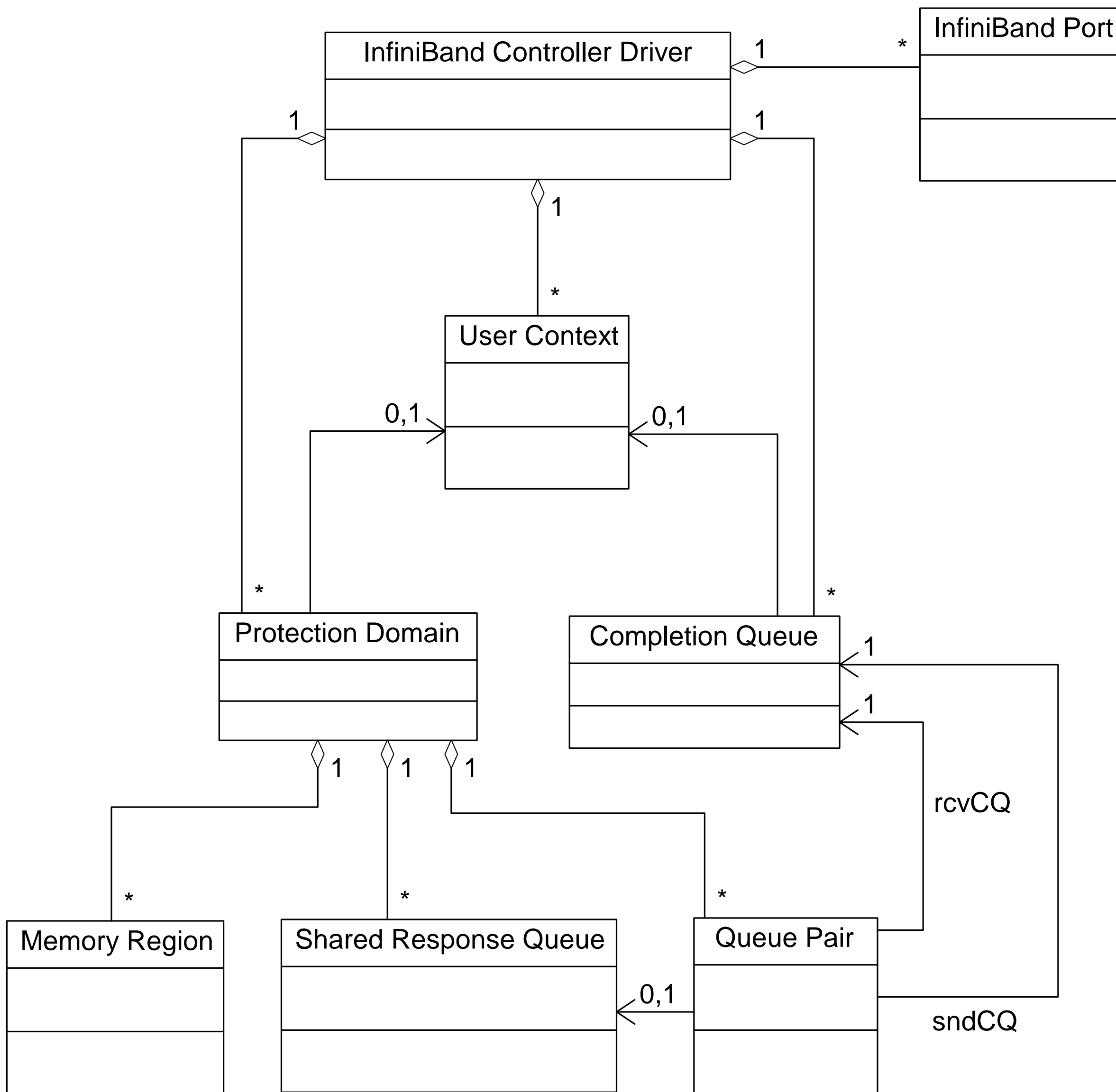


Figure B.14: Types of objects exported by an InfiniBand controller driver to the OS.

The specification of the `InfiniBandController` protocol and its subprotocols are shown in Figures B.15–B.26. The protocol uses a number of data structures declared opaque. The actual definition of these structures can be found in Linux kernel header files.

B.5.1 `InfiniBandController` methods

`advertiseControllerProperties` - Advertise InfiniBand device attributes during initialisation ().

`advertisePort` - Advertise a physical port of the controller and its attributes. This method is called by the driver during initialisation once for each port of the device. This method spawns a new Tingu subport through which the OS controls the physical port via the `IBPort` protocol.

`queryDevice, queryDeviceFailed, queryDeviceComplete` - Query InfiniBand device attributes.

`modifyDevice, modifyDeviceFailed, modifyDeviceComplete` - Modify device attributes.

`allocUContext, allocUContextComplete, allocUContextFailed` - Allocate a new user context.

`freeUContext, freeUContextComplete` - Deallocate a user context.

`allocPD, allocPDFailed, allocPDComplete` - Allocate a new InfiniBand protection domain. Spawns a new port through which the OS controls new protection domain via the `IBProtectionDomain` protocol.

`createCQ, createCQFailed, createCQComplete` - Allocate a new completion queue. Spawns a new port through which the OS controls the completion queue via the `IBCompletionQueue` protocol.

`catastrophicError` - Report a catastrophic device error to the OS.

```
protocol InfiniBandController {
  types:
    opaque struct ib_device_attr;
    opaque struct ib_port_attr;
    opaque struct dib_udata;
    opaque struct ib_ucontext;
    opaque struct ib_device;
    opaque struct ib_device_modify;
    opaque struct vm_area_struct;
    opaque struct task_struct;
  messages:
    out advertiseControllerProperties (ib_device_attr * props, u32 features);
    out advertisePort (ib_port_attr * props) spawns hcport;
    in  queryDevice (ib_device_attr * props);
    out queryDeviceFailed (error_t error);
    out queryDeviceComplete ();
    in  modifyDevice (u32 mask, ib_device_modify * props);
    out modifyDeviceFailed (error_t error);
    out modifyDeviceComplete ();
    in  allocUContext (dib_udata * udata);
    out allocUContextComplete (ib_ucontext * ctx);
    out allocUContextFailed (error_t error);
    in  freeUContext (ib_ucontext * ctx);
    out freeUContextComplete ();
    in  allocPD (ib_ucontext * ctx, dib_udata * udata);
    out allocPDFailed (error_t error);
    out allocPDComplete () spawns pd;
    in  createCQ (s32 entries, s32 vector, ib_ucontext *ctx,
                  dib_udata * udata, task_struct * task);
    out createCQFailed (error_t error);
    out createCQComplete () spawns cq;
    out catastrophicError ();
  dependencies:
    Lifecycle lc {
      listens probe; restricts probeFailed;
      restricts probeComplete; restricts stop;};
  variables:
    set<ib_ucontext*> contexts;
  ports:
    IBPort hcport [u8] <lc/lc>;
    IBProtectionDomain pd [] <lc/lc>;
    IBCompletionQueue cq [] <lc/lc>;
  transitions:
    import(format=rhapsody,location="InfiniBandControllerSM@ioprotocols.sbs");
};
```

Figure B.15: The `InfiniBandController` protocol declaration.

