Automatic Device Driver Synthesis with Termite

Leonid Ryzhyk[*†] Peter Chubb[*†] Ihor Kuz[*†] Etienne Le Sueur[*†] Gernot Heiser[*†‡]
[*]NICTA[*] [†]The University of New South Wales [‡]Open Kernel Labs
Sydney, Australia
leonid.ryzhyk@nicta.com.au

ABSTRACT

Faulty device drivers cause significant damage through down time and data loss. The problem can be mitigated by an improved driver development process that guarantees correctness by construction. We achieve this by synthesising drivers automatically from formal specifications of device interfaces, thus reducing the impact of human error on driver reliability and potentially cutting down on development costs.

We present a concrete driver synthesis approach and tool called Termite. We discuss the methodology, the technical and practical limitations of driver synthesis, and provide an evaluation of non-trivial drivers for Linux, generated using our tool. We show that the performance of the generated drivers is on par with the equivalent manually developed drivers. Furthermore, we demonstrate that device specifications can be reused across different operating systems by generating a driver for FreeBSD from the same specification as used for Linux.

Categories and Subject Descriptors

D.4.4 [**Operating Systems**]: Input/Output; B.4.2 [**Input/Output and Data Communications**]: Input/Output Devices

General Terms

Languages, Reliability, Verification

Keywords

Device Drivers, Software Synthesis, Domain-Specific Languages, Two-Player Games.

1. INTRODUCTION

Faulty device drivers are a major source of operating system failures, causing significant damage through downtime and data loss [10, 24]. A number of static [1, 3, 8] and runtime [9, 12, 15, 16, 24, 28] techniques have been proposed to detect and isolate driver faults. These techniques, however, suffer from serious limitations. Existing static analysis tools are capable of detecting a limited subset of errors, such as common OS API rule violations [1] and certain memory allocation and synchronisation errors [8]. Stronger correctness properties, such as memory safety, race-freedom, and correct use of device interfaces, are currently beyond the reach of these tools. Runtime isolation architectures are capable of detecting broader classes of errors, but do so at the cost of introducing a CPU overhead in the order of 100% [4, 9, 24].

An alternative to fault detection and isolation is an improved driver development process that guarantees correctness by construction. One way to achieve this is to synthesise device drivers automatically from a device specification, thus reducing the impact of human error on driver reliability and potentially cutting down on development costs. We have implemented a tool called Termite that does exactly that. Termite combines two formal specifications: one describing the device's registers and behaviour, and one describing the interface between the driver and the OS, to synthesise a complete driver implementation in C. With Termite, the problem of writing a correct driver is reduced to one of obtaining or developing correct specifications.

Separating device description from OS-related details is a key aspect of our approach. It allows the people with the most appropriate skills and knowledge to develop specifications: device interface specifications can be developed by device manufacturers, and OS interface 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 interface specifications are independent of any OS, so drivers for different OSes can be synthesised from a single specification developed by a device manufacturer, thus avoiding penalizing less popular OSes 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-interface 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 programming errors, including memory management and synchronisation bugs, missing return value checks, etc. A bug in a driver can occur only as a result of an error in the specification.

[*]NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

To appear in the 22nd ACM Symposium on Principles of Operating Systems (SOSP), October 2009, Big Sky, MT, USA

The likelihood of errors due to incorrect OS interface specifications is reduced because these specifications are shared by many drivers and are therefore subject to extensive testing. 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.

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, our 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 [10]. 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 interface specifications are OS independent, providing support for driver synthesis allows the reuse of all existing Termite device interface specifications, leading to potential increases in an operating system's base of compatible devices.

In this paper we make the following contributions. First, we present an approach to driver synthesis based on separate specifications of device and OS interfaces. Second, we define a formal language for specifying such interfaces. Third, we describe an algorithm based on game theory to generate drivers from the specifications. Finally, we evaluate the proposed approach based on our experience synthesising Linux and FreeBSD drivers for two real devices: a Secure Digital (SD) card host controller, and a USB-to-Ethernet adapter.

The rest of this paper is structured as follows. We start with a high-level overview of the synthesis methodology in Section 2. Section 3 describes the specification language of Termite, while Section 4 presents a sample driver specification illustrating how real device and OS interfaces are defined in this language. Section 5 outlines our driver synthesis algorithm, followed by a qualitative and quantitative evaluation of the Termite approach and the resulting drivers in Section 6. Section 7 discusses the limitations of the current Termite implementation. Finally, we survey related work in Section 8 and draw conclusions in Section 9.

2. OVERVIEW OF DRIVER SYNTHESIS

Termite generates an implementation of a driver based on a formal specification of its device and OS interfaces. The device interface specification describes the programming model of the device, including its software-visible states and behaviours. The OS interface 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 interface description is found in the device data sheet, whereas the OS interface 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 interface 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 them independent?

Our solution is inspired by conventional driver development practices. Consider, for example, the task of writing a Linux driver for the RTL8139D Ethernet controller. Linux requires all Ethernet drivers to implement the `hard_start_xmit` entrypoint described in the Linux driver developer's manual [6]:

> `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 [19], 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 1 shows a high-level view of the synthesis process. The following subsections elaborate on each of the three specifications involved in driver synthesis.

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 [13], describes common behaviours of Ethernet controller devices, including packet transmission and reception, link status detection, speed autonegotiation, etc. Other I/O protocol standards include SCSI, USB, AC'97, IEEE 1394, SD, etc.

In Termite, device-class functionality is formalised as a set of events. The Ethernet controller device class, for example, includes such events as packet transmission, completion of autonegotiation, and link status change.

