Generating a Statically-Checkable Device Driver I/O Interface

Lea Wittie
Bucknell University
lwittie@bucknell.edu

Chris Hawblitzel
Microsoft
Chris.Hawblitzel@microsoft.com

Derrin Pierret
Bucknell University
dpierret@alum.bucknell.edu

ABSTRACT

Device drivers are known to be a main source of operating system bugs. Several research groups have created driver specification languages that dynamically check pre- and postconditions on the IO operations of a device driver. The low-level type-safe language, Clay, has the facilities to statically check the safety of a device driver but is difficult to use directly. We have created a new device driver specification language, Laddie, which compiles the IO interface of a device driver to Clay thus leveraging its static safety checking while remaining simple to use.

1. INTRODUCTION

Modern operating systems are known to contain many bugs. A study by Chou et al. [3] examined the number and location of bugs in the Linux OS and found that the device drivers, which accounted for 70% of the OS code, accounted for almost 90% of the bugs. One possible explanation is that drivers are written by a wide range of programmers who know the kernel interface well but may be unfamiliar with the device or vice versa. The programmers are likely to make mistakes and, using cut and paste, propagate the mistakes over many drivers. Another explanation is that most drivers are written in non-type-safe languages like C and assembly.

Language technologies can help to alleviate the driver problem by detecting bugs. There have been three main specification languages for device drivers in the past few years; Devil [11, 17], NDL [4], and Hail [19, 25]. The specifications describe how a driver is supposed to function. Although all three languages perform some simple static checking, these languages compile to C and use assertions, state machines, and if-statements to detect bugs in the driver code at run-time. These languages are discussed in greater detail in Section 7.

Detecting bugs at run-time isn't sufficient to produce bug free drivers because in order for a programmer to be sure all bugs have been fixed, they must test every execution path in the driver. Drivers are frequently complex enough that this would be an impossible task and the end-user will still be the one to notice any remaining bugs. Run-time error detection is also less useful in embedded systems where patches are expensive.

This paper makes two contributions: a low-level C-like type-safe language for systems programming where the types are statically checked, and a high-level specification language for device driver IO which compiles into our type-safe language and leverages its static checking. Our type-safe language, Clay, provides the following compile-time guarantees: each process only handles its own memory, system state is tracked, and state invariants are verified in function pre- and postconditions. However to accomplish this, Clay is also similar to a typed assembly language and thus difficult for programmers to use. Unlike Clay, our device driver language, Laddie, has a simple syntax based on Hail and can be easily used by programmers to specify a driver/device IO interface. Laddie's syntax is also very similar to the description in the manual for many devices.

We have validated these techniques by building a compiler that generates Clay code from Laddie code and a working device driver written in Laddie and Clay.

2. DEVICE DRIVERS

A device is hardware attached to your computer, such as a mouse or a network card. A device driver is OS code that allows your computer to communicate with a specific device. A driver consists of a set of functions it performs and a communication interface between the driver and the device. The driver is usually described in a large manual, frequently using a description in English or a set of charts to explain the driver functions and the communication interface. This paper is focused on the communication interface between a driver and its device.

Communication between a driver and its device is usually accomplished through a series of register reads and writes. The registers may be mapped onto memory or use a special set of read and write functions. The rules governing when a register can be accessed frequently include specified conditions on the logical state of the device. For example, some registers should not be accessed when device interrupts are turned on. Although it is relevant to safety, logical state, such as device interrupt status, is not usually represented by values in driver memory.

An IO specification for a driver states the rules for accessing each device register. These rules depend on read or write access, byte size, the read or written value, and logical device state. A register may be readable, writable, or both. The access to a register has a specific byte size. For instance a register may accept 1 or 2 byte writes but only 2 byte reads. Frequently, a writable register can only safely accept a specific set of write-values. Values outside that set might cause unknown or unwanted behavior. Similarly, a readable register may have a specified set of values that might be read from it. Access to a register may also depend on the current logical state of the device and this state may change as a result of

accessing the register. Altogether these factors provide a complex set of rules for accessing the registers of a device.

When a driver fails to meet the IO specification for its device, the device could be placed in an untenable state where it breaks or reboots. Rebooting can mean data loss; a CD-RW writing a CD, network card sending packets. The driver could also read incorrect data from the device. For instance, a mouse driver thinks it is asking for the mouse X-coordinates and the mouse thinks the driver wants to know if interrupts are on. The data may look realistic enough that the driver may not notice something has gone wrong. Or, in the reverse, a device could perform unintended actions on data written by the driver. For example, garbage written to a CD-RW, a mouse pointer that gets stuck in a corner of the screen, a network card that drops packets without reporting any errors, etc.

Regardless of the exact outcome, driver IO mistakes are a source of serious bugs and should be prevented. Our work provides a language for encoding the communication specification laid out in a device manual and generates code in a low-level language that can enforce the specification at compile time.