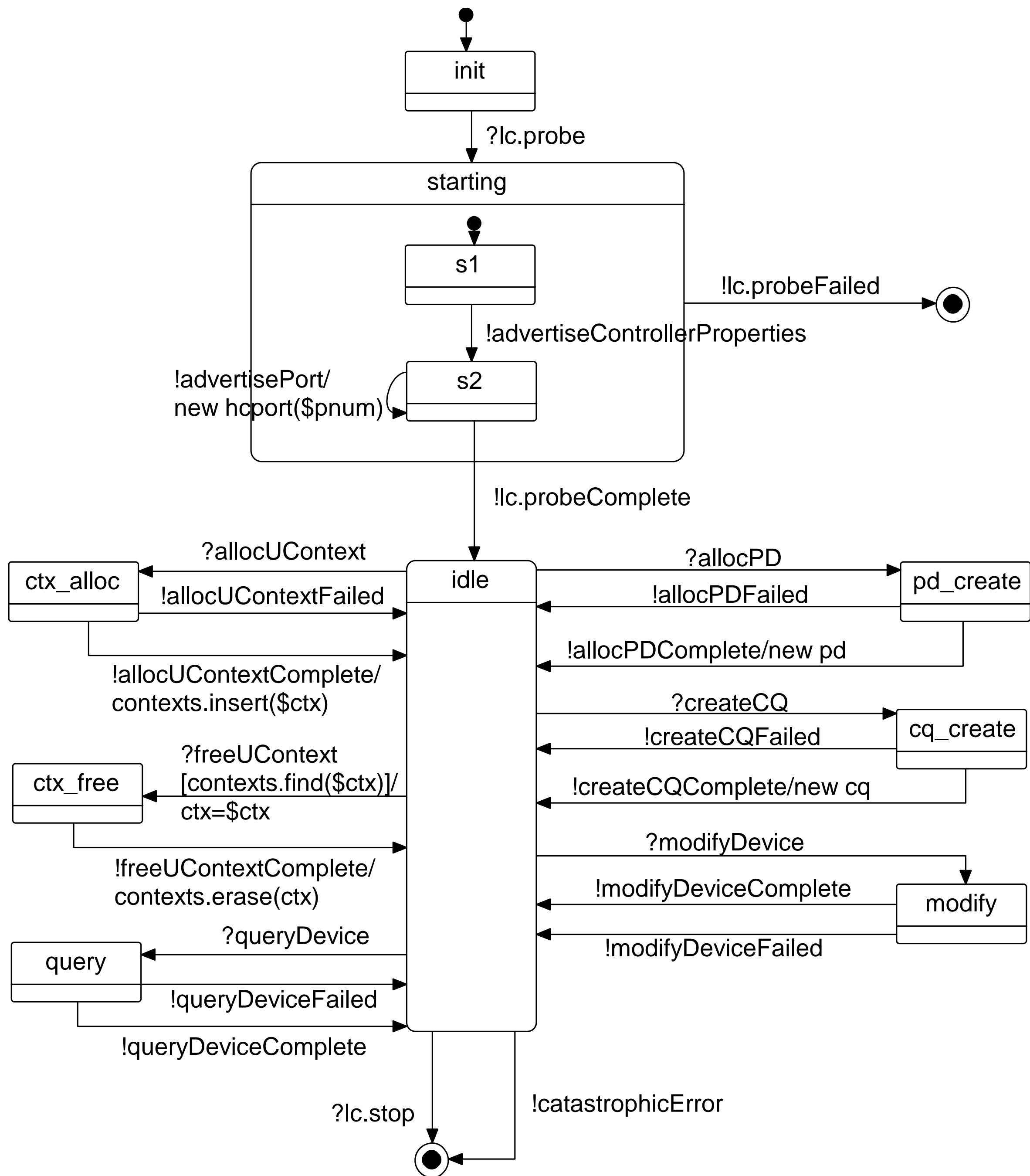


Figure B.16: The `InfiniBandController` protocol state machine.

B.5.2 `IBPort` methods

`active, error` - Report physical port error conditions.

`queryPort, queryPortFailed, queryPortComplete` - Query port attributes.

`modifyPort, modifyPortFailed, modifyPortComplete` - Modify port attributes.

`queryPKey, queryPKeyFailed, queryPKeyComplete` - Retrieve a network partition key in the partition key table associated with the port.

`queryGID, queryGIDFailed, queryGIDComplete` - Retrieve port's global identifier (the identifier that uniquely identifies the port inside a multicast group).

```
protocol IBPort
{
  types:
    opaque struct ib_port_modify;
    opaque union ib_gid;
    opaque struct ib_grh;
    opaque struct ib_mad;
    opaque struct ib_mad_send_buf;
    opaque struct ib_wc;
    opaque enum ib_wc_status;
    ib_mad_send_buf * pib_mad_send_buf;
  messages:
    out active ();
    out error ();
    in  queryPort (ib_port_attr * props);
    out queryPortFailed (error_t error);
    out queryPortComplete ();
    in  modifyPort (s32 mask, ib_port_modify * props);
    out modifyPortFailed (error_t error);
    out modifyPortComplete ();
    in  queryPKey (u16 index, u16 * pkey);
    out queryPKeyFailed (error_t error);
    out queryPKeyComplete ();
    in  queryGID (s32 index, ib_gid * gid);
    out queryGIDFailed (error_t error);
    out queryGIDComplete ();
  dependencies:
    Lifecycle lc {restricts stop;};
  transitions:
    import(format=rhapsody,location="IBPortSM@ioprotocols.sbs");
};
```

Figure B.17: The BPort protocol declaration.

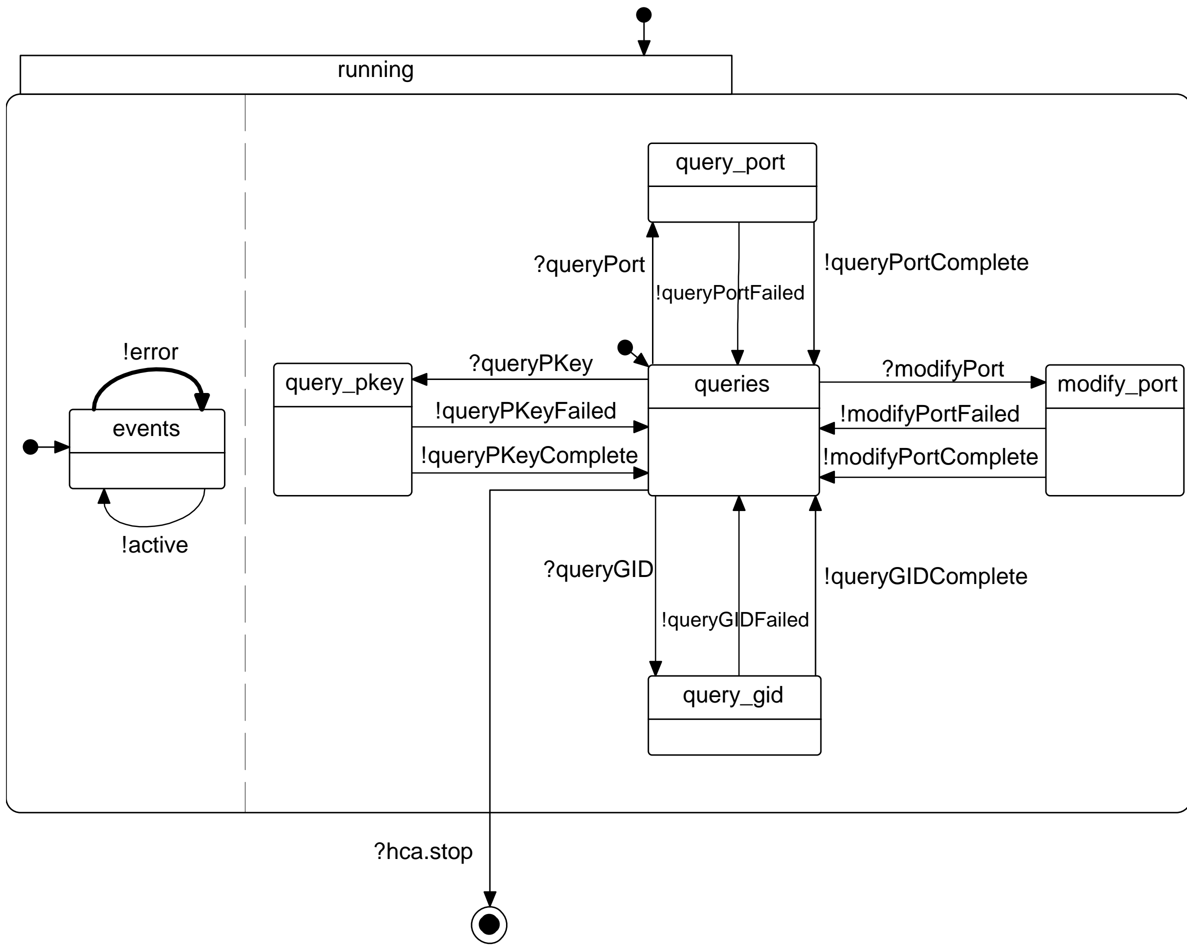


Figure B.18: The `IBPort` protocol state machine.

B.5.3 `IBProtectionDomain` methods

`createQP, createQPFailed, createQPComplete` - Allocate a new queue pair inside the domain. Spawns a new port implementing the `IBQueuePair` protocol.

`getDMAMR, getDMAMRFailed, getDMAMRComplete` - Retrieve a memory region descriptor for a memory region registered with the protection domain.

`regUserMR, regUserMRFailed, regUserMRComplete` - Add a new user memory region to the domain.

`regPhysMR, regPhysMRFailed, regPhysMRComplete` - Add a new physical memory region to the domain.

`deregMR, deregMRComplete` - Remove a memory region from the domain.

`createSRQ, createSRQFailed, createSRQComplete` - Create a new shared response queue. Spawns a new port implementing the `IBSharedRQ` protocol.

`free, freeComplete` - Destroy the protection domain.

```
protocol IBProtectionDomain {
  types:
    opaque struct ib_qp_init_attr;
    opaque struct ib_mr;
    opaque struct ib_srq_init_attr;
    opaque struct ib_phys_buf;
    ib_ah * pib_ah;
  messages:
    in  createQP (IBCompletionQueue rcq, IBCompletionQueue scq,
                  IBSharedRQ srq, ib_qp_init_attr *init_attr,
                  dib_udata *udata, task_struct * task);
    out createQPFailed (error_t error);
    out createQPComplete () spawns qp;
    in  getDMAMR (s32 acc);
    out getDMAMRFailed (error_t error);
    out getDMAMRComplete (ib_mr * mr);
    in  regUserMR (u64 start, u64 length, u64 virt_addr,
                   s32 access_flags, dib_udata *udata);
    out regUserMRFailed (error_t error);
    out regUserMRComplete (ib_mr * mr);
    in  regPhysMR (ib_phys_buf *buffer_list, s32 num_phys_buf,
                   s32 acc, u64 *iova_start);
    out regPhysMRFailed (error_t error);
    out regPhysMRComplete (ib_mr * mr);
    in  deregMR (ib_mr * mr);
    out deregMRComplete ();
    in  createSRQ (ib_srq_init_attr *init_attr, dib_udata *udata,
                   task_struct * task);
    out createSRQFailed (error_t error);
    out createSRQComplete () spawns srq;
    in  free ();
    out freeComplete ();
  dependencies:
    Lifecycle lc {restricts stop;};
  variables:
    set<ib_mr*> mrs;
  ports:
    IBQueuePair qp[]<self/pd>;
    IBSharedRQ srq[]<self/pd>;
  transitions:
    import(format=rhapsody,location="IBProtectionDomainSM@ioprotocols.sbs");
};
```

Figure B.19: The `IBProtectionDomain` protocol declaration.