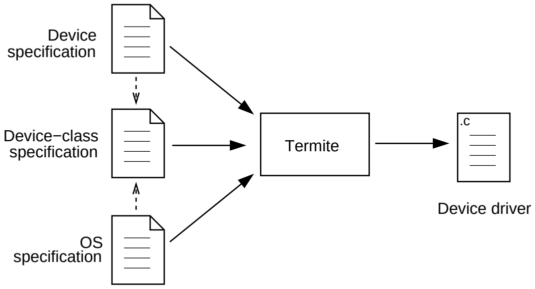


Figure 1: Driver synthesis with Termite. Solid arrows indicate inputs and outputs of the synthesis tool; dashed arrows indicate references from the device and OS specifications to the device-class specification.

Since device-class specifications must be agreed upon by all device vendors, we envisage that it can be maintained by the appropriate regulatory body as part of the corresponding I/O protocol standard.

In practice, many I/O devices are not 100% standards compliant. For example, some devices support non-standard configuration parameters accessible to applications via `ioctl()` or similar mechanisms. If such a configuration parameter is shared by several devices, it can be included in the standard device-class specification as an optional feature that need not be supported by all implementations. In case this feature is unique to the device, the device manufacturer has to develop an extended version of the device-class specification describing the given feature. The extension must be strictly incremental, so that it can be combined with an existing OS interface specification that is not aware of the new function. The device manufacturer must also develop an extended OS interface specification in order to enable access to this function in a specific OS.

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 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 3). Here, steps 1, 3, and 4 represent actions of the device interface, 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 device 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.

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

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 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 equivalent to the one in Figure 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. This order is established automatically by the synthesis algorithm. In contrast, the specification derived from the existing driver source code specifies an explicit command sequence. The Termite synthesis tool, described in this paper, 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 vision 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. In principle, such abstraction can be computed automatically, thus further reducing the effort required to create a device driver. At the moment, however, support for automatic abstraction has not been implemented, therefore it must be performed manually.

Since the RTL description is used as the source for generating the actual device circuit, it constitutes an accurate and complete

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 4: A fragment of the Ethernet controller driver interface specification.

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. We have not implemented support for such model checking yet.

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

2.3 OS specifications

The OS interface 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.

The OS interface state machine describes 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 interface between the Linux kernel and an Ethernet driver specified in Figure 4. This specification states that the driver must respond 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 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.

A typical driver interacts with the OS through several interfaces—one for each service provided or used by the driver. In particular, every driver implements at least one interface, through which the OS accesses the device functionality. Most devices are connected to the host CPU via an I/O bus, such as PCI or USB. Drivers for such devices use the bus transport service provided by the OS to access the device. In addition, the driver may use generic OS services, such as timers and memory allocators.

Termite allows each OS interface to be defined independently, in a separate specification. Multiple interfaces can then be combined in a driver declaration given to Termite.

2.4 The synthesis process

The goal of the Termite synthesis algorithm is to generate a driver implementation that complies with all relevant interface 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 enabled in the OS interface specification in the current state, i.e. the driver should only perform external actions when allowed by the OS interface. Likewise, every OS callback performed by the driver must trigger a transition in the OS interface state machine.

2. **Liveness**. The driver must be able to meet all its goals: whenever the OS interface 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.

This can be formalised as a two-player game between the driver and its environment comprised of the device and the OS. 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 interface 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 5.

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.

3. THE TERMITE SPECIFICATION LANGUAGE

This section presents the Termite specification language used to develop driver interface specifications that can be processed by the Termite synthesis tools. An example of a driver specification written in this language is presented in Section 4.

3.1 Requirements

The Termite specification language must be suitable for modelling the behaviour of complex I/O devices, containing multiple functional units. Such systems cannot be feasibly described by explicitly enumerating its states. 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

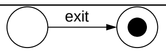
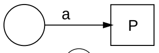
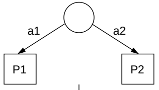
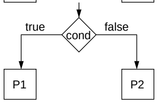
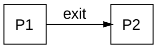
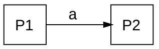
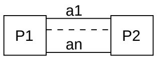
Name	Syntax	Semantics	Description
Termination	`exit`		Successful termination
Message prefixing	`a; P`		A process that performs action a and then behaves as process P
Choice	`a1; P1 [] a2; P2`		A process that performs either a1 or a2 and then behaves as P1 or P2 respectively
Conditional	`if[cond]P1[]else P2`		A process that behaves as P1 if cond holds or as P2 otherwise
Sequencing	`P1 >> P2`		Start process P2 after P1 terminates
Preemption	`P1 [> P2`		Execution of P1 is interrupted when the first action of P2 occurs
Parallel composition	`P1[|a1...an|]P2`		P1&P2 run concurrently, synchronising on actions a1..an, i.e., one of these actions can only occur when it triggers a state transition in both processes
Interleaving	`P1|||P2`		P1&P2 run concurrently; actions can arbitrarily interleave

Table 1: Termite control structures. Circles denote individual states. Squares denote entire state machines, aka processes. A circle with a dot denotes a final state.

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 present version only accepts specifications written in our own domain-specific language, presented below.

The design of the Termite specification language was inspired by the LOTOS process calculus [14] and our earlier work on the Tingu software protocol specification language [20].

3.2 Messages, interfaces, and components

In the Termite language, all interactions between the driver, the device, and the OS, including reading and writing device registers, interrupt notifications, OS requests and callbacks, are modelled as messages. There are three types of messages: inbound messages sent by the device or the OS to the driver, outbound messages sent by the driver, and internal messages that model device-class events and do not represent any directly observable interactions. A message can carry data defined by its arguments.

