Checking the Hardware-Software Interface in Spec#

Kevin Bierhoff
Carnegie Mellon University
http://www.cs.cmu.edu/~kbierhof/

Chris Hawblitzel
Microsoft Research
http://research.microsoft.com/~chrishaw/

ABSTRACT

Research operating systems are often written in type-safe, high-level languages. These languages perform automatic static and dynamic checks to give basic assurances about run-time behavior. Yet such operating systems still rely on unsafe, low-level code to communicate with hardware, with little or no automated checking of the correctness of the hardware-software interaction. This paper describes experience using the Spec# language and Boogie verifier to statically specify and statically verify the safety of a driver's interaction with a network interface, including the safety of DMA.

1. INTRODUCTION

Once upon a time, programmers wrote operating systems in assembly language. As operating systems grew and compilers improved, OS programmers moved from assembly language to C and C++, which provide higher-level abstractions and more static checking than assembly language. The past 15 years have seen OS code written in Modula-3 [3], Java [10], C# [9], ML [4], and even Haskell [11].

We can map this progress using Cardelli's classification [5], shown in Table 1. The trend is towards languages that are safer and more statically typed than assembly language. In our own work on the Singularity operating system [9], for example, we have enthusiastically embraced static typing and safety in nearly every aspect of the system, from the kernel to the device drivers to the applications.

One aspect of OS programming has made relatively little progress, though: the operating system's communication with the hardware almost always consists of unverifiable sequences of low-level operations on IO memory and hardware registers. Consider Singularity's driver for the DEC Tulip network interface [7]. This driver is written entirely in C# code, using only safe, high-level C# language constructs. Nevertheless, some of the C# code looks suspiciously unsafe and low-level. For instance, this code disables packet transmission (`ST`) and reception (`SR`), and then sets the addresses of the receive and transmit DMA queues:

```
uint mode = csr6.Read32();
mode &= ~(CSR6.SR | CSR6.ST);
csr6.Write32(mode);
rxRing.Reset();
csr3.Write32(rxRing.BaseAddress.ToUInt32());
txRing.Reset();
csr4.Write32(txRing.BaseAddress.ToUInt32());
```

In this code, `csr3`, `csr4`, and `csr6` are Tulip control/status registers. Each register has a particular meaning to the Tulip hardware. The `csr6` register, for example, contains bits that control the network interface's current mode of operation. The `csr3` register contains the physical address of the receive queue. If the driver accidentally used `csr3` in place of `csr6` and `csr6` in place of `csr3`, then the network card would use the mode bits as a physical DMA address and an address as mode bits, causing unpredictable and unsafe behavior. Since `csr3` and `csr6` have the same C# type, the C# type checker would not catch this mistake at compile time. Nor would C#'s run-time system catch the mistake at run time. Thus, in Cardelli's classification, the code shown above fits more closely in the "unsafe, untyped" category with assembly language than in the "safe, typed" category with C#, ML, and Java.

Safe, typed interaction with hardware is challenging for several reasons:

- **Arithmetic**. In the example code above, the meaning of `csr6.Write32(mode)` varies dramatically depending on which bits are set in `mode`. More generally, words sent to the device may have integer bit fields with constraints on the allowed integer values. For example, the Tulip device's DMA queues contain bit fields for the lengths of the buffers in the queues. To avoid buffer overflow, these length bit fields must be no larger than the actual lengths of the buffers. Such arithmetic constraints aren't difficult to check at run time, but are beyond the abilities of most static type checkers.
- **State.** Reasoning about the correctness of a device driver requires reasoning about the state of the device. For example, a driver must properly establish valid transmit and receive queues (in `csr3` and `csr4`) before enabling packet transmission and packet reception (the `ST` and `SR` bits of `csr6`). This is difficult to verify at run-time; even if a run-time assertion walked the entire

Table 1. Types and Safety (from Cardelli [5])

	Typed	Untyped
Safe	ML, Java	LISP
Unsafe	C	Assembler

queues to check that each entry in each queue was valid, another thread could concurrently modify the queues to destroy this validity. Compile-time verification is more practical, but requires careful tracking of aliasing, and a consideration for how the device and driver interact concurrently.

