Laddie: The Language for Automated Device Drivers (Ver 1) Bucknell Computer Science Technical Report #08-2

Lea Wittie

March 12, 2008

Abstract

Device drivers, which make up a large portion of operating systems, are notorious for buggy code. Recent work in language support for device driver writing has produced several specification languages which aid a driver writer in producing an IO interface between the driver and its device. These languages are all meant for systems programming and thus far none of them are type safe so errors in using the produced interface are still possible. We present a specification language, Laddie, which uses a type-safe language as a bridge to the final device driver. Our work maintains the ease of use of the other specification languages while adding the guarantees of a type-safe language. This type-safety means that IO interfaces written in Laddie cannot be misused by the remained of the device driver. Using Laddie, we have successfully produced IO interfaces for several devices.

1 Introduction

The Language for Automated Device Drivers (Laddie), developed by Lea Wittie and Derrin Pierret, was intended to provide a simple and safe way to encode the IO specifications for a device driver. Its syntax is based on the language Hail [9, 11], an easy to use driver IO specification language which checks specification usage at run-time. Laddie compiles to a type-safe language Clay which statically guarantees any driver using this IO interface obeys the specification.

2 Background

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.

3 Comments in Laddie

C-style // and /* */ comments are allowed. Unlike C, the /* */ comments may be nested so /* /* */ */ is allowed but /* /* */ is not.

4 Specification Layout

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 declares components in the logical state of the device. The bottom section lists the IO rules for communicating with the device. Figure 1 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 1 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. Component types may be integer or boolean. An integer component with a set of allowed values may only be a value within the set. This is enforced globally over the whole device driver. When no set of allowed values is shown for an integer, the default is any value in the 32 bit range. The booleans are C^{++} -style and equate to zero and one semantically. A boolean state component may not come with a set of values. State component names are in scope in the entire specification.

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

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]. The device tracks its current window but the information is not stored in driver memory. Since this is a PIO network card, the receive and send queues are on the device (as opposed to in memory). The device tracks the amount of data in the receive queue so that it can be read by the driver. 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 1 shows the IO rules for several registers.

[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 [8]

```
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;

  ...
  [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 1: Part of the 3Com 3c509 [3] specification

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.

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

Each IO rule must have a unique name which is in scope in the entire specification. 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() which 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 separated rules governing reads and writes as seen in the Command and Status register of the 3c509. It is also useful when the same register offset has a different meaning in different windows.

Fields

Registers are frequently divided up into named fields which hold logically distinct data. The fields are optional but any given fields must fit within the register size and may not overlap one another. Each field is given a range of bits and a name. Field names are in scope within their IO rule so multiple rules may re-use field names. The Status register in Figure 1 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. Multiple fields may be reserved and these fields may not appear in the pre- or postconditions. Bit 11 of the Status register is explicitly reserved. Bits 10-8 are reserved implicitly.

Field attributes

There are several special field attributes that apply to writing a specific field within a register when you are not concerned with writing the other fields that happen to be in the same register. In this situation, the other fields all need a default write-value. A named or reserved field may have a write attribute of preserve or fixed. A preserved field needs to preserve the current value when written. A fixed field has an integer value that must be written to it. Omitted fields are fixed at 0. Reserved fields are fixed at 0 unless otherwise stated.

```
  [15:12] field1 write_attrib=preserve;
  // field 11:9 ommitted, fixed(0)
  [8:5]  reserved; // fixed(0)
  [4:1]  field2 write_attrib=fixed(3);
```

A preserved field will require an IO read of that register to gain the current value before the write can be done.

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 state components and fields. 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.

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

The preconditions of a readable register may not include the fields because their value is not known yet. See below for special pre- and postconditions on a readwrite register.

Conditions may be given as one boolean expression or several which are implicitly joined with an &&. As in C, a block of multiple conditions is surrounded by braces.

```
requires {
  Window==1;
  Busy==false;
}
```

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$0 && Window$\leq$6) to every pre- and post condition where Window is mentioned.

It is possible to write false conditions such as

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

however, these will be caught when the specification is checked for consistency since no IO call could satisfy these conditions.

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.

4.3 Syntactic sugar

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)
        && ... ;
```

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.