3. CLAY

We informally present the low-level language Clay which uses dependent types to provide pre- and post conditions on functions and statically checks these conditions for each function call. Clay is also able to statically detect many other errors including buffer overflows, kernel stack overflows, NULL pointer uses, freed memory uses, and aliasing errors. Static error detection moves the burden of error checking from run time to compile time thereby allowing the programmer to find errors while compiling and lessening the number of errors the end user sees. Writing in Clay guarantees safety, but like Proof Carrying Code [15], Clay is intended for verification and is difficult to write in.

The Clay programming language is a type-safe variant of C/C++. This language was developed at Dartmouth College by Chris Hawblitzel, Gary Morris, Eric Krupski, Ed Wei, and Lea Wittie.

Clay's type system is similar to dependently typed languages such as DML [22]. It uses dependent types, also known as indexed or singleton types, to track variable values at compile-time. Clay has been formalized as a typed lambda-calculus named λ-low . This formal language has been shown to be type-safe and the proofs of progress, preservation, and erasure in λ-low as well as a translation to System F [13] (polymorphic lambda-calculus) can been found in [10].

Clay is able to express pre- and postconditions on function calls.

```
exists [int X; X>=0 && X<=5] Int[X] add
[int N, int M, int P; M>=0 && 2*P==N]
(Int[N] n, Int[M+4] m) {
  let x = n + m;
  if (x > 5)  return 5;
  else return x;
}
```

This function adds two numbers. If the sum is greater than 5, the function returns 5, otherwise it returns the sum. This example uses several logical-level variables, also known as specification-level variables, ghost variables, or type-level variables. These logical-level variables, seen in capital letters, are used at compile-time to check the function specification but are not around at run-time. For instance the singleton type Int[N] can be read as the set of integers i where i is equal to N, or more formally $\{i : \text{int} \mid i == N\}$.

Int the above example, the type information after the function name forms the precondition $M \geq 0$ and $2 * P == N$ so M is positive and N is even. The return type shows a postcondition $X \geq 0$ and $X \leq 5$ so X is between zero and five. Constructs on pre- and postconditions are encoded in the same style as DML, DTAL [24], and ATS [23]. They are also very similar to the conditions in ESC Java [8]. The conditions to access a device register can be encoded in the header of a Clay function.

Clay is also able to track logical device state by creating a new singleton type for each state component.

```
@type0 State[int Parameter] = native
```

This type depends on an integer type variable. Values of this type can be passed in and out of functions and the type variable can be used in pre- and postconditions. The beginning of the declaration, @type0, declares the type to be linear and logical rather than actual state. These features provide Clay's memory safety [20].

```
exists [int X; X>=0 && X<=5] @[Int[X],State[Y]] add
[int N, int M, int P, int Y; M>=0 && 2*P==N && N>Y]
(Int[N] n, Int[M] m, State[Y] y) {
  let x = n + m;
  if (x > 5)  return 5;
  else return @(x,y);
}
```

This example modifies the previous example to add a parameter for device state, y, and include $N > Y$ in the precondition. We use a tuple (@[]), common in functional languages like ML or Haskell, to return both x and the state y.

Clay's typechecker verifies that all function calls obey the function precondition. It also verifies that given each function precondition and function body, it can conclude that the postcondition also holds. The verification is done using the Omega [16] constraint checker. The typechecker also verifies that the code is free of other errors previously mentioned.

After typechecking, the Clay compiler translates the Clay code to C++code. Clay currently uses a templated struct containing an array to handle tuple return types. This is the only actual C++feature in the generated C++code which can be linked and compiled with C code.

Since Clay's typechecker has already verified that function pre- and postconditions are always true, these conditions do not generate run-time checks in C++. Likewise, logical device state variables such as the State example shown above have also been checked statically and will only generate C++code where the typechecker could not verify safety statically; i.e. where the device state may have changed and the driver must query the device. The logical device state variables themselves do not generate C++variables and therefore do not use additional space in memory. This is valuable in systems with space constraints such embedded systems. Additionally, the static typechecking produces robust code with fewer errors at run-time. This could lead to fewer patches later in the life of software written in Clay.

Although Clay is expressive enough to include a driver IO specification in its code and powerful enough to statically check that the specification is obeyed, it is a difficult language to write in. Figure 1 shows just the header of one register read function in Clay.

Clay is relevant to embedded systems programming because it has many of the properties laid out in [9]. *Correctness*: a Clay program is statically guaranteed to meet the specification in it's types and pre- and postconditions. *Concurrency*: Clay has built-in features for resource sharing and concurrency via locks [10, 20]. Clay programs have the same *time and space constraints* as C++as well as its ability to handle *asynchronous events*.

