The Evolution of Erlang Drivers and the Erlang Driver Toolkit

Scott Lystig Fritchie
Snookles Music Consulting
Minneapolis, Minnesota, USA
slfritchie@snookles.com

ABSTRACT

Erlang is gaining a reputation as a good language for rapid prototyping, but one area where its reputation is weaker than those of traditional scripting languages is extensibility. Erlang is actually fairly easy to extend, but the learning curve is steep. To reduce the time necessary to create Erlang extensions, called "drivers," for existing code libraries written in C, the Erlang Driver Toolkit (EDTK) was developed. Its code generator can produce all or nearly all of the Erlang and C code required to implement both major types of Erlang drivers. Although it is still under active development, EDTK has already proven to be a time- and effort-saving tool for creating robust, full-featured driver extensions for three well-known Open Source C libraries.

General Terms

Erlang, language extensibility, functional programming, code generation

1. INTRODUCTION

Erlang receives a lot of well-deserved recognition for supporting development of fault-tolerant, distributed, soft real-time applications. It also has a growing reputation as a rapid prototyping environment. The combination of these traits creates something remarkable: rapid development of prototypes that are good enough for use in production systems.

There is one area, however, where Erlang's reputation for quick development of effective code is not very good: development of "drivers" used to extend the base functionality of the language. The Erlang Questions mailing list [4] has received numerous queries for extension drivers for Dialogic and other telecom interface cards, for databases such as Oracle and Sybase, for `mmap()` and System V shared memory, for the message-passing library MPI, and many others. A small fraction of these people have actually implemented the wished-for driver themselves. At least one such motivated list subscriber, however, halted development of his driver due in part to the effort required by the initial implementation [8].

ACM SIGPLAN Erlang Workshop '02 Pittsburg, PA USA

Writing an Erlang driver is well within the reach of a modestly skilled Erlang programmer who is also familiar with C or some other programming language. However, it takes a well-motivated individual to locate the documentation on writing drivers, find and study the source code for a variety of existing drivers, and so on. The Erlang Driver Toolkit (EDTK) was begun by the author as a collection of Erlang and C tools to reduce the amount of time and effort spent writing code that is inevitably similar for most drivers.

EDTK has developed into a code generator capable of producing all or almost all of the Erlang and C code necessary to create Erlang drivers for existing C libraries. The EDTK compiler can implement drivers using both techniques supported by the Erlang virtual machine: as an external operating system process or as a dynamically-loadable library. EDTK has been used to generate functional and robust drivers for three well-known Open Source libraries: libnet (a network packet creation and transmission library), libpcap (a network packet capture and filtering library), and Berkeley DB (a commercially-supported embedded database).

Section 2 makes a small survey of how three popular scripting languages provide their well-known extensibility and contrasts them with Erlang's extensibility mechanism. Section 3 examines related work in the area of interface code generators and extensibility. Section 4 explores some of the issues that make extensibility a tougher problem for Erlang than many other languages. Section 5 presents a high-level overview of the EDTK driver code generator and discusses why it was developed the way it was. Section 6 looks at how EDTK implements a small example driver. Section 7 summarizes the experience the author has had using EDTK to develop drivers for non-trivial third-party C libraries. Section 8 suggests areas for future development of EDTK. Sections 9 and 10 present conclusions and describe how to obtain more information about EDTK.

2. LANGUAGE EXTENSIBILITY

Many programming languages become popular not only because they provide a useful core of features, but also because they have mechanisms for extending the language beyond what their designers originally intended. The extension mechanisms of three languages, Tcl, Python, and Perl,

are briefly examined in order to provide context for comparison to Erlang's extension mechanism, the "port."

2.1 Extensibility Mechanisms of Other Languages

2.1.1 Tcl

Tcl was originally designed as a simple command interpreter to be embedded in other applications. Tcl's fundamental data type is the string: Tcl objects are converted to and from strings as necessary.