Conditions in a readwrite IO rule

Conditions in a readwrite IO rule may specify if they apply solely to the read or write. This is useful syntactic sugar for a register that has very similar but not identical rules for read and write access. Although the 3c509 did not need this, the SMSC LAN91C111 Ethernet device [8] uses one in the MMUCR register.

```
requires BSR_BANK == 2;
requires write MMUCR_BUSY == false
  || (CMD != 4 && CMD != 5);
```

In this example, both conditions apply to a register write while only the first condition applies to a register read. The default on a condition is both read and write. A write condition on a read-only IO rule has no effect and the same for a read condition on a write-only IO rule.

4.4 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 [4] 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 [3] are explained in a nearly identical fashion. [2]

5 Compiling

A Laddie file must have a .lad extension.

```
Laddie foo.lad
```

Successful compilation will produce the following output at the prompt

```
Testing tree formation...
Static semantic checks...
Code generation...
```

and produce these files

```
fooIO.clh
fooMacro.clh
fooNative.clh
```

A .clh file in Clay serves the same purpose as a .h file in C^{++}. The IO functions with all their pre and post conditions are located in fooIO.clh. Macros to call these functions are located in fooMacro.clh. The Readme file contains some #defines that a user will need to use the generated Clay files. Lastly, the C^{++}code that makes the actual IO calls is located in fooNative.clh.

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

5.1 Compilation options

The Laddie compiler accepts the following flags:

- **-translation or -t** This flag will allow you to double-check the translation into correct Clay syntax by producing some test files.
- **-consistency or -c** This flag will allow you to check a specification for consistency.
- **-all or -a** This flag is the equivalent of using all other flags.

The translation flag

You will need to copy the native.h file to your directory prior to running the test. To run the translation test, type

```
clay fooTest.clay
g++ fooMain.cc fooTest.cc
```

This test checks that the generated Clay syntax is correct and all types have been declared. It does not check any pre- or postconditions. There will be no output on a correct translation. In general, there is no need to run this test unless you suspect you have found a bug in the Laddie to Clay translation.

The consistency flag

This flag will generate a check for all register rules that have test values declared for all state components and fields in the precondition. The input values are declared at the end of the register rule as follows

```
port 0x0E {
  name = Command;
  ...
  test_input Busy=false, command=0, argument=0;
}
port 0x08 {
  name = RxStatus;
  ...
  test_input Busy=false, Window=1;
}
```

The input needs to include a possible set of values for all fields (on a write) and all states mentioned in the precondition and all old states mentioned in the post condition. If any necessary test values are missing, the consistency test will use 0 and, possibly falsely, report a bad pre- or postcondition. There is currently no static checking done on these values prior to the consistency test. (If you are testing a read function with no states in the precondition, give a test value for some other state to force the test case to be generated).

This flag generates test cases which are put in the /tmp directory. If you have no /tmp directory, unknown errors will occur during compilation and you are advised not to use this flag. A test script is also generated (in the directory of your laddie file) and named foo_testscript. You will need to copy the std.clh file to your directory prior to using the script. The script requires that you have perl installed in /usr/bin/perl. If it is located elsewhere, edit the first line of the script to indicate its location. The script also requires that you have clay installed so that it can be invoked with

```
clay foo.clay
```

at the prompt.

To run the script, type

```
foo_testscript
```

or

```
perl foo_testscript
```

When run, the script will test each IO function header one by one. For each header, it will report errors using the format

```
fooreadname1.clay : Impossible postcondition
foowritename2.clay : Impossible precondition
```

The above output should be interpreted as:

- The name1 register of the foo specification has an impossible postcondition on read access.
- The name2 register of the foo specification has an impossible precondition on write access.

If both the pre- and postcondition are inconsistent in a register rule, then only one will be reported and you will need to re-run it to make sure you fixed all errors. Warning: if your input test values are bad then you may see a false alarm.

How does the consistency test work?

In Clay, function preconditions are checked for each call of the function. For normal functions (ex. driver functions), Clay also checks that the precondition and the function body imply the postcondition. However, for native functions (ex. IO functions), the function body is in native code and performs side effects where Clay cannot see them. Therefore, we assume the postcondition rather than checking it.

The consistency tests will catch errors where no inputs could satisfy a precondition and errors where assuming a postcondition will introduce false into the system.