4. LADDIE

The Language for Automated Device Drivers (Laddie) was intended to provide a simple and safe way to encode the IO specifications for a device driver. Laddie was developed by Lea Wittie and

```
native exists [u32 Q2, u1 IC, u1 ER, u3 Code,
               u11 RxBytes; Q2==RxBytes]
@[Base[D,A], Busy[D,B], RxQUsed[D,Q2],
  Window[D,W], Int[IC], Int[ER],
  Int[Code], Int[RxBytes]]
read_RxStatus
[u32 A, u32 D, u1 B, u32 Q2, u32 W;
 (W>=0 && W<=6) && (W==1 && B==0)]
(Int[A] addr, Base[D,A] base, Busy[D,B] busy,
 RxQUsed[D,Q1] rxQUsed, Window[D,W] window);
```

Figure 1: Function header for the 3c509 RxStatus register

Derrin Pierret. Three recently developed driver specification languages, Devil [11, 17], NDL [4], and Hail [19, 25], provide much of the expressiveness we needed but they check for most specification usage errors at run-time. The languages are discussed in more detail in Section 7. The authors of this paper and several undergraduate computer science students looked over the syntax of NDL, Devil, and Hail. Of the three, Hail's syntax was the easiest to read and the closest match to the descriptions seen in device manuals. Therefore, we based Laddie's syntax on Hail.

A Laddie specification declares the IO rules for reading and writing the registers of a device. These rules give the pre- and postconditions for reading or writing each register. A specification is separated into two sections. The top section lists any logical state used in the pre- and postconditions. The bottom section lists the IO rules for reading or writing each register. Figure 2 shows part of the 3Com 3c509 network card specification.

4.1 Logical Device State

IO operations are frequently dependent on logical device state which is not tracked in memory. The top of Figure 2 shows the logical device state on a 3c509 network card. Each component consists of a name, a type, and for integer components, an optional set of allowed values. The types are booleans and 32-bit integers. The booleans are C++ -style and equate to zero and one semantically. The 3c509 has several components to its logical state. Although these are not stored in computer memory, the 3c509 driver still needs to track them. Each register is accessed through an IO port. The set of ports allocated to this device is smaller than the number of registers the 3c509 actually uses. Therefore the registers are divided up into 7 windows [1]. Since this is a PIO network card, the receive and send queues are on the device (as opposed to in memory). The driver depends on the amount of data in these queues. Some IO operations put the device in a busy state and it is dangerous to do IO operations aside from querying the status register while the device remains busy. The Statistics registers can only be read when statistics gathering is disabled.

4.2 IO Rules

The IO rule for a register consists of basic information, subfields within the register, and pre- and postconditions to access the register. Below the logical state, Figure 2 shows the IO rules for four registers.

Basic information

The basic information in an IO rule, except for the register offset, is given using C-style assignment statements which set the variables name, size, type, and access.

[1]This is a common phenomenon occurring in many other devices such as the SMSC LAN91C111 10/100 Non-PCI Ethernet Single Chip MAC + PHY [18]

```
integer Window [0:6];
integer RxQUsed;
boolean Busy;
boolean StatsEnabled;

port 0x0E {
  name = Command;
  size = 2;
  access = write;

  [15:11] command;
    enum {GlobalReset, SelectRegWindow, ...,
    StatsEnable=21, StatsDisable, ...};
  [10:0] argument;

  requires Busy==false;
  requires switch command {
    GlobalReset: argument==0;
    StatsEnable: argument==0;
    ...
  }
  ensures switch command {
    GlobalReset: Busy==true;
    SelectRegWindow: Window==argument;
    StatsEnable: StatsEnabled==true;
    ...
  }
}

port 0x0E {
  name = Status;
  size = 2;
  access = read;

  [15:13] window;
  [12] command_in_progress;
  [11] reserved;
  [7] update_stats;
  ...

  ensures Window==window && Busy==command_in_progress;
}

port 0x08 {
  name = RxStatus;
  size = 2;
  access = read;

  [15]    IC;
  [14]    ER;
  [13:11] code;
  [10:0]  rxBytes;

  requires Window==1 && Busy==false;
  ensures RxQUsed==RxBytes;
}

port 0x00 {
  name = RX_PIO_DataRead;
  size = 4;
  type = repeated;
  access = read;

  requires Window==1 && Busy==false && RxQUsed>4*count;
  ensures RxQUsed==old(RxQUsed)-(4*count);
}
```

Figure 2: Part of the 3Com 3c509 specification

```
name = RX_PIO_DataRead;
size = 4;
type = repeated;
access = read;
```

Each IO rule must have a unique name. The register size gives the number of bytes in this register. The size statement is optional and the default size is 1 byte. If this is a repeated IO call, such as insl(), then the type is repeated and the pre- and postconditions can refer to a variable named count. The IO function will take an input count which determines how many times it repeats the IO call. The default type is non-repeated since that is the standard IO call. The access for a rule may be read, write, or readwrite. A readwrite access implies that the rule is for both reading and writing the register. The access statement is optional and defaults to readwrite. The register offset is declared before the IO rule. Separate read and write rules can be given for the same register provided they use different names.

```
port 0x0E { name = Command; }
port 0x0E { name = Status; }
```

This feature is used when a register has separate rules governing reads and writes as seen in the Command and Status register of the 3c509. It is also useful when the same port offset has a different meaning in different windows.