- **Performance**. Safety should not impose a large performance penalty. For example, for performance's sake, Singularity's network drivers are designed to be zero copy, so that the driver passes the network stack's buffers directly to and from the DMA queues without copying the data. To ensure safety for a zero-copy implementation, the network stack must not be allowed to deallocate a buffer while the device is performing DMA to the buffer.
- **Hardware diversity**. Each device defines its own interface to the software. These interfaces are sometimes quirky and counterintuitive. (A write of "1" to the Tulip `csr5` register means "set to 0", for example.) Operations that are safe for one device may be unsafe for another. This makes it impossible to provide a universal safe hardware interface suitable for all drivers. Operating systems and programming languages cannot eliminate the diversity of hardware-software interfaces, but good software engineering can manage the problem.

This paper advocates an incremental engineering approach to structuring drivers: isolate the hardware-software interface code in a per-device safe hardware interaction layer (SHIL) that is independent of the rest of the driver. Each device SHIL should consist of a small collection of trivial primitive operations (e.g. "write the receive queue address to `csr3`"). Each operation must specify pre-conditions and post-conditions that describe the device state and ensure the safety of the operation. Therefore, if the device SHIL is written correctly, then the rest of the driver cannot cause the device to behave unsafely. Of course, the device SHIL could be incorrect, but in practice it's easier to find mistakes in a small, self-contained SHIL than in a large driver.

The SHIL approach applies incrementally to existing drivers. A programmer can start by just declaring an entire existing driver to be a SHIL, and then gradually move code out of the SHIL. At the furthest extreme, a SHIL could consist of just two functions, one to write data at an address and one to read data from an address, as long as these functions contain pre-conditions and post-conditions detailed enough to describe every possible safe write and read operations on device registers and DMA memory.

We have applied the SHIL approach incrementally to an existing Singularity Ethernet adaptor driver, using the Spec# language to express pre-conditions, post-conditions, and invariants, and the Boogie static verifier to check that the pre-conditions, post-conditions, and invariants hold. In particular, the key operations are encapsulated in SHIL methods: starting and stopping transmission and reception, enqueuing and dequeuing DMA buffers, handling interrupts, and polling the device status. Some of the SHIL methods are larger than we would like, but they are still small enough for us to express and statically verify many interesting properties about the driver, including the following:

- All integer values passed to the SHIL are within allowed ranges. (Checking this property revealed an out-of-allowed-range integer value bug in the original driver.)
- Packets are enqueued only when queues aren't full, and dequeued only when queues aren't empty.
- SHIL methods are only invoked when the device is in a state appropriate to the method.
- The driver releases control of any buffers passed to the SHIL, and does not regain access to the buffer until the SHIL determines that the device is no longer using the buffer.

2. SINGULARITY, SPEC#, AND BOOGIE

The Singularity OS runs applications, services, and drivers written in the Sing# programming language [9]. Sing# is an extension of Spec# [2], which in turn extends C#. For this paper, the most relevent extensions to C# are:

- **Pre-conditions and post-conditions**. Spec# methods may declare `requires` and `ensures` clauses that specify pre-conditions and post-conditions about the arguments. For example:

  ```
  int SquareRoot(int n)
      requires n >= 0;
      ensures Square(result) <= n;
      ensures n < Square(result + 1);
  {...method body...}
  ```

  Spec# pre-conditions and post-conditions may also refer to mutable fields of objects (including the `this` object). We use this to require and ensure properties about the state of SHIL objects, which in turn reflect the state of the device.
- **Object invariants**. Spec# classes may declare invariants on fields. An object's constructor initializes the fields to establish the invariant. After construction, methods may temporarily "expose" an object to break and reestablish the object's invariants.
- **Linear types**. Sing# supports linear types, which restrict aliasing in order to enable static reasoning about state and ownership. For example, a Sing# program can safely deallocate a linear object, because the language guarantees that no aliases to the object exist elsewhere in the program. Although Singularity disallows shared data between processes, a Singularity process may transfer ownership of linear data structures to another process. For example, the Singularity network stack exchanges linear data buffers with the network drivers to avoid copying data between the network stack process and driver process.