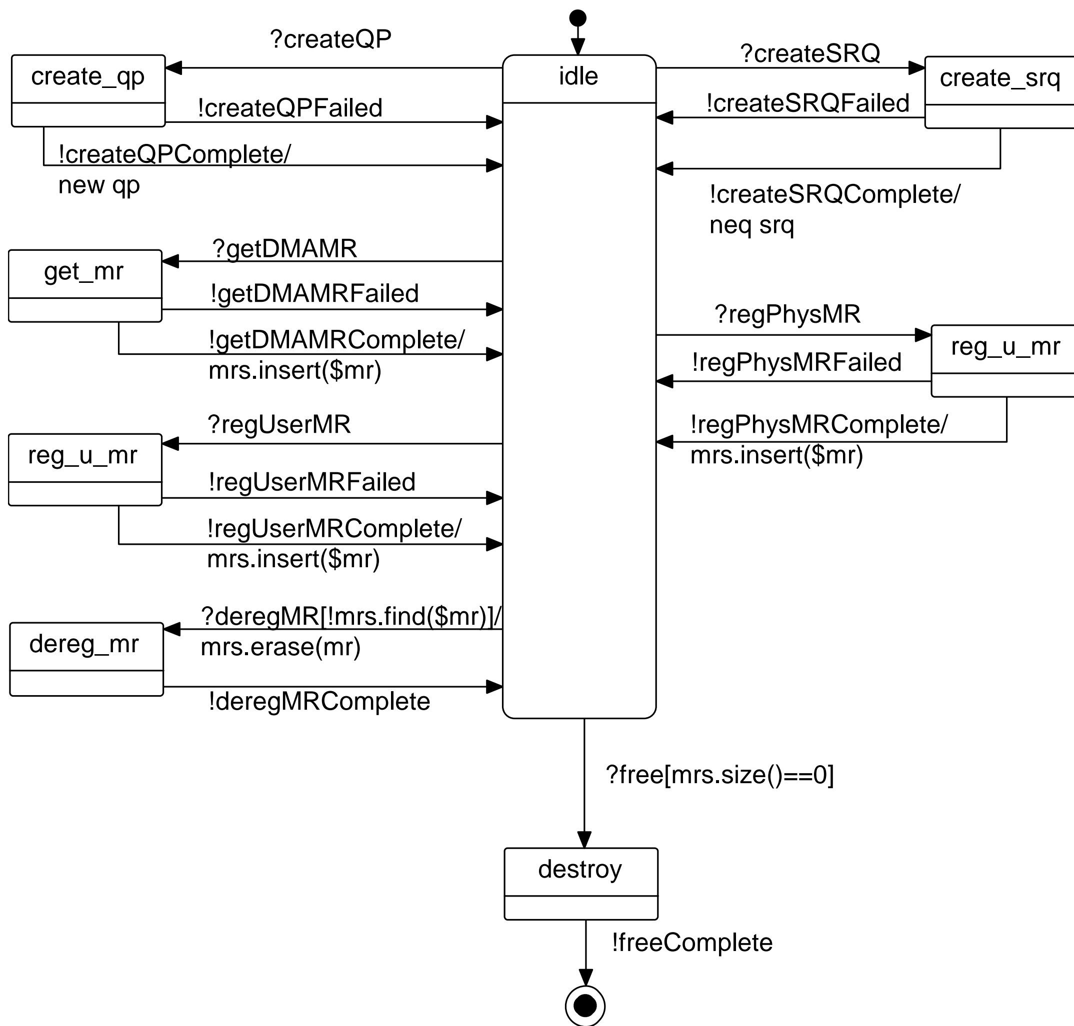


Figure B.20: The `IBProtectionDomain` protocol state machine.

B.5.4 `IBQueuePair` methods

`initialise, initialiseFailed, initialiseComplete` -

`recvEnable, recvEnableFailed, recvEnableComplete` -

`sendEnable, sendEnableFailed, sendEnableComplete` -

`setErrorState, setErrorStateFailed, setErrorStateComplete` -

`reset, resetFailed, resetComplete` -

`sqDrainStart, sqDrainStartFailed, sqDrainStartComplete` -

`sqDrained` - The InfiniBand standard defines several states through in which a queue pair can be during its lifecycle: RESET, READY-TO-RECEIVE, READY-TO-SEND, SEND-QUEUE-DRAIN, and ERROR. The above methods implement transitions between these states.

`query, queryFailed, queryComplete` - Query queue pair attributes.

`modify, modifyFailed, modifyComplete` - Modify queue pair attributes.

`postSend` - Post a new send or RDMA request.

`postRecv` - Post a new receive request.

`destroy, destroyComplete` - Destory the queue pair.

```
protocol IBQueuePair
{
  types:
    opaque struct ib_qp_attr;
    opaque struct ib_qp;
  messages:
    in  initialise (ib_qp_attr * attr, s32 attr_mask, dib_udata * udata);
    out initialiseFailed (error_t error);
    out initialiseComplete ();
    in  recvEnable (ib_qp_attr * attr, s32 attr_mask, dib_udata * udata);
    out recvEnableFailed (error_t error);
    out recvEnableComplete ();
    in  sendEnable (ib_qp_attr * attr, s32 attr_mask, dib_udata * udata);
    out sendEnableFailed (error_t error);
    out sendEnableComplete ();
    in  sqDrainStart (ib_qp_attr * attr, s32 attr_mask, dib_udata * udata);
    out sqDrainStartFailed (error_t error);
    out sqDrainStartComplete ();
    out sqDrained ();
    in  setErrorState (ib_qp_attr * attr, s32 attr_mask, dib_udata * udata);
    out setErrorStateFailed (error_t error);
    out setErrorStateComplete ();
    in  reset (ib_qp_attr * attr, s32 attr_mask, dib_udata * udata);
    out resetFailed (error_t error);
    out resetComplete ();
    in  query (ib_qp_attr *attr, s32 attr_mask, ib_qp_init_attr * init_attr);
    out queryFailed (error_t error);
    out queryComplete ();
    in  modify (ib_qp_attr * attr, s32 attr_mask, dib_udata * udata);
    out modifyFailed (error_t error);
    out modifyComplete ();
    in  postSend (ib_send_wr * wr, pib_send_wr * bad_wr, out error_t error);
    in  postRecv (ib_recv_wr * wr, pib_recv_wr * bad_wr, out error_t error);
    in  destroy ();
    out destroyComplete ();
  dependencies:
    IBProtectionDomain pd {restricts free;};
  transitions:
    import(format=rhapsody,location="IBQueuePairSM@ioprotocols.sbs");
};
```

Figure B.21: The `IBQueuePair` protocol declaration.