Fields

Registers are frequently divided up into named fields which hold logically distinct data. The Status register in Figure 2 is divided into many fields, four of which are shown here.

```
[15:13] window;
[12] command_in_progress;
[11] reserved;
[7] update_stats;
```

Fields can be reserved or omitted which means they should not be used by the driver. Bit 11 of the Status register is explicitly reserved. Bits 10-8 are reserved implicitly. Laddie's static semantics forbid fields to overlap or go beyond the size of the register.

Pre- and postconditions

The precondition section is given before the postcondition section. Both sections are optional and may be omitted. A precondition is preceded by the keyword requires and a post condition is preceded by ensures. The actual conditions are a set of boolean statements on the device state and register fields.

```
requires Window==1 && Busy==false;
ensures RxQUsed==RXbytes;
```

Conditions may use the standard boolean and relational operators as well as the $+$, $-$, and $*$ math operators. The relational and math operators perform standard C operations on 32-bit integers. We will describe the conditions more formally in Section 5. The preconditions of a readable register may not include the fields because their value is not known yet.

The set of allowed values in a logical state declaration is an implicit pre- and postcondition on every register access.

```
integer Window [0:6];
```

The restrictions on Window add the condition (Window$\geq$0 and Window$\leq$6) to every pre- and post condition where Window is mentioned.

Debugging

It is possible to write false conditions such as

```
requires 1>5;
ensures Window<1 && 3<Window;
```

however, these will be caught during a consistency test that checks for pre- and postconditions that are impossible to satisfy. This test and Laddie's static semantic checking allow specifications to be checked for errors before writing the Clay portion of the driver.

Postconditions referring to old device state

Postconditions may refer to old device state using the keyword old.

```
ensures RxQUsed==old(RxQUsed)-(4*count);
```

This postcondition relates the postcondition value of device state to its precondition value.

The remaining topics in Laddie are syntactic sugar for notational convenience.

Enumerated field values

Fields may be declared with an enumerated set of allowed values as seen in the command field of the Command register

```
[15:11] command;
  enum {GlobalReset, SelectRegWindow, ...,
  StatsEnable=21, StatsDisable, ...};
```

The format is similar to the standard C enum. On a write, this field can only take one of the enumerated values. On a read, assuming the register was readable, the device will return one of those values for this field. (Note: Enum values on a read can only be enforced dynamically and constitute a check on the device rather than the driver. Laddie is primarily intended for statically checking the driver.)

Conditions on the whole IO value

Conditions can refer to the whole read or written value rather than to its fields. The preconditions of a readable register may not include the value because it is not known yet. This avoids making a special field that is the size of the whole register when the register in not normally partitioned into fields. We use the keyword value to make this problem simpler to express. The 3c509 does not use this feature but the Signature register of the Logitech busmouse driver does.

```
requires value == 0xa5;
```

Switch statements

For convenience, a switch statement on a register field may be used instead of a complex boolean expression.

```
ensures switch command {
  SelectRegWindow : Window==argument;
  StatsEnable     : StatsEnabled==true;
  ...
}
```

is equivalent to

```
ensures ((command==SelectRegWindow && Window==argument)
  || command!=SelectRegWindow)
  && ((command==StatsEnable && StatsEnabled==true)
  || command!=StatsEnable)
  && ... ;
```

The body of each switch case is a boolean statement. Like C, Laddie switch statements have an optional default case.

Multiple pre- or postconditions

Multiple conditions may be given for a register. In the Command register of the 3c509, we used multiple preconditions for simplicity.

```
requires Busy==false;
requires switch command { ... }
```

The set of preconditions is interpreted as if there was an && between them.

4.3 Relationship to Manual

Register information in a device manual is frequently presented in chart form with a text description of each fields and any pre- and post conditions.

For example, the Interrupt Enable register (IER), port 0x01, of the National Semiconductor PC16550D UART [6] is shown in chart form as

bits	0	1	2	3	4..7
field	ERBFI	ETBEI	ELSI	EDSSI	zeros

The DLAB field, in the Line Control register, is explained as follows:

> "This bit is the Divisor Latch Access Bit (DLAB). It must be set high (logic 1) to access the Divisor Latches of the Baud Generator during a Read or Write operation. It must be set low (logic 0) to access the Receiver Buffer, the Transmitter Holding Register, or the Interrupt Enable Register."