The Sing# compiler (built on top of the Spec# compiler) performs both standard C# type checking and linearity checking. A separate tool called Boogie checks Spec#'s pre-conditions, post-conditions, and invariants. Type checking and linearity checking are simple, decidable problems for the Sing# compiler. Boogie, on the other hand, must generate verification conditions and pass these to a separate automated theorem prover. This is more difficult, because Spec# pre-conditions, post-conditions and invariants are arbitrary first-order logical formulas that may contain arithmetic expressions and universal and existential quantifiers. In theory, no automatic theorem prover can always decide whether these formulas are valid. In our experience, the automatic theorem prover was able to verify all verification conditions for our driver in about 15 minutes. This is much slower than type checking, but much faster than interactive (not completely automatic) theorem proving. For properties whose static verification is too onerous, it's often possible to insert an explicit run-time check into the program (much like a run-time cast in ordinary C# and Java). Although nearly all of our verification was static, we did add a handful of small run-time checks to ease the static checking and overcome limitations in the theorem prover.

3. A TULIP SHIL

We chose Singularity's DEC Tulip driver as a case study for the SHIL approach, because the Tulip device is well documented [7], widely known, widely cloned, and has non-trivial requirements for safe usage. In particular, the DMA protocol requires careful coordination between the driver and the device. The Tulip device uses two rings, both stored in the host's physical memory. Each ring may be implemented as either an array or a linked list, although our SHIL currently only supports the array implementation. The first ring holds a queue of packets to be transmitted to the network, and the second ring holds a queue of packets received from the network. Figure 1 depicts the receive ring (the transmit ring is nearly identical). The device's `csr3` register points to the base of the receive ring, which is an array of entries, where each entry contains four 32-bit words. The last two words of the entry contain physical addresses of up to two data buffers. The second word contains the size of each buffer (or 0 for an unused buffer), plus some flags (including an "end-of-ring" flag that marks the last element of the array). The first word contains an ownership bit, plus status flags set by the device.

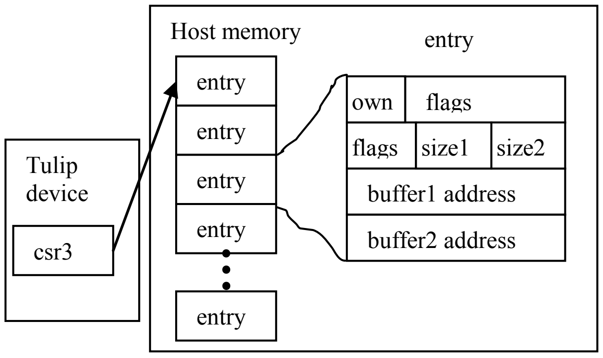

Figure 1. Receive Ring

The ownership bit is the key to the safe usage of the device. When this bit is 1, the device owns the entry and the driver should not modify the entry or the buffers pointed to by the entry. When the bit is 0, the driver owns the entry and the device should not touch the entry or its buffers. Initially, the driver establishes a receive ring of entries all owned by the driver, with no buffers. The network stack grants buffers to the driver. The driver places these buffers into entries and then (and only then) marks the entries as device-owned. The device uses DMA to receive packet data into the buffers of a device-owned entry, and then switches the entry's ownership back to the driver, which then transfers the buffers back to the network stack. As the device receives packets, it proceeds through the ring sequentially (wrapping around to the first entry of the array after reaching the last entry). If the device ever encounters a driver-owned entry, it assumes that the ring is full and may drop incoming packets until the driver transfers the entry's ownership back to the device.