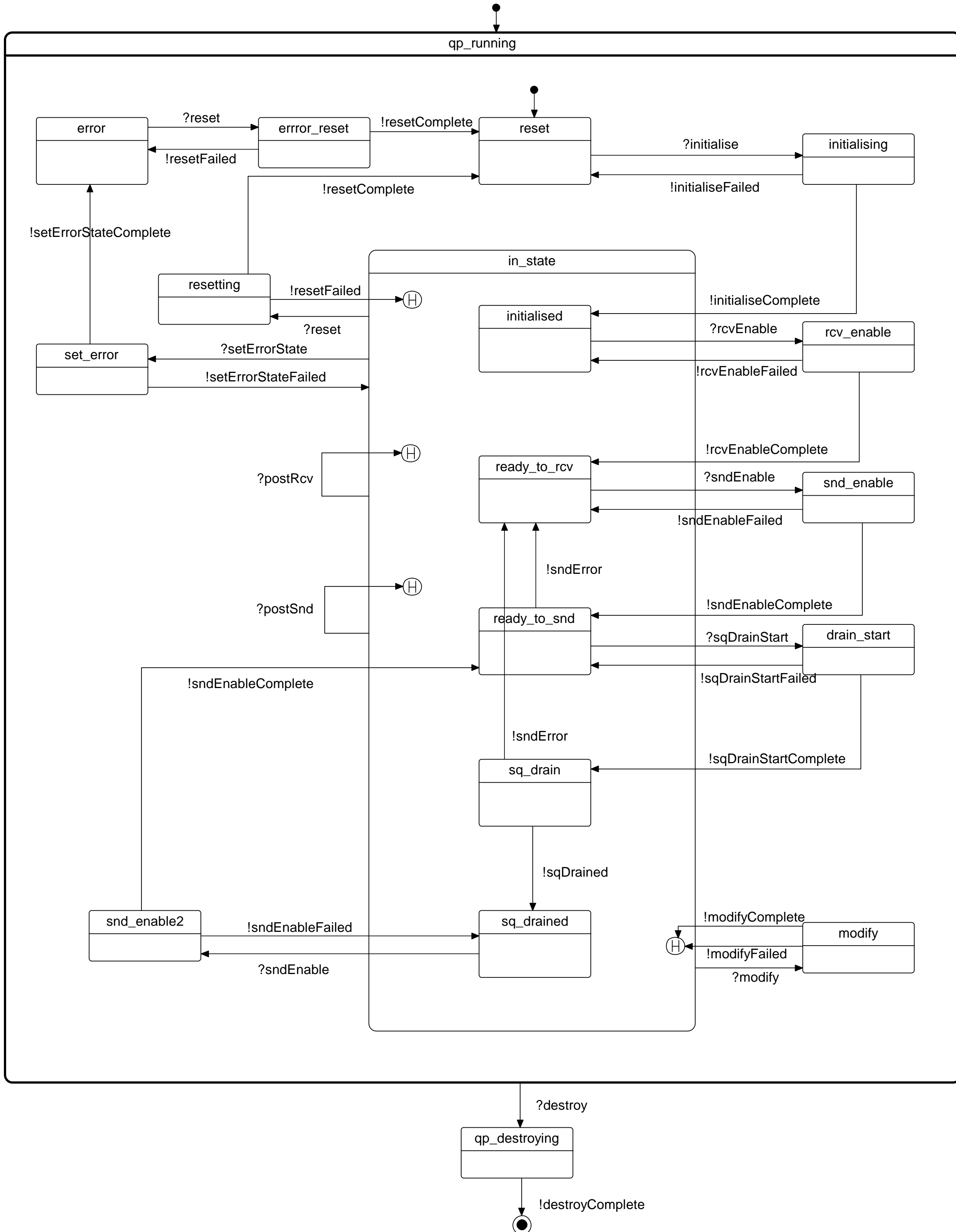


Figure B.22: The `IBQueuePair` protocol state machine.

B.5.5 `IBSharedRQ` methods

`query, queryFailed, queryComplete` - Query queue attributes.

`modify, modifyFailed, modifyComplete` - Modify queue attributes.

`postRecv` - Post a new receive request.

`arm, armFailed, armComplete` - Arm the queue to signal to the OS when the number of queue entries drops below the specified threshold.

`limitReached` - Notify the OS that the queue has reached the threshold.

`error` - Report an error.

`destroy, destroyComplete` - Destory the queue.

```
protocol IBSharedRQ
{
  types:
    opaque struct ib_srq_attr;
    opaque enum ib_srq_attr_mask;
    opaque struct ib_recv_wr;
    opaque struct ib_send_wr;
    opaque struct ib_srq;
    ib_recv_wr * pib_recv_wr;
    ib_send_wr * pib_send_wr;
  messages:
    in  query (ib_srq_attr * srq_attr);
    out queryFailed (error_t error);
    out queryComplete ();
    in  modify (ib_srq_attr *attr, ib_srq_attr_mask attr_mask,
                dib_udata *udata);
    out modifyFailed (error_t error);
    out modifyComplete ();
    in  postRecv (ib_recv_wr * wr, pib_recv_wr * bad_wr,
                  out error_t error);
    in  arm (u32 srq_limit);
    out armFailed (error_t error);
    out armComplete ();
    out limitReached ();
    out error ();
    in  destroy ();
    out destroyComplete ();
  dependencies:
    IBProtectionDomain pd {
      restricts free;
    };
  transitions:
    import(format=rhapsody,location="IBSharedRQ@ioprotocols.sbs");
};
```

Figure B.23: The `IBSharedRQ` protocol declaration.

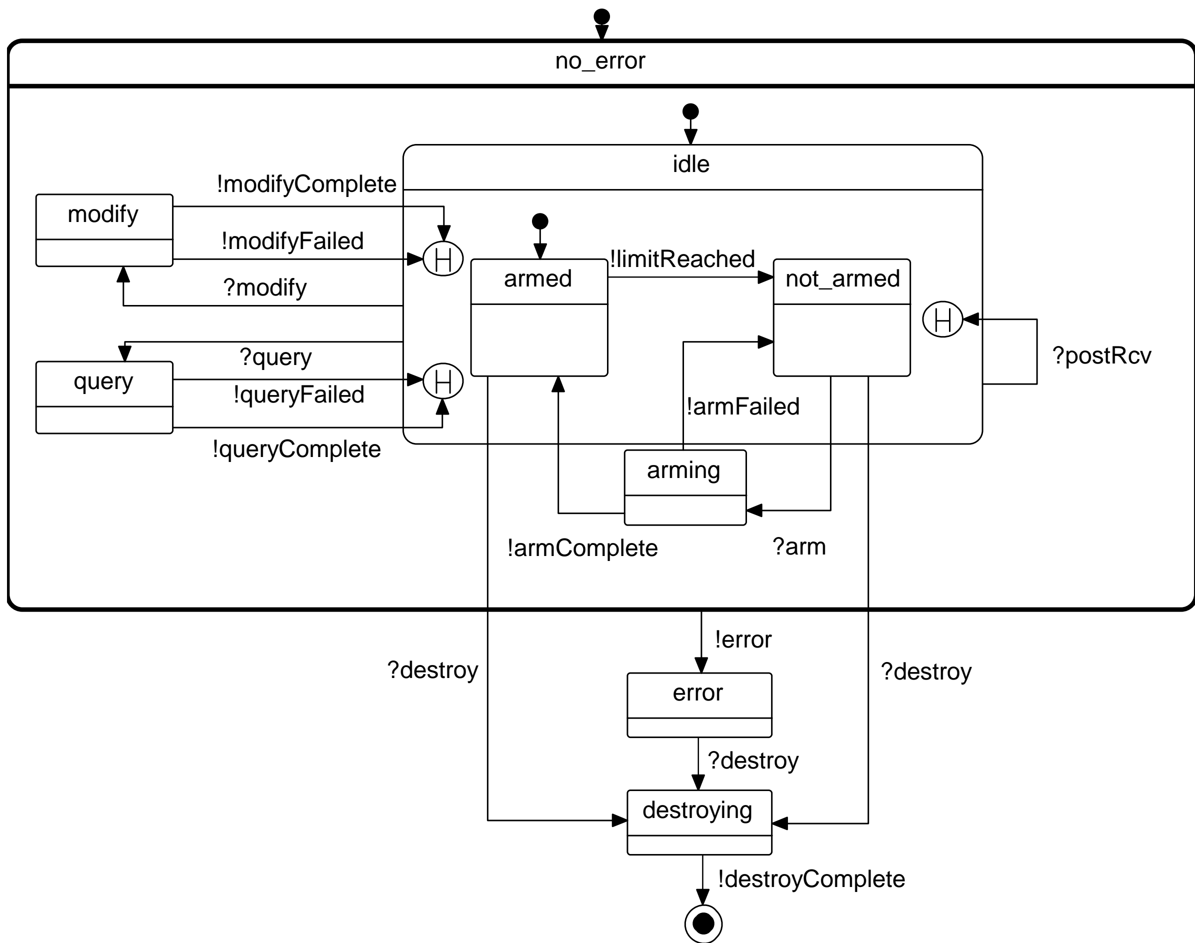


Figure B.24: The `IBSharedRQ` protocol state machine.

B.5.6 `IBCompletionQueue` methods

`poll` - Dequeue the requested number of completion entries.

`arm` - Arm the completion queue to notify the OS about completed requests added to the queue.

`notify` - Notify the OS about completed requests added to the queue.

`resize, resizeComplete, resizeFailed` - Change the number of entries in the queue.

`error` - Report an error.

`destroy, destroyComplete` - Destroy the queue.

```
protocol IBCompletionQueue
{
  types:
    opaque enum ib_cq_notify_flags;
    opaque struct ib_cq;
  messages:
    in  poll (u32 nentries, ib_wc * completions, out int ret);
    out notify (u32 nentries);
    in  arm (ib_cq_notify_flags flags, out error_t err);
    in  resize (s32 entries, dib_udata * udata);
    out resizeComplete ();
    out resizeFailed (error_t error);
    out error ();
    in  destroy ();
    out destroyComplete ();
  dependencies:
    Lifecycle lc {
      restricts stop;
    };
  transitions:
    import(format=rhapsody,
           location="IBCompletionQueueSM@ioprotocols.sbs");
};
```

Figure B.25: The IBCompletionQueue protocol declaration.