Messages are grouped into interfaces. A Termite interface corresponds to the informal notion of an interface used in the previous sections. Device and OS vendors define an interface for every device and for every OS service used or implemented by the driver. In addition, device-class events are grouped in a separate interface.

Data associated with the interface is modelled using variables, for example, to represent device register values.

The behaviour associated with the device or OS interface is modelled as a state machine whose transitions are triggered by occurrences of interface messages or device-class events. The latter possibility allows the specification of relative orderings of device or OS messages and device-class events. When a transition is taken, it may update the values of interface variables.

An interface declaration consists of several sections. The `types` section declares data types used by the interface. The Termite type system currently supports booleans, arbitrarily sized integers, enumerations, and structures. The `messages` section declares inbound, outbound, and internal messages associated with the interface. The `variables` section declares variables associated with the interface. The `transitions` section describes the state machine of the interface.

The top-level entity in the Termite language is the component, which describes a device driver to be synthesised by listing its interfaces. It is provided as the main input to the Termite synthesis tool.

3.3 Interface state machines

An interface state machine consists of messages combined into processes. A simple process is a sequence of messages named after its initial state. For example, the following process allows messages `m1` and `m2` to occur before returning to the initial state.

```
process FOO
  m1; m2; FOO
endproc
```

More complex processes are composed out of simple ones using sequential and parallel composition operators listed in Table 1.

An individual message occurrence in a process has the following syntax:

```
<message_name>[<guard>]/<action>:timed
```

with optional `<guard>`, `<action>` and `timed` components. Here, `guard` is a boolean expression over interface variables and message arguments, which defines conditions under which the transition is enabled. The `action` specifies how interface variables are updated when the transition is taken. The `timed` keyword is used to specify liveness requirements of the interface: when 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 message, then it is the responsibility of the driver to generate this message; otherwise, the other side of the interface (the device or the OS) guarantees that it will deliver the message to the driver.

The following specification fragment illustrates the use of the above syntax:

```
bar[$arg==m_var]/m_var=0
```

It describes an occurrence of the `bar` message with the `arg` argument equal to the value of the `m_var` interface variable ("$" denotes a message argument in the Termite language). When the transition is taken, the value of the variable is changed to 0.

The Termite specification language supports two special types of transition. An `await` transition is triggered when its guard expression evaluates to truth. A `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.

4. EXAMPLE

In this section, we illustrate 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 explain in the limited space available, yet represents a real device allowing us to show what Termite specifications for real hardware look like.

4.1 Overview

An SD host controller acts as a bridge between the host CPU and an SD card device connected to the SD bus (Figure 5). 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.

For this example we have targeted an open-source SD host controller implementation published by the OpenCores project [7]. The device interface specification presented here has been manually derived from the register-transfer level Verilog HDL design of the controller.

The top-level driver declaration is shown in Figure 6. It describes the driver by listing the interface specifications that the driver must implement. These include the SD host controller device class specification, the SD host controller driver OS interface specification, and the specification of the OpenCores SD host controller device interface. The following subsections consider each of these interfaces in detail.

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

Note that results for synthesising drivers from unabridged device specifications are presented in Section 6.

4.2 The device-class specification

As mentioned in Section 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 [21], 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.

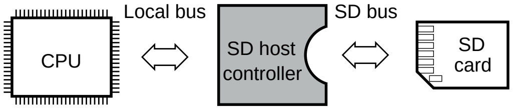


Figure 5: SD host controller device.

```
component sdhc_opencores
{
  SDHostClass class;
  SDHostOS os;
  SDHostOpenCores dev;
};
```

Figure 6: The SD host controller driver component specification.

```
 1 interface 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 messages:
18   /*Device initialized*/
19   internal on();
20   /*Device inactive*/
21   internal off();
22   /*Successful completion of a command stage*/
23   internal commandOK(
24     sdh_cmd_t command,
25     unsigned<32> response);
26   /*Command stage failed*/
27   internal commandError(
28     sdh_cmd_t command,
29     sdh_status_t status);
30   /*Successful completion of a data stage*/
31   internal blockTransferOK(
32     paddr_t mem_addr, //host address of the block
33     unsigned<32> card_addr); //card address
34   /*Data transfer failed*/
35   internal blockTransferError(
36     paddr_t mem_addr,
37     unsigned<32> card_addr,
38     sdh_status_t status);
39   /*Bus frequency changed*/
40   internal busClockChange(u32 divisor);
41 };
```

Figure 7: The SD host controller device-class specification.

Figure 7 shows the specification of the SD host controller device class. It is defined as an interface with only internal messages (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.