The SHIL must track the state of each ring, the state of each entry in each ring, and the overall state of the device. The SHIL's `TulipDevice` class tracks the overall state of the device. This class has four private boolean fields, `txConfig`, `rxConfig`, `txStarted`, and `rxStarted`, along with public properties to read the boolean fields (`TxConfigured`, `RxConfigured`, `TxStarted`, `RxStarted`). Each `Config` field is true only if the corresponding ring is set up and ready for the device to access. The following SHIL instance method inside `TulipDevice` starts reception and transmission:

```
internal void StartRxTxMiiSym()
  requires RxConfigured && TxConfigured;
  modifies this.rxStarted, this.txStarted;
  ensures RxConfigured && TxConfigured
       && RxStarted && TxStarted;
{
  csr6.Write32(CSR6.MBO | CSR6.HBD | CSR6.PS
               | (3u << CSR6.TR_ROLL)
               | CSR6.ST | CSR6.SR);
  rxStarted = txStarted = true;
}
```

Note that the `requires` clause forces the driver to configure the rings before starting transmission and reception; without this requirement, the driver could tell the device to access uninitialized rings.

The receive buffer descriptor ring is abstracted as an object of type `TulipRxRing` that provides two methods `Update` and `GiveToDevice` that allow the verified part of the device driver to read and write individual ring entries, respectively. Each receive ring entry is represented as an object of type `TulipRxDescriptor` that will be described in more detail below. It exposes the two flags mentioned before as boolean properties `OwnedByDevice` and `EndOfRing`. Two additional boolean properties keep track of buffers 1 and 2 being set (together with their lengths). With these properties, `GiveToDevice` is specified as follows.

```
internal void
GiveToDevice(TulipRxDescriptor! descriptor)
    requires 0 <= descriptor.Index
          && descriptor.Index < Capacity;
    requires descriptor.EndOfRing ==
       (descriptor.Index == Capacity - 1);
    requires
        descriptor.OwnedByDevice == false;
    requires descriptor.Buffer1Set
          && descriptor.Buffer2Set;
    ensures
        descriptor.OwnedByDevice == true;
    …
```

The method argument represents the entry to be written into the ring. (The ! marks the argument as non-null.) The first pre-condition requires the entry's index to be within bounds of the ring array. The second condition requires the `EndOfRing` flag to be set if *and only if* the entry to be written is the last one in the array. The entry must furthermore *not* be currently owned by the device, and the buffer pointers must be set. The ownership requirement is crucial for avoiding race conditions with ring accesses by the device while the buffer requirement is a step towards memory safety: buffer addresses and lengths must be initialized before they can be written into the ring, avoiding DMA accesses to random memory locations.

Notice that `GiveToDevice`'s post-condition formalizes what the method name suggests: After writing the entry, its ownership flag is set, effectively putting the entry under control of the device. The driver now has to wait until the device relinquishes control of the entry before it can write to it again (per the `OwnedByDevice == false` pre-condition).

The driver can use the `Update` method to test whether the device relinquished control of an entry. `Update` implements querying of a ring entry and is specified as follows.

```
internal bool Update(
    TulipRxDescriptor! descriptor)
    requires 0 <= descriptor.Index
          && descriptor.Index < Capacity;
    ensures
        descriptor.OwnedByDevice == result;
    …
```

Again, the argument is a non-null `TulipRxDescriptor` object with a valid array index. It reads the specified (possibly changed by the device) entry from the ring and updates (hence the name) the argument accordingly. The boolean method result indicates whether the device (still) owns the entry, which is formalized by the method post-condition.

`TulipRxDescriptor` is also part of the SHIL and formalizes the operations that the driver can perform on individual ring entries. The driver can in particular query the various status bits and set the `EndOfRing` flag mentioned above. It can also set the buffer pointers and lengths.

The methods to give buffers to receive descriptors are specific to the data structures used by Singularity's network stack and are designed to guarantee that neither the driver (nor any other process) can access buffers while they are accessed by the device.

```
internal void Buffer1Claim(Packet! packet)
    requires packet.Full;
    requires Buffer1Set == false;
    ensures packet.Empty;
    ensures Buffer1Set == true;
    …
```

`Buffer1Claim` in `TulipRxDescriptor` can be used to pass a buffer (of type `Packet`) to a descriptor. Notice that this sets the descriptor's `Buffer1Set` property. The "claiming" of the buffer by the descriptor is indicated by the change from `Full` (in the pre-condition) to `Empty` (in the post-condition). Only `Full` buffers can be accessed (this is enforced by `Packet`'s method specifications). The inverse method `Buffer1Retrieve` lets the driver later—when the device does not own the entry—retrieve the filled buffer from the descriptor and pass it up the network stack. But the simple Full—Empty switch effectively prevents any buffer accesses by the driver while the buffer is being processed.