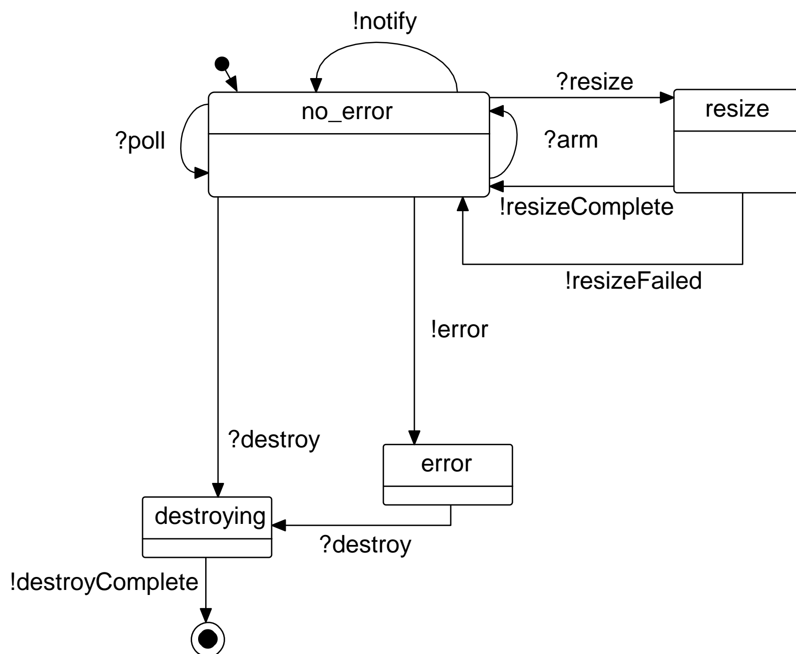


Figure B.26: The IBCompletionQueue protocol state machine.

Appendix C

The OpenCores SD host controller device specification

Figure C.1 shows the complete OpenCores SD host controller device [Edv] protocol specification. The corresponding device-class and OS protocol specifications are given in Section 6.5.

```
protocol SDHostOpenCores
{
types:
  /* device registers */
  struct command_reg {
    unsigned<2> RTS;
    unsigned<8> RESERVED;
    unsigned<6> CMDI;
  };
  /* argument register */
  struct argument_reg {
    unsigned<32> CMDA;    // command argument
  };
  /* card status register */
  struct status_reg {
    unsigned<1> CICMD;    // command inhibit
    unsigned<15> RESERVED;// reserved
  };
  /* command response register */
  struct response_reg {
    unsigned<32> CRSP; // response of the last command
  };
  /* software reset reg */
  struct reset_reg {
    unsigned<1> SRST;     // software reset
    unsigned<7> RESERVED; // reserved
  };
  /* normal interrupt status register */
  struct isr_reg {
    unsigned<1> CC;        // command complete
    unsigned<14> RESERVED; // reserved
    unsigned<1> EI;        // error interrupt
  };
  /* error interrupt status register */
  struct eisr_reg {
    unsigned<1> CTE;      // command timeout
    unsigned<1> CCRC;     // command CRC error
    unsigned<14> RESERVED;// reserved
  };
  /* clock divider register */
  struct clock_div_reg {
    unsigned<8> CLKD;        // clock divider
  };
```

Figure C.1: The OpenCores SD host controller device specification *(continued on the next page)*.

```
  /* BD buffer status register */
  struct bd_status_reg {
    unsigned<8> FBTX;         // free TX buffer descriptors
    unsigned<8> FBRX;         // free RX buffer descriptors
  };
  /* data interrupt status register */
  struct disr_reg {
    unsigned<1> TRS;          // transmission successful
    unsigned<1> TRE;          // transmission error
    unsigned<14> RESERVED;    // reserved
  };
  /* RX buffer descriptor register */
  struct bdrx_reg {
    unsigned<32> BDRX;
  };
  /* TX buffer descriptor register */
  struct bdtx_reg {
    unsigned<32> BDTX;
  };
  struct block_descr {
    unsigned<32> mem_addr;  //memory address
    unsigned<32> card_addr; //card address
  };
methods:
  /*register read/write methods */
  out write_command_reg (command_reg v);
  out read_command_reg (out command_reg v);
  out write_argument_reg (argument_reg v);
  out read_argument_reg (out argument_reg v);
  out read_status_reg (out status_reg v);
  out read_response_reg (out response_reg v);
  out write_reset_reg (reset_reg v);
  out read_reset_reg (out reset_reg v);
  out write_isr_reg (isr_reg v);
  out read_isr_reg (out isr_reg v);
  out write_eisr_reg (eisr_reg v);
  out read_eisr_reg (out eisr_reg v);
  out write_clock_div_reg (clock_div_reg v);
  out read_clock_div_reg (out clock_div_reg v);
  out read_bd_status_reg (out bd_status_reg v);
  out write_disr_reg (disr_reg v);
  out read_disr_reg (out disr_reg v);
  out write_bdrx_reg (bdrx_reg v);
  out write_bdtx_reg (bdtx_reg v);
  in irq ();
```

Figure C.1: The OpenCores SD host controller device specification *(continued)*.

```
dependencies:
  SDHostClass class {
    restricts on;
    restricts off;
    restricts commandOK;
    restricts commandError;
    restricts blockTransferOK;
    restricts blockTransferError;
    restricts busClockChange;
  };
variables:
  command_reg m_command_reg;
  reset_reg m_reset_reg;
  argument_reg m_argument_reg;
  status_reg m_status_reg;
  response_reg m_response_reg;
  isr_reg m_isr_reg;
  eisr_reg m_eisr_reg;
  bd_status_reg m_bd_status_reg;
  disr_reg m_disr_reg;
  clock_div_reg m_clock_div_reg;
  block_descr m_tx_descr;
  block_descr m_rx_descr;
  block_descr m_curr_descr;
  unsigned<1> m_new_command;
  unsigned<1> m_data_command;
  sdhost_command_t m_command;
transitions:
  write_reset_reg[$v.SRST==1]/{m_comand_reg=0;m_status_reg=0;...};
  write_reset_reg[$v.SRST==0];
  class.on;
  SDHOST
 where
  process SDHOST
  (REGISTERS
  |||
    (COMMAND_MASTER |[class.off]| DATA_MASTER)
  |||
    CLOCK_DIVIDER)
  [>
    write_reset_reg[$v.SRST == 1]/{m_comand_reg=0;
                                   m_status_reg=0; ...};
    write_reset_reg[$v.SRST==0];
    SDHOST
  endproc
```

Figure C.1: The OpenCores SD host controller device specification *(continued)*.

```
process CLOCK_DIVIDER
  write_clock_div_reg/m_clock_div_reg=$v;
  class.busClockChange[$divisor==m_clock_div_reg.CLKD];
  CLOCK_DIVIDER
endproc

process REGISTERS
    read_argument_reg[$v==m_argument_reg];
    REGISTERS
  []
    write_argument_reg[m_status_reg.CICMD==1]/m_argument_reg=$v;
    REGISTERS
  []
    write_argument_reg[m_status_reg.CICMD==0]
                   /{m_argument_reg=$v; m_new_command=1;
                     m_data_command=0; m_status_reg.CICMD=1;};
    REGISTERS
  []
    read_reset_reg[$v==m_reset_reg];
    REGISTERS
  []
    write_command_reg/m_command_reg = $v;
    REGISTERS
  []
    read_command_reg[$v == m_command_reg];
    REGISTERS
  []
    read_bd_status_reg[$v == m_bd_status_reg];
    REGISTERS
  []
    /* two consecutive writes to the BDTX register form
       a 64-bit buffer descriptor */
    write_bdtx_reg/m_tx_descr.mem_addr = $v.BDTX;
    write_bdtx_reg/{m_tx_descr.card_addr = $v.BDTX;
                    m_bd_status_reg.FBTX = 0;};
    REGISTERS
  []
    /* two consecutive writes to the BDRX register form
       a 64-bit buffer descriptor */
    write_bdrx_reg/m_rx_descr.mem_addr = $v.BDRX;
    write_bdrx_reg/{m_rx_descr.card_addr = $v.BDRX;
                    m_bd_status_reg.FBRX = 0;};
    REGISTERS
endproc
```

Figure C.1: The OpenCores SD host controller device specification *(continued)*.

```
process COMMAND_MASTER
  await[m_new_command==1]
       /{m_command.index=m_command_reg.CMDI;
         m_command.arg=m_argument_reg.CMDA;
         m_command.response=m_command_reg.RTS;
         m_command.data=m_data_command;
         m_new_command=0;};
  irq : timed;
  read_isr_reg/m_isr_reg=$v : timed;
  read_eisr_reg/m_eisr_reg=$v : timed;
  read_response_reg/m_response_reg=$v : timed;
  write_isr_reg[$v==0] : timed;
  write_eisr_reg[$v==0] : timed;
  (
  if[m_isr_reg.CC == 1]
    class.commandOK
      [($command==m_command) &&
       ($response==m_response_reg.CRSP)]
      /m_status_reg.CICMD=0 : timed;
    COMMAND_MASTER
  []
  else
    class.commandError
      [($command==m_command) &&
       ($status==(m_eisr_reg.CCRC ?
                  SDH_CMD_ECRC : SDH_CMD_ETIMEOUT))]
      /m_status_reg.CICMD=0 : timed;
    COMMAND_MASTER
  )
[]
  class.off;
  exit
endproc

process DATA_MASTER
/* A data transfer is triggered when either the TX or the
   RX buffer descriptor is written and the command inhibit
   bit is zero (i.e., the command master is not busy) */
```