4.3 The OS interface specification

The OS interface specification (Figure 8) describes the service that an SD host controller driver must provide to the OS. OS re-

```
interface SDHostOS
{
types:
  struct sdhc_request_t {
    unsigned<32> opcode; /*cmd index*/
    unsigned<32> arg;    /*cmd argument*/
    bool response;       /*response present*/
    bool data_present;   /*data stage present*/
    paddr_t block;       /*block address*/
  };
  struct sdhc_response_t{
    int<32> cmd_status;   /*cmd stage status*/
    unsigned<32> response;/*response from card*/
    int<32> data_status;  /*data stage status*/
  };

messages:
  /*Probe and initialise the controller*/
  in  probe ();
  out probeComplete (int<32> status);

  /*Shut down the device and terminate the driver*/
  in  remove ();
  out removeComplete ();

  /*Issue a command on the bus, followed by a data
    transfer stage (if the command involves one)*/
  in  request (sdhc_request_t request);
  out requestComplete (sdhc_response_t response);

  /*Change the bus clock frequency*/
  in  setClock (unsigned<32> divisor);
  out setClockComplete ();

variables:
  unsigned<32> m_reqDiv;/*requested divisor*/
  sdhc_request_t m_request;
  sdhc_response_t m_response;

transitions:
  probe;
  class.on:timed;
  probeComplete[$status==0]:timed;
  REQUESTS

 where

  process REQUESTS
    /*A remove request*/
    remove;
    /*The driver must switch the device off
      before sending the completion message*/
    class.off:timed;
    removeComplete:timed;
    /*The interface state machine terminates*/
    exit
  []
    /*A setClock request*/
    setClock/m_reqDiv=$divisor;
    /*The driver must change the bus clock divisor
      to the requested value before sending the
      completion message*/
    class.busClockChange[$divisor==m_reqDiv]:timed;
    setClockComplete:timed;
    REQUESTS
  []
    /*Command without a data transfer stage*/
    request[$request.data_present==false]
           /m_request=$request;
    (
      class.commandOK
        [($command.index==m_request.opcode)&&
         ($command.arg==m_request.arg)&&
         ($command.response==m_request.response)&&
         ($command.data==false)]
        /{m_response.cmd_status=0;
          m_response.response=$response;}:timed;
      requestComplete[$response==m_response]:timed;
      REQUESTS
    []
      class.commandError
        /{m_response.cmd_status=$status;
          m_response.response=0;}:timed;
      requestComplete[$response==m_response]:timed;
      REQUESTS
    )
  []
    /*Command 17 (block read request) and command 24
      (block write request) are handled similarly*/
    request[($request.data_present==true)&&
            (($request.opcode==17)||
             ($request.opcode==24))]
           /m_request=$request;
    (
      /*Command stage completes successfully*/
      class.commandOK
        [($command.index==m_request.opcode)&&
         ($command.arg==m_request.arg)&&
         ($command.response==m_request.response)&&
         ($command.data==true)]
        /{m_response.cmd_status=0;
          m_response.response=$response;}:timed;
      (
        /*Data transfer stage completes successfully*/
        class.blockTransferOK
            [$mem_addr==m_request.block]
            /m_response.data_status=0 : timed;
        requestComplete[$response==m_response]:timed;
        REQUESTS
      []
        /*Data transfer fails*/
        class.blockTransferError
            /m_response.data_status=$status;
        requestComplete[$response==m_response]:timed;
        REQUESTS
      )
    []
      /*Command stage fails*/
      class.commandError
        /{m_response.cmd_status=$status;
          m_response.response=0;
          m_response.data_status=0;};
      requestComplete[$response==m_response]:timed;
      REQUESTS
    )
  endproc
};
```

Figure 8: The SD host controller driver OS interface specification.

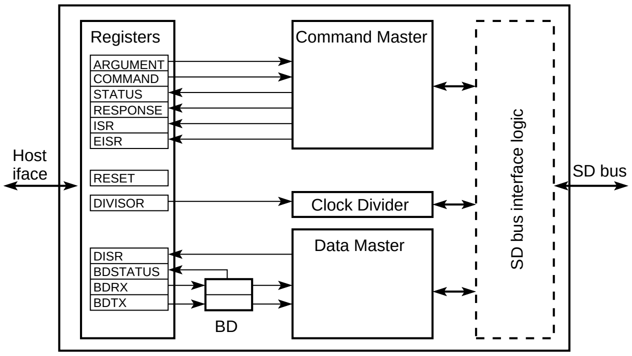


Figure 9: The OpenCores SD host controller device architecture.

quests are modelled as inbound messages; driver responses are modelled as outbound messages, as defined in lines 17–33.

The main part of the specification is the interface state machine, which 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 41–43, which specify how the driver must handle a `probe` request from the OS. Before replying to this request in line 43, the driver must ensure that the `class.on` message in line 42 occurs. This message refers to the `on` event defined in the device-class specification (Section 4.2). In other words, the precondition for sending the `probeComplete` message 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 interface state machine executes the `REQUESTS` process (line 48). In its initial state, this process performs a choice between incoming requests defined in lines 50, 59, 68, and 90 using the `[]` operator (lines 57, 66, and 87). This means that the driver must wait for one of these messages from the OS.

We consider one of the four requests in detail in order to illustrate the use of interface variables and transition guards in OS specifications. Line 67 describes a request to issue an SD command without a data transfer stage. 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 71) or the command completes with an error (line 81). The guard in lines 72–75 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 77) and sent to the OS in a `requestComplete` message (line 78). If the command fails, the driver stores the error code in the `m_response` variable (line 82) and reports the failure to the OS via a `requestComplete` message (line 84).

Handling of the other requests is explained using comments in Figure 8. Note that for a command with a data stage the driver must also wait for the data transfer to complete before signalling success.

4.4 The device interface specification

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

Figure 9 shows the internal architecture of the device in question, as defined in its HDL specification and Table 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.

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 2: SD host controller registers.

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 10. Ellipses are used throughout the specification to indicate omission of code fragments; the complete specification consists of 468 lines of code.