4. A STATICALLY CHECKED DRIVER

The untrusted part of the Tulip driver interacts with a `TulipDevice` object to configure the device, change its state (to, e.g., start and stop transmitting packets), and query its status through methods like `StartRxTxMiiSym` (section 3). We use Boogie to verify statically that the driver meets the pre-conditions of these methods. This (1) enforces validity of arguments used to communicate with the device (such as value ranges) and (2) ensures that requirements regarding the state of the device (such as the requirement that transmit and receive ring addresses be configured before transmitting and receiving buffers) are met.

The driver manages the transmit and receive buffer descriptor rings as circular FIFO queues (that are processed by the device in order), and uses the SHIL ring and descriptor objects (see section 3) to populate and query the rings. Again, Boogie verifies that descriptor state (ownership) and argument validity pre-conditions are met by the driver implementation. Invariants are used to keep track of the state of individual descriptors.

It took about 5 person-weeks (by someone who had a little previous experience with Boogie but no experience with Singularity) to refactor the driver into SHIL and untrusted parts, add trusted annotations to the SHIL, and add enough untrusted annotations to the untrusted part of the driver to allow verification. Some of this effort went into reading the device documentation (200 pages, though not all of it was relevant to driver safety) and translating this into SHIL annotations. Some of the effort went into issues with Boogie, which is itself a research project; occasionally we ran into bugs or unimplemented features in Boogie. Much of the effort was just a matter of translating our intuitive ideas about why the driver was safe into explicit pre-conditions, post-conditions, and invariants; this was generally an iterative process of adding some annotations, receiving error messages from Boogie because the annotations weren't strong enough to imply what Boogie was trying to prove, and then adding more annotations until Boogie reported no errors. For example, if a method f calls method g, which calls the SHIL, then we may add a pre-condition to g to help satisfy the SHIL's pre-condition, which may then require us to add a pre-condition to f to help satisfy g's pre-condition.

Altogether, the resulting SHIL contained about 160 annotations (requires, ensures, invariant, and modifies) to specify packets, packet fragments, packet FIFOs, and packet addresses (these are applicable to all network devices, not just the Tulip device), and about 300 annotations to specify the Tulip device. The untrusted part of the driver used about 270 annotations. These annotations were substantial, but still smaller than the original driver code, which was about 1800 (non-comment) lines. The final driver code, after refactoring and adding annotations, was about 3200 (non-comment) lines: 1100 lines of code + 300 annotations in the SHIL, and 1500 lines of code + 270 annotations in untrusted part.

In a few places, we were unable or unwilling to add enough annotations for verification to succeed entirely statically, and we decided to add run-time checks instead. In about 10 places, the theorem prover's arithmetic checker couldn't verify properties about bitwise arithmetic and about the modulus operator. In about 10 other places, it was either not worth the effort to add many annotations, or the property depended on the network stack (e.g. the number of fragments allowed in a transmitted packet); the network stack was beyond the scope of our verification.

A large part of the complexity of the untrusted part of the Tulip driver stems from the complex interaction between the driver, the Singularity kernel, and the network stack. The kernel-driver interaction follows a protocol of its own: the kernel first creates the driver, then initializes, starts, and stops it, in this order. The driver performs different steps in each of these methods: it creates the ring objects in the constructor but does not initialize them until it is initialized itself. It immediately configures the device with the newly initialized ring addresses, but it only starts the device when the kernel tells the device to do so, and so forth. More than half a dozen complex invariants keep track of all these state changes. Boogie verifies that the driver preserves these invariants throughout the driver's lifetime. The invariants, in turn, allow Boogie to verify the driver-SHIL interaction.