Figure C.1: The OpenCores SD host controller device specification *(continued)*.

```
(
  /* The RX buffer descriptor has been written.  Start a
     read transfer.  The first step is to send command 17 to
     the card.  To this end, the data master writes command
     and argument registers and triggers the command master
     by generating the m_new_command signal.  The argument
     of command 17 is the address of the SD card block to be
     written.  Thid value is take from thr rx buffer
     descriptor (m_rx_descr.card_addr). */
  await[(m_bd_status_reg.FBRX == 0) &&
        (m_status_reg.CICMD == 0)]
       /{m_curr_descr = m_rx_descr;
         m_command_reg.CMDI = 17;  // issue command 17
         m_command_reg.RTS = 1; //expect a response from command
         m_argument_reg.CMDA = m_rx_descr.card_addr; // write
                           // block address to the arg register
         m_new_command = 1;
         m_data_command = 1;
         m_status_reg.CICMD = 1;};
  /* The data master then waits for the command master to
     complete the command. */
  (
    await[(m_status_reg.CICMD == 0) && (m_command_ok == 0)];
    /* If the command master fails, the data master deadlocks.
       The only way to get out of this deadlock is by resetting
       the controller (see the DATA_MASTER process above) */
    stop
  []
    /* If the command went through successfully, ... */
    await[(m_status_reg.CICMD == 0) && (m_command_ok == 1)];
    /* ...the data master starts a data transfer.  Completion
       of the transfer is indicated by an interrupt; the
       outcome is determined by values in the data interrupt
       status register.  If the transfer was successful, the
       TRS bit is set to 1.  In this case, a
       blockTransferComplete event is generated.  Otherwise, a
       blockTransferFailed event is generated.  In either case,
       the data master also sets the number of available rx
       descriptors in the BD status register
       (m_bd_status_reg.FBRX) to 1, indicating that it is
       ready for the next data transfer. */
    irq:timed;
    read_disr_reg/m_disr_reg = $v : timed;
    write_disr_reg :timed;
```

Figure C.1: The OpenCores SD host controller device specification *(continued)*.

```
    (
      if[m_disr_reg.TRS == 1]
        class.blockTransferComplete
          [($mem_addr == m_curr_descr.mem_addr)]
          /{m_disr_reg.TRS = 0; m_disr_reg.TRE = 0;
           m_bd_status_reg.FBRX = 1;} : timed;
        DATA_MASTER
      []
        else
          class.blockTransferFailed
            [($mem_addr == m_curr_descr.mem_addr) &&
             ($status == SDH_DATA_ERROR)]
            /{m_eisr_reg.CCRC = 0; m_eisr_reg.CTE = 0;
              m_bd_status_reg.FBRX = 1;} : timed;
          DATA_MASTER
    )
  )
)
[]
(
  /*Same for TX*/
  await[(m_bd_status_reg.FBTX == 0) && (m_status_reg.CICMD == 0)]
       /{m_curr_descr = m_tx_descr;
         m_command_reg.CMDI = 24;    // issue command 24
         m_command_reg.RTS = 1;
         m_argument_reg.CMDA = m_tx_descr.card_addr;
         m_new_command = 1;
         m_data_command = 1;
         m_status_reg.CICMD = 1;};
  (
    await[(m_status_reg.CICMD == 0) && (m_command_ok == 0)];
    stop
  []
    await[(m_status_reg.CICMD == 0) && (m_command_ok == 1)];
    irq:timed;
    read_disr_reg/m_disr_reg = $v : timed;
    write_disr_reg :timed;
```

Figure C.1: The OpenCores SD host controller device specification *(continued)*.

```
    (
      if[m_disr_reg.TRS == 1]
        class.blockTransferComplete
          [($mem_addr == m_curr_descr.mem_addr)]
          /{m_disr_reg.TRS = 0; m_disr_reg.TRE = 0;
            m_bd_status_reg.FBTX = 1; : timed};
            DATA_MASTER
    []
      else
        class.blockTransferFailed
          [($mem_addr == m_curr_descr.mem_addr) &&
           ($status == SDH_DATA_ERROR)]
          /{m_eisr_reg.CCRC = 0; m_eisr_reg.CTE = 0;
            m_bd_status_reg.FBTX = 1;}: timed;
        DATA_MASTER
    )
  )
)
[]
  class_off;
  stop
endproc
```

Figure C.1: The OpenCores SD host controller device specification *(the end)*.

Bibliography

[ABB+86] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the 1986 Summer USENIX Technical Conference*, pages 93–112, Atlanta, GA, USA, 1986.

[Adv00] Advanced Micro Devices, Inc. AMD I/O virtualization technology (IOMMU) specification. Revision 1.26, February 2000.

[AHT+02] A. Adya, J. Howell, M. Theimer, W. Bolosky, and J. Douceur. Cooperative task management without manual stack management. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 289–302, Monterey, CA, USA, June 2002.

[Ale72] Michael T. Alexander. Organization and features of the Michigan terminal system. In *AFIPS Conference Proceedings, 1972 Spring Joint Computer Conference*, pages 585–591, 1972.

[App06] Apple Inc. Introduction to I/O Kit fundamentals, November 2006.

[AS04] Zaher S. Andraus and Karem A. Sakallah. Automatic abstraction and verification of Verilog models. In *Proceedings of the 41st annual Design Automation Conference*, pages 218–223, San Diego, CA, USA, jun 2004.

[ASI08] ASIX Electronics Corporation. Ax88772 USB to 10/100 fast Ethernet/HomePNA controller, June 2008.

[BBC+06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *Proceedings of the 1st EuroSys Conference*, pages 73–85, Leuven, Belgium, April 2006.

[BCG+97] Felice Balarin, Massimiliano Chiodo, Paolo Giusto, Harry Hsieh, Attila Jurecska, Luciano Lavagno, Claudio Passerone, Alberto Sangiovanni-Vincentelli, Ellen Sentovich, Kei Suzuki, and Bassam Tabbara. *Hardware-software co-design for embedded systems: the Polis approach*. Kluwer Academic Press, 1997.

[BCLR04] Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Proceedings of the 4th International Conference on Integrated Formal Methods*, pages 1–20, 2004.

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, USA, October 2003.

[BH70] Per Brinch Hansen. The nucleus of a multiprogramming operating system. *Communications of the ACM*, 13:238–250, 1970.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast: Applications to software engineering. *International Journal on Software Tools for Technology Transfer*, 9(5):505–525, 2007.

[BHL00] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in KaffeOS: isolation, resource management, and sharing in Java. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 333–346, San Diego, California, October 2000.

[Bit] The BitKeeper Linux kernel repository. `http://linux.bkbits.net`.

[BPT07] Andreas Bauer, Markus Pister, and Michael Tautschnig. Tool-support for the analysis of hybrid systems and models. In *Proceedings of the 44th ACM/IEEE Conference on Design, Automation and Test in Europe*, pages 924–929, Nice, France, 2007.

[BR01] Thomas Ball and Sriram K. Rajamani. Slic: A specification language for interface checking. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[BS02] Herbert Bos and Bart Samwel. Safe kernel programming in the OKE. In *Proceedings of the Fifth IEEE Conference on Open Architectures and Network Programming*, pages 141–152, New York, USA, June 2002.

[BT07] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007.

[BYMK+06] Muli Ben-Yehuda, Jon Mason, Orran Krieger, Jimi Xenidis, Leendert Van Doorn, Asit Mallick, Jun Nakajima, and Elsie Wahlig. Utilizing IOMMUs for virtualization in Linux and Xen. In *Proceedings of the 2006 Ottawa Linux Symposium*, Ottawa, Canada, July 2006.

[Car98] David N. Card. Learning from our mistakes with defect causal analysis. *IEEE Software*, 15(1):56–63, 1998.

[CBC+92] Ram Chillarege, Inderpal S. Bhandari, Jarir K. Chaar, Michael J. Halliday, Diane S. Moebus, Bonnie K. Ray, and Man-Yuen Wong. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on Software Engineering*, 18(11):943–956, 1992.

[CBMM94] Michael Condict, Don Bolinger, Dave Mitchell, and Eamonn McManus. Microkernel modularity with integrated kernel performance. Technical report, OSF Research Institute, June 1994.

[CCJR07] Prakash Chandrasekaran, Christopher L. Conway, Joseph M. Joy, and Sriram K. Rajamani. Programming asynchronous layers with CLARITY. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pages 65–74, Dubrovnik, Croatia, 2007.

[CD94] David R. Cheriton and K. Duda. A caching model of operating system functionality. In *Proceedings of the 1st USENIX Symposium on Operating Systems Design and Implementation*, pages 14–17, Monterey, CA, USA, November 1994.

[CE04] Christopher L. Conway and Stephen A. Edwards. NDL: a domain-specific language for device drivers. In *Proceedings of the Conference on Language, Compiler and Tool Support for Embedded Systems'04*, pages 30–36, Washington, DC, USA, June 2004.

[CFH05] Andy Chou, Bryan Fulton, and Seth Hallem. Linux kernel security report, 2005.

[CGJ88] Robert P. Colwell, Edward F. Gehringer, and E. Douglas Jensen. Performance effects of architectural complexity in the Intel 432. *ACM Transactions on Computer Systems*, 6(3), 1988.

[CGJ+03] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[Che84] David R. Cheriton. The V kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, 1984.

[CJK04] Edmund Clarke, Himanshu Jain, and Daniel Kroening. Predicate abstraction and refinement techniques for verifying Verilog. Technical Report CMU-CS-04-139, Carnegie Mellon University, 2004.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly & Associates, Inc, 3rd edition, 2005.

[CYC+01] Andy Chou, Jun-Feng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 73–88, Lake Louise, Alta, Canada, October 2001.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. *ACM SIGPLAN Notices*, 35(5):35–46, 2000.