To test the consistency of pre- and postconditions, we call each IO function on input values you supply using the test_input keyword. Then we call a function with an impossible precondition. If your precondition is impossible, Clay will be unable to conclude it from your input. If your postcondition is impossible, Clay will assume it and introduce false into the system and the second function will then pass the typechecker. If both of your conditions are possible, they will pass the typechecker and the second function will fail.

5.2 Generated Clay files

We suggest you read the Clay User's Guide if you wish to understand this section. Here is the Clay IO function stub for the RxStatus register from Figure 1.

```
native exists [u32 E0out, u3 G0, u11 H0, u1 I0, u1 J0; true && (E0out==H0)]
@[StateType_Base[D,A], StateType_Busy[D,D0in], StateType_RxQUsed[D,E0out],
  StateType_Window[D,F0in], Int[G0], Int[H0], Int[I0], Int[J0]]
read_RXStatus
   [u32 A, u32 D, u1 D0in, u32 E0in, u32 F0in;
    true && ((F0in>=0 && F0in<=6)) && ((F0in==1 && D0in==0))]
(Int[A] addr, StateType_Base[D,A] stateVar_Base,
 StateType_Busy[D,D0in] stateVar_Busy,
 StateType_RxQUsed[D,E0in] stateVar_RxQUsed,
 StateType_Window[D,F0in] stateVar_Window);
```

Laddie has generated a type for each state component.

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

and a type for the base address to which the register offset is added

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

Both types are indexed first by D, the memory location of the 3c509 driver data. This prevents passing in the wrong state components if there are multiple 3c509 cards present. It also ties the base address to this device. Unlike the logical device state components, such as interrupts, 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.

There is a matching C^{++}function for every Clay 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 the 0-bit values are erased leaving us with a function that takes the base address and returns the two 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 has been done for each field [i:j] using

$$(\text{value} \;>>\; i) \;\&\; \sum_{n=0}^{j-i} 2^n$$

For a write function such as the Command, the bit twiddling formula for each field [i:j] is

$$(\text{value} \;<<\; i) \;\&\; \sum_{n=i}^{j} 2^n$$

so the C^{++}function for this register is

```
inline void write_Command(unsigned long addr,
                          unsigned long command,
                          unsigned long argument)
{
    command = (command << 11) & 63488; // bits 11:15
    argument = (argument << 0) & 2047; // bits 0:10
    outw(addr + 0x0E, command + argument);
}
```

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, \
    s_WindowSTATE##2, code, RxBytes, IC, ER) = \
  read_RXStatus(IOADDR, s_baseSTATE, s_BusySTATE, \
    s_RxQUsedSTATE, s_WindowSTATE); \
  s_BusySTATE = s_BusySTATE##2; \
  s_WindowSTATE = s_WindowSTATE##2; \
  s_baseSTATE = s_baseSTATE##2;
```

```
#define W_Command(command, argument) \
let [] (s_baseSTATE##2, s_StatsEnabledSTATE, \
    s_BusySTATE, s_WindowSTATE) = \
  write_Command(IOADDR, s_baseSTATE, s_StatsEnabledSTATE, \
    s_BusySTATE, s_WindowSTATE, command, argument); \
  s_baseSTATE = s_baseSTATE##2;
```

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

```
outb(ioaddr + COMMAND, STATS_ENABLE | 0);
```

with its respective safe IO call

```
W_Command(STATS_ENABLE, 0);
```

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 [10].

7 Related Work

The languages that have the most in common with Laddie are Devil [5, 7], NDL [2], and Hail [9, 11]. 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.

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.

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 [6, 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.

8 Status

Currently, all features mentioned above are supported except for read_value and write_value and the enum in a read. These will compile but do not effect generated code.

The enum in a write currently produces an extra pre-condition on writes and acts as a local variable within the conditions. Eventually, the enum values may produce #defines for general usage. A naming convention such as the one used in Hail or globally unique enum value names will have to be considered first.

The pre- and post conditions specified in Laddie produce compile time checks in Clay. A driver programmer will need to write the body of the driver in Clay and then compile it. Clay's compiler will do all the compile time checking and will inform the programmer if any run-time checks are necessary.

Future work includes adding support for read_value and write_value as well as address descriptions which define the bus access methods. In the long term, we envision creating compilers from C to Clay so that writing and verifying the remainder of the driver is an easier process.

9 Availability

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

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

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

[5] 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.

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

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

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

[9] 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.

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

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