The network stack uses inter-process communication channels to interact with the device. These interactions can only occur after the driver is started (see above). Additionally, the driver defines contracts on the data structures passed between network stack and driver. These contracts were already expressed by the original driver developers as (dynamically checked) Sing# `requires` and `ensures` clauses. We were able to verify the correctness of the `ensures` clauses (post-conditions), while verification of the network stack to statically prove the driver's `requires` clauses (pre-condition) is future work. For now, we rely on the run-time checks automatically generated by the Sing# compiler.

5. RELATED WORK

The DevIL language [12] provides a concise, declarative syntax for specifying the bit-level layout of device registers and the legal bit-level operations on these registers. DevIL can perform some simple static checks on the declared layout, such as checking that bit fields do not overlap. The NDL language [6] and the HAIL language [13] build on DevIL's ideas, including state specifications as well as layout specifications. However, DevIL, NDL, and HAIL do not have powerful verification condition generators and theorem provers like Boogie, so they are not able to statically verify deeper properties about arithmetic and state. We believe that the DevIL/NDL/HAIL approaches are complementary to our approach. For example, if we could generate code like Section 3's `StartRxTxMiiSym` method automatically from a concise, declarative specification, then the SHIL would be smaller and more trustworthy.

Wittie [14] wrote a network device driver in a type-safe language called Clay that supported linear types and static verification of arithmetic constraints. This work did not cover DMA, though, and required porting drivers to a new and unfamiliar language. Our SHIL approach using Spec# and Boogie presents an easier path for incrementally verifying properties of existing drivers.

The Metal [8] and SDV [1] tools have found an amazing number of bugs in OS code with very little programmer effort. These tools exploit the fact that programmers tend to make simple mistakes, such as forgetting to release a lock, over and over again. As far as we know, though, these tools have not been applied to hardware-software interfaces. Our approach requires programmer effort to specify hardware-software interfaces, but rewards the programmer with more thorough checking of programmer-specified properties.

6. ACKNOWLEDGMENTS

We thank Rustan Leino, Mike Barnett, Ronald Middelkoop, and Manuel Fähndrich for their assistance with Sing#, Spec#, and Boogie, and Orion Hodson for helping us understand the Singularity Tulip driver.

7. REFERENCES

[1] 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*. EuroSys, 2006.

[2] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. *The Spec# programming system: An overview*. CASSIS, 2004

[3] Brian Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, David Becker, Marc Fiuczynski, Craig Chambers, Susan Eggers. *Extensibility, Safety and Performance in the SPIN Operating System*. Symposium on Operating System Principles (SOSP), 1995.

[4] Edoardo Biagioni. *A structured TCP in standard ML*. SIGCOMM, 1994.

[5] Luca Cardelli. *Type systems*. The Computer Science and Engineering Handbook. CRC Press, 2004. Chapter 97.

[6] Christopher L. Conway and Stephen A. Edwards. *NDL: A Domain-Specific Language for Device Drivers*. Languages, Compilers, and Tools for Embedded Systems (LCTES), 2004.

[7] Digital Equipment Corporation. *DIGITAL Semiconductor 21140A PCI Fast Ethernet LAN Controller Hardware Reference Manual.* http://www.intel.com/design/network/manuals/21140ahm.pdf

[8] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. *Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code*. Symposium on Operating Systems Principles (SOSP), 2001.

[9] 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*. EuroSys, 2006.

[10] Michael Golm, Meik Felser Christian Wawersich, and Jürgen Kleinöder. *The JX Operating System*. USENIX Annual Technical Conference, 2002.

[11] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, Andrew Tolmach. *A Principled Approach to Operating System Construction in Haskell*. International Conference on Functional Programming (ICFP), 2005.

[12] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. *Devil: An IDL for Hardware programming*. Operating Systems Design and Implementation (OSDI), 2000.

[13] Jun Sun, Wanghong Yuan, Mahesh Kallahalla, Nayeem Islam. *HAIL: a language for easy and correct device access*. EMSOFT, 2005.

[14] Lea Wittie. *Type-Safe Operating System Abstractions*. Ph.D. Thesis, 2004. Dartmouth Technical Report TR2004-526.