[Deu73] L. Peter Deutsch. A LISP machine with very compact programs. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, page 697, 1973.

[DF01] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[Don00] Dave Donohoe. Talking to hardware under QNX Neutrino, 2000. `http://www.qnx.com/developer/articles/nov2000a/index.html`.

[Dus88] Patrick H. Dussud. Lisp hardware architecture: the Explorer II and beyond. *SIGPLAN Lisp Pointers*, 1(6):13–18, 1988.

[DVH66] Jack B. Dennis and Earl C. Van Horn. Programming semantics for multiprogrammed computations. *Communications of the ACM*, 9:143–155, 1966.

[EAV+06] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: software guards for system address spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 75–88, Seattle, Washington, November 2006.

[ECCH00] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 1–16, San Diego, CA, October 2000.

[EDE07] Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. A memory allocation model for an embedded microkernel. In *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems*, pages 28–34, Sydney, Australia, January 2007. NICTA.

[Edv] Adam Edvardsson. SD card mass storage controller. `http://www.opencores.org`.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 251–266, Copper Mountain, CO, USA, December 1995.

[Eng72] D. M. England. Operating system of System 250. In *Proceedings of the International Switching Symposium*, June 1972.

[FAH+06] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the 1st EuroSys Conference*, pages 177–190, Leuven, Belgium, April 2006.

[FGB91] Alessandro Forin, David Golub, and Brian Bershad. An I/O system for Mach 3.0. In *Proceedings of the USENIX Mach Symposium*, pages 163–176, Monterey, CA, USA, November 1991.

[FHL+99] Brian Ford, Mike Hibler, Jay Lepreau, Roland McGrath, and Patrick Tullmann. Interface and execution models in the Fluke kernel. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*, pages 101–115, New Orleans, LA, USA, February 1999. USENIX.

[Fle97] Brett D. Fleisch. The failure of personalities to generalize. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 8–13, Cape Cod, MA, USA, May 1997.

[GE-64] GE-625/635 programming reference manual, July 1964.

[GGP06] Archana Ganapathi, Viji Ganapathi, and David Patterson. Windows XP kernel crash analysis. In *Proceedings of the 20th USENIX Large Installation System Administration Conference*, pages 101–111, Washington, DC, USA, 2006.

[GKB01] Michael Golm, Jürgen Kleinöder, and Frank Bellosa. Beyond address spaces: Flexibility, performance, protection, and resource management in the type-safe JX operating system. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, pages 3–8, Schloß Elmau, Germany, May 2001.

[GRB+08] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. The design and implementation of microdrivers. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 168–178, Seattle, WA, USA, March 2008.

[Gre] Green Hills Software. INTEGRITY real-time operating system. `http://www.ghs.com/products/rtos/integrity.html`.

[Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

[HBB+98] Hermann Härtig, Robert Baumgartl, Martin Borriss, Claude-Joachim Hamann, Michael Hohmuth, Frank Mehnert, Lars Reuther, Sebastian Schönberg, and Jean Wolter. DROPS—OS support for distributed multimedia applications. In *Proceedings of the 8th SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[HBG+06] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM Operating Systems Review*, 40(3):80–89, July 2006.

[HBG+07] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Failure resilience for device drivers. In *Proceedings of the 37th International Conference on Dependable Systems and Networks*, pages 41–50, Edinburgh, UK, June 2007.

[HBG+09] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Fault isolation for device drivers. In *Proceedings of the 39th International Conference on Dependable Systems and Networks*, Lisbon, Portugal, July 2009.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 66–77, St. Malo, France, October 1997.

[HJLT05] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew Tolmach. A principled approach to operating system construction in Haskell. In *Proceedings of the 10th International Conference on Functional Programming*, pages 116–128, Tallinn, Estonia, September 2005.

[HJM03] Thomas Henzinger, Ranjit Jhala, and Rupak Majumdar. Counterexample-guided control. In *Proceedings of the 30th International Colloquium on Automata, Languages and Programming*, pages 886–902, July 2003.

[HLM+03] Hermann Härtig, Jork Löser, Frank Mehnert, Lars Reuther, Martin Pohlack, and Alexander Warg. An I/O architecture for microkernel-based operating systems. Technical Report TUD-FI03-08, TU Dresden, July 2003.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17:549–57, 1974.

[Hoh02] Michael Hohmuth. *Pragmatic nonblocking synchronization for real-time systems*. PhD thesis, Dresden University of Technology, July 2002.

[Hol03] Gerard J. Holzmann. *The SPIN model checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[How] David Howells. Linux kernel memory barriers.

[HWF+05] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kottsovinos, and Dan Magenheimer. Are virtual machine monitors microkernels done right? In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems*, pages 1–6, Sante Fe, NM, USA, June 2005. USENIX.

[Hyp08] HyperTransport Consortium. HyperTransport I/O link specification. Revision 3.10, July 2008.

[IEE] IEEE 802.3 Ethernet working group. `http://grouper.ieee.org/groups/802/3/`.

[Inf08] InfiniBand Trade Association. InfiniBand architecture specification. Release 1.2.1, January 2008.

[Int99] Intel Corporation. Universal Host Controller Interface (UHCI) design guide. Revision 1.1, September 1999.

[Int02] Intel Corporation. Enhanced Host Controller Interface specification for Universal Serial Bus. Revision 1.0, March 2002.

[Int08] Intel Corporation. Intel virtualization technology for directed I/O, September 2008.

[JMG+02] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 275–288, June 2002.

[Joh77] Stephen Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, December 1977.

[Kee78] J. Leslie Keedy. The MONADS operating system. In *Proceedings of the 8th Australasian Computer Conference*, Canberra ACT, Australia, 1978.

[Kie02] Richard B. Kieburtz. P-logic: property verification for Haskell programs, February 2002.

[KKK07] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *Proceedings of the 2007 USENIX Annual Technical Conference*, pages 1–14, Santa Clara, CA, USA, June 2007.

[KRS09] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating hardware device failures in software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, 2009.

[KSF00] Tetsuro Katayama, Keizo Saisho, and Akira Fukuda. Prototype of the device driver generation system for UNIX-like operation systems. In *Proceedings of the International Symposium on Principles of Software Evolution*, Kanazawa, Japan, November 2000.

[LBB+91] Jochen Liedtke, Ulrich Bartling, Uwe Beyer, Dietmar Heinrichs, Rudolf Ruland, and Gyula Szalay. Two years of experience with a μ-kernel based OS. *ACM Operating Systems Review*, 25(2):51–62, April 1991.

[LCFD+05] Ben Leslie, Peter Chubb, Nicholas Fitzroy-Dale, Stefan Götz, Charles Gray, Luke Macpherson, Daniel Potts, Yueting (Rita) Shen, Kevin Elphinstone, and Gernot Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, September 2005.

[LCTSdL07] H. Andres Lagar-Cavilla, Niraj Tolia, M. Satyanarayanan, and Eyal de Lara. Vmm-independent graphics acceleration. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 33–43, San Diego, CA, USA, jun 2007.

[Lev84] Henry M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984. `http://www.cs.washington.edu/homes/levy/capabook/`.

[Lie93] Jochen Liedtke. Improving IPC by kernel design. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 175–188, Asheville, NC, USA, December 1993.

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, USA, December 1995.

[Lin] Linux IOMMU support. Linux kernel documentation, `Documentation/Intel-IOMMU.txt`.

[LMB^{+}96] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE Journal of Selected Areas in Communications*, 14(7):1280–1297, 1996.

[LN78] H. C. Lauer and R. M. Needham. On the duality of operating system structures. In *Proceedings of the 2nd International Symposium on Operating Systems*, pages 3–19, Rocquerncourt, France, October 1978.

[LOT89] LOTOS - a formal description technique based on the temporal ordering of observational behavior, 1989. ISO Standard 8807.

[LUSG04] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified device driver reuse and improved system dependability via virtual machines. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, San Francisco, CA, USA, December 2004.

[Mad96] Peter W. Madany. JavaOS: A standalone Java environment. a white paper, May 1996.

[MCZ06] Aravind Menon, Alan L. Cox, and Willy Zwaenepoel. Optimizing network virtualization in Xen. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 15–28, Boston, MA, USA, June 2006.

[Mic] Microsoft Corporation. Windows Hardware Development Central. `http://www.microsoft.com/whdc/`.

[Mic06] Microsoft Corporation. Architecture of the kernel-mode driver framework, 2006.

[Mic07] Microsoft Corporation. Architecture of the user-mode driver framework, 2007.

[MIO87] John D. Musa, Anthony Iannino, and Kazuhira Okumoto. *Software reliability: measurement, prediction, application*. McGraw-Hill, Inc., 1987.

[MJHS90] R. G. Mays, C. L. Jones, G. J. Holloway, and D. P. Studinski. Experiences with defect prevention. *IBM Systems Journal*, 29(1):4–32, 1990.

[Moo87] David A. Moon. Symbolics architecture. *IEEE Computer*, 20(1):43–52, 1987.

[MR96] Jeffrey C. Mogul and K. K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proceedings of the 1996 USENIX Annual Technical Conference*, pages 99–111, San Diego, CA, USA, January 1996.