From this we see that the IER may only be accessed when the DLAB is zero. The IER allows 1 byte read or write access, broken into four fields and bits 4...7 should be written with zeros (the same as a reserved field) and reading those bits has no meaning. Therefore the Laddie specification for the IER part of the PC16550D UART is

```
integer DLAB [0:1];

port 0x1 {
  name=IER; size=1; access=readwrite;
  [0] ERBFI;
  [1] ETBEI;
  [2] ELSI;
  [3] EDSSI;
  requires DLAB==0;
}
```

We chose to make the DLAB state component an integer to match the description in the manual. It would be equally correct to use a boolean.

The 3c509 registers in the 3Com manual [5] are explained in a nearly identical fashion. [2]

5. TRANSLATION OF LADDIE TO CLAY

Although our eventual target language is C, we first compile Laddie to Clay in order to leverage Clay's type system and static type-checking. Clay is able to express the invariants on registers and logical state seen in a Laddie specification. A formal translation of Laddie to Clay is available at [21]. Due to space considerations, we present an informal translation here.

Several features of Laddie are syntactic sugar to provide ease-of-use. Our informal translation uses a desugared version of Laddie. This version includes logical device state, basic register information, fields, pre- and postconditions, and references to old state. All other features of Laddie can be simplified into this desugared format.

Because Clay is a language without side effects, it must drop to native C++to perform any IO calls or memory accesses. Therefore, each IO function in Clay makes a call to native C++code where the actual IO is performed. A Laddie specification produces three Clay files: IO function stubs in Clay, the actual IO functions in C++, and macros to facilitate using the Clay IO functions.

[2]The 3Com copyright forbids reproduction of the 3c509 manual in any form.

```
native exists [u32 SRxQUsedout, u3 Fcode,
  u11 FrxBytes, u1 Fincomplete, u1 Ferror;
  (RxQUsedout==G0)]
@[Statebase[D,A], StateBusy[D,SBusyin],
  StateRxQUsed[D,SRxQUsedout],
  StateWindow[D,SWindowin], Int[Fcode],
  Int[FrxBytes], Int[Fincomplete], Int[Ferror]]
read_RxStatus
[u32 A, u32 D, u1 B0in, u32 SRxQUsedin,
 u32 SWindowin; (SWindowin>=0 && SWindowin<=6)
 && (SWindowin==1 && SBusyin==0)]
(Int[A] addr, Statebase[D,A] base,
 StateBusy[D,SBusyin] cap_Busy,
 StateRxQUsed[D,SRxQUsedin] cap_RxQUsed,
 StateWindow[D,SWindowin] cap_Window);
```

Figure 3: Generated Clay IO function stub for the RxStatus register

5.1 The Clay IO function stub

The generated Clay IO function stub for the RxStatus register from Figure 2 is shown in Figure 3. Notice that it is very similar to the handwritten Clay function stub in Figure 1. The basic format of the produced Clay IO stub is

```
native postcondition return_type
  function_name precondition (inputs);
```

The native keyword in Clay indicates that this is a function stub with a matching function in native C++code.

Function input

Each IO function requires the base address of the device, the register offset, and any data if the IO is a write. Since the base address is used in all IO functions, it is omitted in Laddie and added back in to the generated Clay code. The function inputs therefore include the base address and its capability as well as capabilities for each logical device state and if this is a write function, the fields. The register offset is fixed for each IO function and will be used in the C++IO function.

```
(Int[A] addr, Statebase[D,A] base, // base address
 StateBusy[D,SBusyin] cap_Busy,      // busy state
 StateRxQUsed[D,SRxQUsedin] cap_RxQUsed,
 StateWindow[D,SWindowin] cap_Window);
```

Each capability has two type variables, one for the value of the capability and one for the memory address of the driver data, always D. We append "in" and "out" to the name of type variables for logical states to differentiate input and output (this allows us to implement **old**). If this is a write function, we generate a singleton integer with one type variable for each field. The F and S designations on field and state type-variables are used to match Laddie's flexible identifier names with Clay's more rigid naming scheme which requires an initial capital letter for certain identifiers. We also generate a singleton integer for the base address of the device (Int[A] addr).

Precondition

The precondition declares all of the type variables used in the inputs and includes the register precondition as well as the global conditions on any logical device states.

```
[u32 A, u32 D, u1 SBusyin, u32 SRxQUsedin,
 u32 SWindowin; (SWindowin>=0 && SWindowin<=6)
 && (SWindowin==1 && B0in==0)]
```

The RxStatus register example shows declarations for the A and D variables as well as those for the three states. Each type variable

actually has kind int but is shown here using syntactic sugar which abbreviates

$$\text{int } X;\ X \geq 0 \,\&\&\, X < 4294967292$$

to u32 X where X is a 32-bit unsigned integer. Likewise, the boolean Busy has an abbreviated kind of u1. The global condition allows Windows between 0 and 6 and the precondition from the RxStatus register forces Window to be 1 and Busy to be 0 (false).

Function name

For simplicity, we coalesce the register name and access into the function name, producing read_RxStatus in our example.