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

Interface variables (lines 23–30) describe internal device registers and signals that influence the device's software-observable behavior. The *XXX*`_reg` variables model device register content. The `m_new_command` variable models the internal signal that no-

```
interface SDHostOpenCores
{
types:
  /* device registers */
  struct command_reg {
    unsigned<2> RTS;
    unsigned<8> RESERVED;
    unsigned<6> CMDI;
  };
  ...
  struct block_descr {
    unsigned<32> mem_addr;  /*memory address*/
    unsigned<32> card_addr; /*card address*/
  };

messages:
  /*register read/write messages */
  out write_command_reg (command_reg v);
  out read_command_reg (out command_reg v);
  ...
  in irq ();

variables:
  command_reg m_command_reg;
  ...
  unsigned<1> m_new_command;
  unsigned<1> m_data_command;
  sdhost_command_t m_command;
  block_descr m_tx_descr;
  block_descr m_rx_descr;

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

  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
  []
    ...
endproc

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
  ...
endproc
```

Figure 10: The OpenCores SD host controller device specification.

tifies 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 interface state machine has been manually derived from the register-transfer-level (RTL) design of the device in the Verilog HDL which is used to synthesize the device hardware. The structure of the state machine reflects the device architecture shown in Figure 9 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 interface 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 33–36 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 interface variables to be known in any state. This white-box assumption does not hold in the initial state of the device interface, since at this point the device registers may have arbitrary values. The 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 34–35).

Once the reset is complete, the device is ready to handle commands, as indicated by the `on` device-class event in line 37. Line 38 invokes the `SDHOST` process (line 42), which describes normal operation of the device. This process consists of four concurrent subprocesses, corresponding to four device modules in Figure 9: `REGISTERS`, `COMMAND_MASTER`, `DATA_MASTER`, and `CLOCK_DIVIDER` (lines 43–49). 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 50 specifies that writing 1 to the reset register (line 51) interrupts normal device operation and resets all registers to their default values. A subsequent writing of 0 to this register (line 54) 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 66) 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 69), a write to this register updates the register value (line 73), but does not trigger any signals. A write to the argument register when the flag is not set (line 73) triggers the `m_new_command` signal (line 75) and sets the command inhibit flag (line 77).

The `m_new_command` signal wakes up the Command Master waiting for this signal in line 84. It uses values in the `COMMAND` and `ARGUMENT` registers to form an SD command (lines 85–87) and sends it over the bus. Upon completion of the command, it raises an interrupt (line 90). 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 white-box assumption is violated, since the exact device state is not known to the software until it reads the values of these registers. Therefore, the device interface state machine specifies a sequence of register reads required to restore the white-box invariant (lines 91–93). Lines 94–95 acknowledge the interrupt by setting the interrupt status registers to zero. Lines 96–111 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.

4.5 The driver state machine

The Termite synthesis algorithm combines the device and the OS interface 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 5. Figure 11 shows a fragment of the resulting state machine responsible for handling SD commands without a data stage.

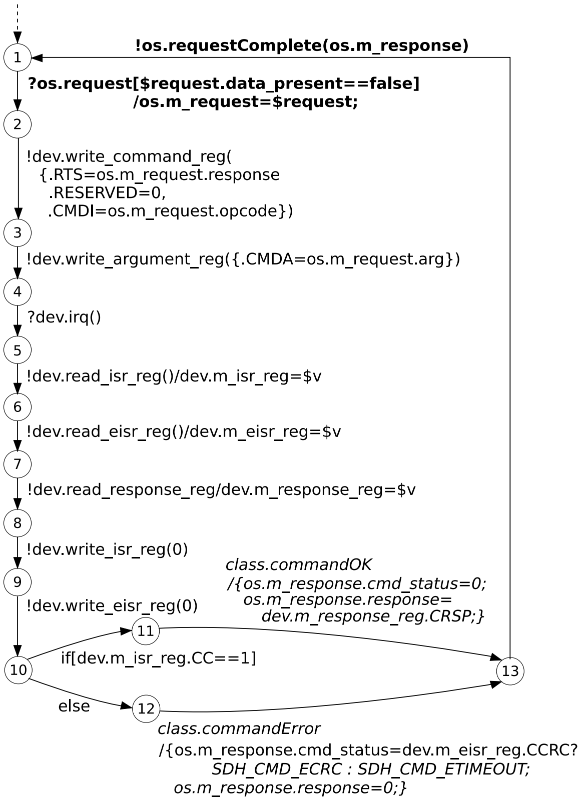

!os.requestComplete(os.m_response)
1
?os.request[$request.data_present==false]
/os.m_request=$request;
2
!dev.write_command_reg(
{.RTS=os.m_request.response
.RESERVED=0,
.CMDI=os.m_request.opcode})
3
!dev.write_argument_reg({.CMDA=os.m_request.arg})
4
?dev.irq()
5
!dev.read_isr_reg()/dev.m_isr_reg=$v
6
!dev.read_eisr_reg()/dev.m_eisr_reg=$v
7
!dev.read_response_reg/dev.m_response_reg=$v
8
!dev.write_isr_reg(0)
9
!dev.write_eisr_reg(0)
class.commandOK
/{os.m_response.cmd_status=0;
os.m_response.response=
dev.m_response_reg.CRSP;}
11
10
if[dev.m_isr_reg.CC==1]
13
else
12
class.commandError
/{os.m_response.cmd_status=dev.m_eisr_reg.CCRC?
SDH_CMD_ECRC : SDH_CMD_ETIMEOUT;
os.m_response.response=0;}

Figure 11: A fragment of the SD host controller driver state machine generated by Termite. Exclamation marks denote messages sent by the driver; question marks denote incoming messages from the device or the OS. OS interface messages are shown in bold font; device interface messages are in normal font; device-class events are in italics.

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 interface specification, either the `commandOK` or the `commandError` device-class event must occur before the driver can send a completion message to the OS (Figure 8, lines 71 and 81). 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 $8 \rightarrow 9$), 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 interface are set to reflect the status of the command. Finally, the driver sends a completion message to the OS (transition $13 \rightarrow 1$).