[MRC+00] Fabrice Mérillon, Laurent Réveillère, Charles Consel, Renaud Marlet, and Gilles Muller. Devil: An IDL for hardware programming. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation*, pages 17–30, San Diego, CA, USA, October 2000.

[MST+05] Aravind Menon, Jose Renato Santos, Yoshio Turner, G. Janakiraman, and Willy Zwaenepoel. Diagnosing performance overheads in the Xen virtual machine environment. In *Proceedings of the 1st USENIX Symposium on Virtual Execution Environments*, pages 13–23, Chicago, IL, USA, 2005.

[Mur04] Brendan Murphy. Automating software failure reporting. *ACM Queue*, 2(8):42–48, November 2004.

[Nak02] Bryce Nakatani. ELJOnline: User mode drivers. `http://www.linuxdevices.com/articles/AT5731658926.html`, 2002.

[NCH+05] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3):477–526, 2005.

[Net] The Netperf benchamark suite. `http://www.netperf.org`.

[NW77] R.M. Needham and R.D.H. Walker. The Cambridge CAP computer and its protection system. In *Proceedings of the 6th ACM Symposium on Operating Systems Principles*, pages 1–10. ACM, November 1977.

[Obj09] Object Management Group. Unified Modeling Language Superstructure. Version 2.2., February 2009.

[OHC99] Open Host Controller Interface specification for USB. Release 1.0a, September 1999.

[OKL] Open Kernel Labs. `http://www.ok-labs.com`.

[OOJ98] Mattias O'Nils, Johnny Öberg, and Axel Jantsch. Grammar based modelling and synthesis of device drivers and bus interfaces. In *Proceedings of the 24th Euromicro Conference*, Vesteras, Sweden, August 1998.

[OPr] The OProfile profiler for Linux. `http://oprofile.sourceforge.net`.

[Org73] Elliott I. Organick. *Computer System Organization: The B5700/B6700 Series*. Academic Press, 1973.

[PCI02] PCI Special Interest Group. PCI-X 2.0a protocol spec, 2002.

[PCI04] PCI Special Interest Group. Conventional PCI 3.0, 2004.

[PCI07] PCI Special Interest Group. PCIe Base 2.1 specification, January 2007.

[PLHM08] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the 3rd EuroSys Conference*, pages 247–260, Glasgow, UK, April 2008.

[PLM06] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *Proceedings of the 1st EuroSys Conference*, pages 59–71, Leuven, Belgium, April 2006.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[Pro01] Project UDI. UDI core specification. Version 1.01, February 2001.

[PWW+03] Parveen Patel, Andrew Whitaker, David Wetherall, Jay Lepreau, and Tim Stack. Upgrading transport protocols using untrusted mobile code. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 1–14, Bolton Landing, NY, USA, October 2003.

[RAA+88] Marc Rozier, Vadim Abrossimov, François Armand, I. Boule, Michel Gien, Marc Guillemont, F. Herrmann, Claude Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. CHORUS distributed operating system. *Computing Systems*, 1(4):305–370, 1988.

[RBK07] Leonid Ryzhyk, Timothy Bourke, and Ihor Kuz. Reliable device drivers require well-defined protocols. In *Proceedings of the 3rd Workshop on Hot Topics in System Dependability*, Edinburgh, UK, June 2007.

[RCK+09] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic device driver synthesis with Termite. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, October 2009.

[RCKH09] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming device drivers. In *Proceedings of the 4th EuroSys Conference*, Nuremberg, Germany, April 2009.

[RDH+80] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, Willian C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. Pilot: An operating system for a personal computer. *Communications of the ACM*, 23:81–92, 1980.

[Rea05] Realtek Corp. Single-chip multifunction 10/100Mbps Ethernet controller with power management. Datasheet., 2005.

[RKH07] Leonid Ryzhyk, Ihor Kuz, and Gernot Heiser. Formalising device driver interfaces. In *Proceedings of the 4th Workshop on Programming Languages and Operating Systems*, Stevenson, Washington, USA, October 2007.

[Ros69] Robert F. Rosin. Supervisory and monitor systems. *ACM Computing Surveys*, 1(1):37–54, 1969.

[RR81] Richard F. Rashid and George G. Robertson. Accent: A communication oriented network operating system kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 64–75, 1981.

[RS09] Matthew J. Renzelmann and Michael M. Swift. Decaf: Moving device drivers to a moderm language. In *Proceedings of the 2009 USENIX Annual Technical Conference*, San Diego, CA, USA, 2009.

[SABL04] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, USA, December 2004.

[Sal65] Jerome H. Saltzer. CTSS technical notes. Technical Report MAC-TR-16, MIT, 1965.

[SBL03] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, Bolton Landing (Lake George), New York, USA, October 2003.

[SD 06] SD Card Association. SD specifications part 1: Physical layer simplified specification, version 2.00, 2006.

[SD 07] SD Card Association. SD specifications part a2: SD host controller simplified specification, version 2.00, 2007.

[SESS96] Margo I. Seltzer, Yasuhiro Endo, Christopher Small, and Keith A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 213–228, November 1996.

[SS98] Christopher Small and Margo I. Seltzer. MiSFIT: Constructing safe extensible systems. *IEEE Concurrency*, 6(3):34–41, July/September 1998.

[STJP08] Jose Renato Santos, Yoshio Turner, G. Janakiraman, and Ian Pratt. Bridging the gap between software and hardware techniques for I/O virtualization. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 29–42, Boston, MA, USA, June 2008.

[SVL01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O devices on VMware Workstation's hosted virtual machine monitor. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 1–14, Boston, MA, USA, June 2001.

[SWSS05] Alan Shieh, Dan Williams, Emin Gün Sirer, and Fred B. Schneider. Nexus: a new operating system for trustworthy computing. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 1–9, Brighton, United Kingdom, October 2005.

[SYKI05] Jun Sun, Wanghong Yuan, Mahesh Kallahalla, and Nayeem Islam. HAIL: a language for easy and correct device access. In *Proceedings of the 5th International Conference on Embedded Software*, pages 1–9, Jersey City, NJ, USA, September 2005.

[Sys03] System Architecture Group, University of Karlsruhe. The L4Ka:Pistachio microkernel. white paper., May 2003.

[SZBH86] Daniel C. Swinehart, Polle T. Zellweger, Richard J. Beach, and Robert B. Hagmann. A structural view of the Cedar programming environment. *ACM Transactions on Programming Languages and Systems*, 8:419–490, 1986.

[Tho95] Wolfgang Thomas. On the synthesis of strategies in infinite games. In *Proceedings of the 12th Symposium on Theoretical Aspects of Computer Science*, pages 1–13, 1995.

[TKRB91] Andrew S. Tanenbaum, M. Frans Kaashoek, Robert Van Renesse, and Henri E. Bal. The Amoeba distributed operating system—a status report. *Computer Communications*, 14(6):324–335, 1991.

[USB00] USB Implementers Forum, Inc. Universal Serial Bus Specification. Revision 2.0., April 2000.

[USB05] USB Implementers Forum, Inc. Wireless Universal Serial Bus specification. Revision 1.0, May 2005.

[USB08a] USB Implementers Forum, Inc. Universal Serial Bus 3.0 specifications, November 2008.

[USB08b] USB Implementers Forum, Inc. Universal Serial Bus mass storage class specification overview. Revision 1.3, September 2008.

[vBCB03] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, pages 19–24, Lihue, Hawaii, USA, May 2003.

[vdB94] Michael von der Beeck. A comparison of Statecharts variants. In *Proceedings of the 3rd International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, 1994.

[VM99] Kevin Thomas Van Maren. The Fluke device driver framework. MSc thesis, University of Utah, December 1999.

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17:337–345, 1974.

[Whe] David A. Wheeler. SLOCCount tools. `http://www.dwheeler.com/sloc/`.

[Wit08] Lea Wittie. Laddie: the language for automated device drivers. Computer Science Technical Report 08-2, Bucknell University, March 2008.

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 203–216, Asheville, NC, USA, December 1993.

[WM03] Shaojie Wang and Sharad Malik. Synthesizing operating system based device drivers in embedded systems. In *Proceedings of the 1st International Conference on Hardware/Software Codesign and System Synthesis*, pages 37–44, Newport Beach, CA, USA, October 2003.

[WMB03] Shaojie Wang, Sharad Malik, and Reinaldo A. Bergamaschi. Modeling and integration of peripheral devices in embedded systems. In *Proceedings of the 40th ACM/IEEE Conference on Design, Automation and Test in Europe*, 2003.

[WRC08] Paul Willmann, Scott Rixner, and Alan L. Cox. Protection strategies for direct access to virtualized I/O devices. In *Proceedings of the 2008 USENIX Annual Technical Conference*, pages 15–28, Boston, MA, USA, June 2008.

[WRW+08] Dan Williams, Patrick Reynolds, Kevin Walsh, Emin Gün Sirer, and Fred B. Schneider. Device driver safety through a reference validation mechanism. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 241–254, San Diego, CA, USA, December 2008.

[ZCA+06] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 45–60, Seattle, WA, USA, November 2006.

[ZZC03] Qing-Li Zhang, Ming-Yuan Zhu, and Shuo-Ying Chen. Automatic generation of device drivers. *SIGPLAN Notices*, 38(6):60–69, 2003.