Return type

The return type is a tuple containing all of the capability types and since this is a read function, the field types.

```
@[Statebase[D,A], StateBusy[D,SBusyin],
  StateRxQUsed[D,SRxQUsedout],
  StateWindow[D,SWindowin],
  Int[Fcode], Int[FrxBytes], Int[Fincomplete],
  Int[Ferror]]
```

As seen in the inputs, the capability types have two type variables and the field types are singleton integers. Any type variables which are the same as those in the inputs use the same names here. For instance, the device base address (A) and the value of Busy (SBusyin) have not changed. Any new values, such as the updated value of the logical state RxQUsed or the returned fields, use a new type variable.

Postcondition

The postcondition has the same general contents as the precondition. Any new type variables are declared using Clay's syntactic sugar to simplify the declarations. Any relevant global conditions and the register postcondition are shown here.

```
exists [u32 SRxQUsedout, u3 Fcode, u11 FrxBytes,
u1 Fincomplete, u1 Ferror; (SRxQUsedout==FrxBytes)]
```

The exists keyword declares the return type to be an existential where the type variables declared here have some value which satisfies the postcondition.

5.2 Types for logical state

Laddie also generated a type for each logical device state and for the base address of the device.

```
@type0 StateWindow [int Device, int Val] = native
@type0 StateBusy [int Device, int Val] = native
@type0 Statebase [int Device, int Val] = native
```

Each type has two type variables. The first is the driver memory address. This will be the same for all states associated with a specific driver and device. The driver address is used to link all of the capabilities to a specific driver and device so the states of two different devices cannot be accidentally interchanged. This allows the produced code to scale for multiples of the same device or different devices that happen to have the same logical state names.

Unlike the logical device state components, the base address is a state component of the driver and is stored in driver memory. Although we could have used the memory capability for the base address, we chose to create a new state component so that Laddie did not need to know the exact location of the base address within driver memory.

5.3 Native C++IO function

There is a matching C++function for every Clay IO function stub.

```
inline struct Clay_Obj<4> read_RxStatus
(unsigned long addr)
{
  int value = inw(addr + 0x08);
  struct Clay_Obj<4> c;
  c.A[0] = (value >> 11) & 7;   // bits 11:13
  c.A[1] = (value >> 0) & 2047; // bits 0:10
  c.A[2] = (value >> 15) & 1;   // bit 15
  c.A[3] = (value >> 14) & 1;   // bit 14
  return c;
}
```

When Clay compiles, all of type annotations and logical device state values are erased leaving us with a function that takes and returns only the base address and fields. A Clay_Obj is a struct containing an array (A) of a given length. It matches the Clay tuple that is returned by the Clay function stub. The necessary bit twiddling is done for each register field [i:j] using

$$\underbrace{(\text{value } >> \ i) \,\&\, \textstyle\sum_{n=0}^{j-i} 2^n}_{\text{for reading}} \qquad \underbrace{(\text{value } << \ i) \,\&\, \textstyle\sum_{n=i}^{j} 2^n}_{\text{for writing}}$$

a similar C++function is produced for register writes.

5.4 IO macros

Finally, because all Clay IO stubs generated by Laddie are repetitive and take the base address as well as a collection of known capabilities, Laddie generates a macro which takes the name of input or output variables.

```
#define R_RxStatus(code, RxBytes, IC, ER) \
let [] (s_baseSTATE##2, \
    s_BusySTATE##2, s_RxQUsedSTATE##2, \
    s_WindowSTATE##2, code, RxBytes, IC, ER) = \
  read_RxStatus(IOADDR, s_baseSTATE, s_BusySTATE, \
    s_RxQUsedSTATE, s_WindowSTATE); \
  s_BusySTATE = s_BusySTATE##2; \
  s_RxQUsedSTATE = s_RxQUsedSTATE##2; \
  s_WindowSTATE = s_WindowSTATE##2; \
  s_baseSTATE = s_baseSTATE##2;
```

A programmer using the generated Clay code would replace unsafe IO calls like

```
rx_status = inw(ioaddr + RX_STATUS);
short error = rx_status & 0x3800;
short pkt_len = rx_status & 0x7ff;
```

with its respective safe IO call and macro usage.

```
R_RxStatus(code, pkt_len, complete, error);
```

6. RESULTS

Laddie is simple language to use and we were able to write Laddie specifications for several drivers

- 3Com 3c509 Network Interface Card
- National Semiconductor PC16550D UART
- National Semiconductor DP8573A Real Time Clock

These specifications took less than an hour to write after we had thoroughly read the respective device manuals published by the manufacturers. Our Laddie specifications are available at [21].

Simple timing tests conducted on a handwritten Clay 3c509 driver and Donald Becker's [3] C 3c509 [2] driver using a series of ping packets indicate only a slight increase in run time for the Clay 3c509 driver.

To evaluate the decrease in programmer workload, we can make code length comparisons between Laddie, Clay, and C.

[3]Donald Becker has written many of the Ethernet device drivers for Linux.