5. THE SYNTHESIS ALGORITHM

In this section we outline the Termite algorithm, which, given a specification of driver interfaces, generates an implementation of the driver in C, satisfying safety and liveness criteria introduced in Section 2.4.

Conceptually, the algorithm proceeds in three steps (the actual implementation involves performance optimisations, discussed below, that make the algorithm iterative). The first step combines individual driver interface specifications into a single specification. The second step produces a driver state machine that has both safety and liveness properties. The third step translates this state machine into a driver implementation in C.

5.1 Computing an aggregate specification

During the first step, the synthesis algorithm computes a state machine that combines constraints imposed by both driver interface state machines. This aggregate state machine is computed as the product of constituent interface state machines, synchronised on device-class events. The state space of the product is a subset of the Cartesian product of the multiplier state spaces. The product is constructed according to the following rules:

$$1.\quad \frac{s_1 \xrightarrow{m_1[g_1]/a_1} s_1'}{\langle s_1, s_2\rangle \xrightarrow{m_1[g_1]/a_1} \langle s_1', s_2\rangle} \qquad 2.\quad \frac{s_2 \xrightarrow{m_2[g_2]/a_2} s_2'}{\langle s_1, s_2\rangle \xrightarrow{m_2[g_2]/a_2} \langle s_1, s_2'\rangle}$$

$$3.\quad \frac{s_1 \xrightarrow{e[g_1]/a_1} s_1' \quad s_2 \xrightarrow{e[g_2]/a_2} s_2'}{\langle s_1, s_2\rangle \xrightarrow{e[g_1 \wedge g_2]/\{a1;a2;\}} \langle s_1', s_2'\rangle}$$

where s_1 and s_2 are states of the device and the OS interfaces state machine respectively, m_1 and m_2 are interface messages, e is a device class event, g_1 and g_2 are transition guards, and a_1 and a_2 are actions associated with transitions.

Rule 1 states that if a device interface message triggers a transition from state s_1 to state s_1' in the device state machine (expression above the horizontal line), then a corresponding transition must be added to the product state machine (expression below the line). Rule 2 is analogous for the OS interface message. Rule 3 states that if a device-class event triggers state transitions in both device and OS interfaces, then it also triggers a transition in the product state machine; the guard of the new transition is the conjunction of the original guards and its action is the concatenation of the original actions.

The resulting state machine describes all legal driver behaviours. It only contains transitions permitted by the original interface state machines and hence satisfies the safety requirement. It does not, however, guarantee liveness. In particular, it may enter a state where some event is required to happen eventually, but not all execution traces starting in this state contain this event. A nonconforming trace either deadlocks after a finite number of transitions or has an infinite loop, which does not include the required event.

5.2 Computing the strategy

The second step of the synthesis algorithm restricts the product state machine, eliminating such invalid traces and enforcing liveness. To this end, it solves the following problem in each reachable state of the product state machine: given a target subset of states, determine the action that the driver should take in the current state in order to reach the target in a finite number of steps, assuming that the device and the OS comply with their interface specifications. The target set of states is computed based on the goals defined in the OS interface specification using `timed` transitions. Note that the product state machine may have more than one goal in a state, e.g., when handling multiple concurrent requests from the OS. To ensure that the resulting driver implementation correctly handles arbitrary request interleavings, the target set includes states where all of these goals are satisfied.

This problem can be formalised as a two-player reachability game problem. A basic algorithm for solving such games is described by Thomas [25]. Given an origin state and a set of goal states, the algorithm recursively computes all states from which the goal can be reached in one step. This includes states where there exists at least one send transition leading to the goal (i.e., the driver can perform an action that will take it to the goal) as well as states where all enabled receive transitions lead to the goal (i.e., any message 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.

The reachability graph determines the action that the driver must take in every state on its way to the goal. This action can be either sending of a specific message to the device or the OS or waiting for an incoming message. In the former case, exactly one send transition is selected in the corresponding state of the product state machine; the remaining send and receive transitions are eliminated. In the latter case the driver must be prepared to handle any of the possible inputs; therefore the algorithm eliminates all send transitions, but keeps all receive transitions in the state.

The state machine constructed using the above procedure satisfies both safety and liveness requirements. Moreover, it specifies exactly how the driver should behave in every state, i.e., whether it should send a specific message or wait for a message from the device or the OS.

5.3 Generating code

The last step of the synthesis process is translating the driver state machine 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 message. The state structure contains a field that represents the state of the driver state machine and an extra field for every variable of the device and the OS interface.

The implementation of a driver entrypoint is generated as follows. It first checks the values of state variables to determine the driver state and identify the state transition that corresponds to the given incoming message. It then updates state variables as defined by the transition. If the target state of the transition is a send state, the driver performs the send operation by invoking the appropriate OS entry point. Otherwise, the driver returns from the entry point to wait for the next incoming message.

The resulting implementation is single threaded: every operation performed by the driver corresponds to a transition in the state machine and must occur atomically to preserve the state machine semantics. In most operating systems, however, drivers must handle invocations from multiple concurrent threads, which violates the atomicity assumption. The two models can be reconciled by either adding support for synchronisation in the synthesised code or changing the OS interface to invoke the driver atomically. We have implemented the latter approach by providing a thin wrapper that performs the translation between the multi-threaded interface supported by the OS and the message-based interface implemented by the driver. Our architecture is similar to the one described in our previous work [20].