C 3c509 driver

Donald Becker's 3c509 driver is around 800 lines of code. It is much shorter than the equivalent Clay driver since it does not include the IO specification.

Clay 3c509 driver

The Clay 3c509 driver has four sections of code:

Portion of Driver	Lines of Code
IO interface	1188
driver functions	1939
system code	1090
capability handling code	411
total	4628

The driver functions are the main body of the driver. The IO code includes all of the IO functions, their native C++translations and their macros. The system code is made up of the included Linux .h files for the original driver since they had to be translated into Clay. However, the .h files are not driver specific and could be re-used. The capability handling code converts the driver memory capability from a tuple to individual memory capabilities and back again. The IO code is about 25 percent of the total Clay code.

Laddie portion of the 3c509 driver

Laddie needed 314 sparse lines of code to replace the Clay IO code. (The generated Clay was 1197 dense lines of code, very similar to the original 1188 hand-written lines of Clay code). The main reason for the dramatic difference between Laddie IO code length and Clay IO code length is that the Clay IO code is very repetitive. The main things that differ from function to function are data size, offset, and pre- and postconditions. Since Laddie's syntax provides a concise way to present the necessary information, the generator is able to provide the rest.

7. RELATED WORK

The languages that have the most in common with Laddie are Devil [11, 17], NDL [4], and Hail [19, 25]. Both these languages and Laddie provide a device driver specification for IO operations on registers. Devil, NDL, and Hail drivers are shorter than C drivers and have similar performance times. Laddie allows much of the functionality of these languages in its register IO specifications. However all three other languages provide safety through some standard static checks and run-time checks on the IO specifications while Laddie compiles to Clay which enforces most IO specification invariants at compile time. This section presents a comparison of these languages with Laddie. It also presents other related static verifiers and type safe languages.

Devil

Devil provides a specification for IO operations on registers as well as a range of legitimate port offsets from the base address and a set of variables tied to register fields. The variables provide a way to track device state at run time. Devil supports pre- and postconditions on IO operations.

Unlike Laddie, Devil is able to define a variable as the concatenation of subfields from several different registers. The compiled Devil code includes read and write functions which hide the details of how each variable is assembled. This allows simpler read and write access to complex variables.

Both Devil and Laddie statically guarantee that read/write access and size constraints are obeyed in the driver functions. Both languages can track logical device state in pre- and postconditions; Devil via standard variables and Laddie via ghost variables.

Conditions in Devil are variable assignments that must be performed before or after the IO operation respectively. In comparison, Laddie allows more flexible conditions because its conditions are boolean expressions on ghost variables. Devil's conditions only allow equality rather than the full range of boolean operators. Laddie's ghost variables only exist at compile time so they do not use extra space in memory during run time. However, Laddie, unlike Devil, is unable to refer to standard program variables in its pre- and postconditions.

NDL

NDL builds on Devil and uses similar syntax. An NDL specification includes IO operations on registers and a collection of driver functions. It also includes a state machine for the logical states a device may be in (reading, sleeping, etc..). Preconditions on the current state are used in the IO specification. NDL does not appear to support post conditions. Both NDL and Laddie allow the same IO location to have two different rules for reading and writing. NDL code compiles to C and uses runtime checks to enforce its preconditions and state machine transitions.

Unlike Laddie, the entire driver is written in NDL's C-like driver syntax. This has the advantage of allowing NDL to support buffer copying to and from a device in one or two lines of code, an operation which normally takes many IO operations.

Hail

Hail provides a specification for IO operations and invariants on registers as well as an address space description and a description of the device instantiation. The Hail compiler is capable of catching inconsistencies in a specification. The Clay compiler can catch similar inconsistencies in a Laddie specification. Like Devil, the Hail compiler generates IO functions in C with optional run time checks on the invariants. The actual driver functions of a Hail driver are written in C using the generated IO functions.

According to the HAIL website, the address space descriptions are not implemented yet in a HAIL compiler. Laddie currently provides #define stubs for the different possible addressing strategies, but Hail's syntax is easier to use and we may adopt this strategy in the future.

SDV

Microsoft SDV, static driver verifier, works on Windows device drivers based on the Windows Driver Model [12, 1]. The SDV statically checks that the driver obeys a set of built-in rules about the driver/kernel interface. The project is similar to Laddie in that both statically verify a driver's specification usage. The SDV focuses on a fixed driver/kernel specification for the Windows Driver Model while Laddie focuses on a user-defined driver/device specification which can be tailored to each device. The SDV is part of the SLAM toolkit.

Type safe languages

Typed Assembly Language [13] is similar to Clay in that both access memory through load and store primitives and use types to guarantee memory safety. TAL is also too low-level to easily write drivers. Popcorn, a safe C-like language which compiles to TAL does not support the complex pre- and postconditions needed by safe driver IO.