5.4 Performance optimisations

This basic algorithm requires some enhancements to achieve satisfactory performance when synthesising real drivers. First, computing a product of interface state machines can lead to a state explosion, since the size of the product is approximately equal to the product of the multiplier sizes. We observe that, while the product state space can be enormous, the state space of the computed driver state machine is always much smaller. We exploit this fact using a lazy state-space exploration technique. The synthesis algorithm computes states of the product on demand, as they are encountered during the strategy construction. This way only a small subset of the complete product state space needs to be explored.

Second, it is computationally infeasible to represent values of variables explicitly during synthesis. For example, modelling all possible values of a single 32-bit pointer would increase the state space by a factor of 2^{32}. Fortunately, in most cases only relations among variables are important, as opposed to their actual values. As an example, when passing a data pointer to a device, the specific value of the pointer is less important than the fact that it is the same as one obtained earlier from the OS.

Therefore, Termite manipulates variables and relations among them symbolically. Internally, the synthesis algorithm represents states of the interface state machines and the product state machine in terms of arithmetic constraints on state variables. These constraints are derived from transition guards and actions. Such a symbolic state describes a set of concrete states, permitting compact representation and efficient manipulation of the state machine.

Our symbolic engine currently handles constraints of the form `(v==C)` and `(v1==v2)` (where `v`, `v1`, and `v2` are variables, and `C` is a constant) and their arbitrary boolean combinations. When Termite encounters a statement that is not expressible via such constraints, e.g., `(v1=v2+v3)`, it assumes that nothing is known about the value of `v1` after execution of the statement. This assumption is conservative: if a winning 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, if a transition leading to the goal has a guard that depends on the value of variable `v1`, this transition 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 will be addressed in the future.

6. EVALUATION

In this section we report on our 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 OSes.

6.1 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 4) and a driver for the ASIX AX88772 100Mb/s USB-to-Ethernet adapter. 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). Since the two devices belong to 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 [22]. This description provided sufficient information to derive a Termite model of the controller interface.

	R5C822	**AX88772**
Native Linux driver	1174	1200
Device interface	653	463
OS interface (SD/Ethernet)	378	213
Bus interface (PCI/USB)	263	96
Synthesised driver	4667	2620

Table 3: Size in lines of code, excluding comments, of the R5C822 and AX88772 driver implementations in Linux, their Termite specifications, and the synthesised drivers.

The AX88772 data sheet 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. 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 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 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, we 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 messages: `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.

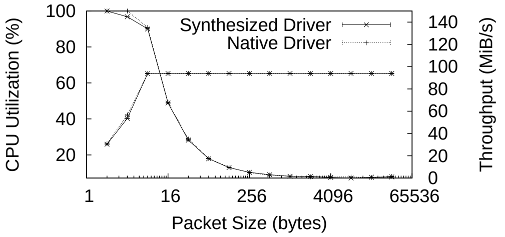


Figure 12: AX88772 TCP throughput benchmark results. The ascending line shows achieved throughput; the descending line shows CPU utilisation.

6.2 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 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 12 compares throughput and CPU utilisation achieved by the synthesised and native drivers under the Netperf TCP_STREAM benchmark. Both drivers showed virtually identical performance even 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.

6.3 Reusing device specifications

In order to validate the claim that device specifications can be reused across different OSes, 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 control driver interface and the PCI bus transport interface. These interfaces differ from their Linux counterparts in a number of aspects, including SD command format, driver initialisation, PCI resource allocation, bus power management, and DMA descriptor allocation. Once these interfaces 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 specification.

7. LIMITATIONS AND FUTURE WORK

Our experience with Termite has demonstrated the feasibility of driver synthesis for real devices. This section summaries 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 interface. 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 our ongoing research.

The synthesis algorithm Several limitations of the current synthesis algorithm complicate modelling. These include the white-box assumption described in Section 4.4, and the lack of support for complex constraints on variables in the symbolic execution engine (Section 5.4).

In addition, Termite currently only allows the manipulation of memory buffers via calls to C functions (Section 6.1). In order to reduce the reliance on manually written code, we are working on adding support for the specification and synthesis of constraints on the memory buffer layout (fragmentation, alignment, etc.) and content (headers, paddings, etc.). 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 Termite 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 will be added as part of the buffer management extension described in the previous paragraph.

Finally, we are working on further improving the synthesis algorithm 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 [11].

Code generation The Termite C code generator requires improvement to reduce the size of the generated code and make it more human-readable. Besides this, it currently produces single-threaded driver code, which requires special run-time support in the OS (see Section 5.3). Extending the code generator to synthesise drivers that can handle concurrency and synchronisation will reduce the complexity of the infrastructure required by Termite.

Debugging support Debugging automatically generated code is notoriously difficult. Fortunately, source-level debugging is rarely necessary for Termite drivers. Along with the C driver implementation, Termite also outputs the state machine of the driver, similar to the one shown in Figure 11, 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 device or the OS does not behave in the way the driver expects it to and trace these situations back to errors in the interface specifications. Future work on Termite includes developing tools for interactive state-machine debugging.

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.

8. RELATED WORK

Most previous work on driver synthesis has been done in the context of hardware/software co-design for embedded microcontrollers [18]. These devices are different than those targeted by Termite as they have a simple internal structure and a small number of input and output signals. Furthermore, the synthesised driver runs without an OS or with only a small RTOS. As such, these approaches do not tackle most of the issues addressed here, including separation of device and OS specifications and dealing with large state spaces. On the other hand, driving an embedded microcontroller often requires ensuring precise timing behaviour, which has been the focus of much of the research in the area [26]. Since the devices that we are concerned with are designed to work with a time-shared OS, strict real-time guarantees are not required.