Hoare Type Theory [14] adds types to the standard Hoare triple to form {P} x:t {Q} where the pre- and postconditions can depend on the types. This is similar to the pre- and postconditions on Clay functions. This language is currently formalized but not implemented.

The Vault [7] programming language embeds keys to manage temporal events in the types of system resources. Keys, like the capabilities of Clay, are associated with a resource and must be held to access the resource. Vault allows a notion of pre- and postconditions with key states. However, they are less flexible than the arithmetic constraints, relations, and inequalities of Clay.

8. CONCLUSIONS AND FUTURE WORK

The combination of Laddie and Clay provides a usable specification language for device IO access functions and static type safety guarantees that the driver uses the device access functions in accordance with the specification. Using Laddie we have written several device IO specifications which are available at [21]. These specifications could be produced by the device manufacturer and distributed to driver writers along with the standard manual.

Currently, Laddie only handles the IO end of a driver. The remainder of the driver must still be in Clay, a more difficult language to use. We expect to be able to automate the translation of the system code (C structs such as the driver memory), from C to Clay with a reasonably straight forward compiler which will also generate the capability-handling code needed to access these data structures.

in Laddie	still in Clay		
IO code	driver functions	system code	capability code

We have left an easier language for writing the remainder of the driver as future work.

A mix of Clay's type system and function pre- and postconditions with a language, such as Hume [9], primarily intended for embedded systems might yield a simpler intermediate language with guarantees of correctness and space and time costs for constrained systems.

9. AVAILABILITY

An ML program formally translating Laddie to Clay, the Laddie and Clay compilers and Users Guides, and the Laddie specifications of several devices are available at [21].

10. 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. In *EuroSys Conference*, Leuven, Belgium, 2006.

[2] Donald Becker. 3Com EtherLink III 3c5x9 driver v. 1.18. http://joshua.raleigh.nc.us/docs/linux-2.4.10_html/284303.html, 2000.

[3] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating system errors. In *ACM Symposium on Operating Systems Principles*, pages 73–88, Banff, Alberta, Canada, 2001.

[4] Christopher L. Conway and Stephen A. Edwards. NDL: A domain-specific language for device drivers. In *ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, Washington, DC, June 2004.

[5] 3Com Corporation. Etherlink III Parallel Tasking ISA, EISA, Micro Channel, and PCMCIA Adapter Drivers Technical Reference. 1-800-NET-3Com, August 1994.

[6] National Semiconductor Corporation. PC16550D Universal Asynchronous Receiver/Transmitter with FIFOs. http://www.national.com, June 1995.

[7] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Snowbird, Utah, June 2001.

[8] Cormac Flanagan, K. Rustan Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Berlin, Germany, 2002.

[9] K. Hammond and G. Michaelson. HUME: A domain specific language for real-time embedded systems. In *International Conference on Generative Programming and Component Engineering*, Erfurt, Germany, October 2003.

[10] Heng Huang, Lea Wittie, and Chris Hawblitzel. Formal properties of linear memory types. Technical report, Dartmouth College, 2003.

[11] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *USENIX Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.

[12] Microsoft. Static driver verifier - finding driver bugs at compile-time. http://www.microsoft.com/whdc/devtools/tools/sdv.mspx, 2007.

[13] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):528–569, May 1999.

[14] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in Hoare type theory. In *International Conference on Functional Programming*, Portland, Oregon, 2006.

[15] George C. Necula and Peter Lee. Safe kernel extensions without run-time checking. In *USENIX Symposium on Operating Systems Design and Implementation*, Seattle, Washington, October 1996.

[16] William Pugh. The Omega project. http://www.cs.umd.edu/projects/omega/, 2007.

[17] L. Réveillère, F. Mérillon, C. Consel, R. Marlet, and G. Muller. The Devil language. Technical Report 1319, IRISA, Rennes, Francs, 2000.

[18] SMSC. LAN91C111 10/100 Non-PCI Ethernet Single Chip MAC + PHY. http://www.smsc.com, 2005.

[19] J. Sun, W. Yuan, M. Kallahalla, and N. Islam. HAIL: A language for easy and correct device access. In *ACM Conference on Embedded Software*, Jersey City, NJ, September 2005.

[20] Lea Wittie. *Type-Safe Operating System Abstractions*. PhD thesis, Dartmouth College, 2004.

[21] Lea Wittie. http://www.eg.bucknell.edu/∼lwittie/research.html, 2007.

[22] Hongwei Xi. *Dependant Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

[23] Hongwei Xi. Applied type system (extended abstract). In *post-workshop Proceedings of TYPES 2003*. Springer-Verlag LNCS 3085, 2004.

[24] Hongwei Xi and Robert Harper. A dependantly typed assembly language. In *International Conference on Functional Programming*, Florence, Italy, 2001.

[25] W. Yuan, J. Sun, and N. Islam. HAIL language specification and user guide. Technical Report DCL-TR-2005-0006, DoCoMo USA Labs, 2005.