Wang et al. [27] describe a tool for more complex driver synthesis for embedded devices. This approach does not separate OS and device interfaces of the driver, forcing the driver designer to specify the complete driver behaviour for every device. In addition, data handling is based on the assumption that driver functionality can be split into non-overlapping control and data parts. While this is the case for some simpler drivers, generally speaking, the control and data path of a driver are tightly interleaved.

Device description languages such as Devil [17], NDL [5], and HAIL [23], allow synthesis of low-level hardware accessor functions based on a specification of device register and memory layouts. This work is complementary to ours. While Termite does not currently support domain-specific data types for describing hardware register layouts, extending the Termite specification language with Devil-style constructs would close this gap.

In recent work, Chipounov and Candea [2] have synthesised device drivers by automatically reverse engineering execution traces of an existing driver for a different OS. The ability of this solution to synthesise a complete driver, functionally equivalent to the original, has not yet been demonstrated. So far, the focus of this research has been on extracting a device specification from an existing driver. The resulting specification could, in principle, be used as input to our synthesis engine, which points to an interesting synergy between the two approaches.

9. CONCLUSIONS

Device driver synthesis is a promising approach to solving the driver reliability problem. In this paper we have demonstrated the feasibility of this approach by describing a driver synthesis methodology and its implementation. The ultimate goal of our 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.

10. ACKNOWLEDGEMENTS

We would like to thank Franck Cassez, Scott Hahn, John Keys, and Mona Vij for their insightful feedback on the Termite architecture. We would like to thank Andrew Baumann, Anton Burtsev, Aaron Carroll, Nicholas FitzRoy-Dale, Timothy Roscoe, Thomas Sewell, our shepherd Michael Swift, and the anonymous reviewers for comments on earlier versions of this paper. Finally, we would like to thank Balachandra Mirla for helping develop the model of the SD controller described in Section 4.

11. REFERENCES

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

[2] V. Chipounov and G. Candea. Reverse-engineering drivers for safety and portability. In *4th HotDep*, San Diego, CA, USA, Dec 2008.

[3] A. Chou, B. Fulton, and S. Hallem. Linux kernel security report, 2005.

[4] M. Condict, D. Bolinger, D. Mitchell, and E. McManus. Microkernel modularity with integrated kernel performance. Technical report, OSF Research Institute, Jun 1994.

[5] C. L. Conway and S. A. Edwards. NDL: a domain-specific language for device drivers. In *LCTES'04*, pages 30–36, Washington, DC, USA, Jun 2004.

[6] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*. O'Reilly & Associates, Inc, 3rd edition, 2005.

[7] A. Edvardsson. SD card mass storage controller. `http://www.opencores.org`.

[8] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *4th OSDI*, pages 1–16, San Diego, CA, Oct 2000.

[9] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula. XFI: software guards for system address spaces. In *7th OSDI*, pages 75–88, Seattle, Washington, Nov 2006.

[10] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *20th LISA*, pages 101–111, Washington, DC, USA, 2006.

[11] T. Henzinger, R. Jhala, and R. Majumdar. Counterexample-guided control. In *30th ICALP*, pages 886–902, Jul 2003.

[12] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: A highly reliable, self-repairing operating system. *ACM Operat. Syst. Rev.*, 40(3):80–89, Jul 2006.

[13] IEEE 802.3 Ethernet working group. `http://grouper.ieee.org/groups/802/3/`.

[14] LOTOS - a formal description technique based on the temporal ordering of observational behavior, 1989. ISO Standard 8807.

[15] B. Leslie, P. Chubb, N. Fitzroy-Dale, S. Götz, C. Gray, L. Macpherson, D. Potts, Y. R. Shen, K. Elphinstone, and G. Heiser. User-level device drivers: Achieved performance. *Journal of Computer Science and Technology*, 20(5):654–664, Sep 2005.

[16] J. Liedtke, U. Bartling, U. Beyer, D. Heinrichs, R. Ruland, and G. Szalay. Two years of experience with a μ-kernel based OS. *ACM Operat. Syst. Rev.*, 25(2):51–62, Apr 1991.

[17] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *4th OSDI*, pages 17–30, San Diego, CA, USA, Oct 2000.

[18] M. O'Nils, J. Öberg, and A. Jantsch. Grammar based modelling and synthesis of device drivers and bus interfaces. In *24th Euromicro Conf.*, Washington, DC, USA, 1998.

[19] Realtek Corp. Single-chip multifunction 10/100mbps Ethernet controller with power management. datasheet., 2005.

[20] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *4th EuroSys Conf.*, Nuremberg, Germany, Apr 2009.

[21] SD Card Association. SD specifications part 1: Physical layer simplified specification, version 2.00, 2006.

[22] SD Card Association. SD specifications part a2: SD host controller simplified specification, version 2.00, 2007.

[23] J. Sun, W. Yuan, M. Kallahalla, and N. Islam. HAIL: a language for easy and correct device access. In *5th EMSOFT*, pages 1–9, Jersey City, NJ, USA, Sep 2005.

[24] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. In *19th SOSP*, Bolton Landing (Lake George), New York, USA, Oct 2003.

[25] W. Thomas. On the synthesis of strategies in infinite games. In *12th STACS*, pages 1–13, 1995.

[26] E. Walkup and G. Borriello. Automatic synthesis of device drivers for hardware/software co-design. Technical Report 94-06-04, University of Washington, Aug 1994.

[27] S. Wang, S. Malik, and R. A. Bergamaschi. Modeling and integration of peripheral devices in embedded systems. In *40th DATE*, 2003.

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