Additional commands are added to Tcl by calling `Tcl_CreateCommand()`, which registers a callback function with the interpreter. Each callback function receives the command's arguments as strings via a convention that should look familiar to all C programmers: `(..., int argc, char *argv[])`. The callback function must parse these strings before calling the "real" extension function. The result(s) are returned to the interpreter as a string or as a Tcl object, using functions such as `Tcl_SetResult()` and `Tcl_SetObjResult()`.

2.1.2 Python

The C extension code of a Python extension module is responsible for data conversion to and from Python structures. The module's initialization function calls `Py_InitModule()` to register C extension functions with the interpreter. In the simplest and most common case, this registration will require the interpreter to pass all arguments to the extension function as a tuple. The extension function will use `PyArg_Parse()` to parse the tuple and copy its values into local C variables. After the "real" extension function is called, the results are typically sent back to Python using `Py_BuildValue()`.

2.1.3 Perl

Perl's popularity as an extensible language rose dramatically with the introduction of the XS language, which specifies an interface between Perl and an existing C library. XS functions can directly manipulate underlying Perl data structures: scalars, hashes, and arrays. Macros are provided for creating these structures, converting them to and from C data types, and changing their reference counts. XS also provides facilities such as custom initialization of function arguments and the ability to insert verbatim C/C++ code into the XS preprocessor's output when XS alone is insufficient to make the interface work.

The XS preprocessor generates much of the glue code required for the extension, including registering the extension functions with the interpreter. If the C library's API is simple or even moderately complex, no additional C code is typically required to complete the extension interface. In contrast, both Tcl and Python extensions require a fair amount of glue code to be written manually or by a code generator.

2.2 Erlang's Port Mechanism

Erlang extensibility mechanism is called a "port." The virtual machine copies data through the port to and from the port's "driver," which actually implements the language extension.

The port identifier data type is used in a manner similar to the process identifier data type. Messages can be sent to a driver through a port by using the same operator, `!`, used to send messages to regular Erlang processes. Messages sent by drivers to Erlang are also received using the same operator, `receive`. Messages sent in either direction are restricted by convention in order to simplify the interface.

2.2.1 Pipe Drivers

The first implementation of a port connects the virtual machine to an external operating system process via a pair of pipes, enabling bi-directional communication. The Erlang side sends a message to the driver in the form `Port ! {self(), {command, Message}}`, where `Message` is a list of bytes to be sent to the external process. The external process side reads `Message`'s bytes from the pipe, parses them, then sends the driver's results as a formatted stream of bytes via the other pipe. The virtual machine places those bytes into the mailbox of the port's owner process using the tuple `{Port, {data, ListOfBytes}}`. This completes the illusion that a port is simply a special kind of Erlang process.

"Pipe drivers" are flexible. The external process can be implemented in C or any other language that can perform I/O on standard input and output file descriptors. Even a Bourne shell script can implement a simple driver. The separation of processes protects the virtual machine from any bug present in the driver. The tradeoff is performance: due to extra data formatting and copying and due to operating system process context switching, the cost of communicating with the external process is high.

2.2.2 Linked-in Drivers

To reduce communication overhead experienced with pipe drivers, a "linked-in" driver API was developed.[1] This API permits the driver's code to execute within the virtual machine's process context. It also preserves the illusion to Erlang that ports are almost like other Erlang processes; therefore, data sent to and from the driver still requires serialization, just like pipe drivers do.

To further lower driver overhead, two other innovations have been helpful. One, drivers can return data using the binary data type (an untyped chunk of bytes) instead of lists of individual bytes. Binaries make both kinds of drivers more efficient because they avoid creating and traversing lists that may be hundreds or thousands of elements long. Two, linked-in drivers can now create arbitrarily-formatted Erlang terms and send them directly to an Erlang process, bypassing the serialization step.

Linked-in drivers compromise safety for the sake of speed. They are much riskier to use than pipe drivers: a driver bug can crash the entire virtual machine and all applications running within it. On the other hand, the latency of communicating with a pipe driver can be 10–25 times greater than with a linked-in driver.

[1]This term is used to describe C drivers that are statically-linked to the virtual machine as well as dynamically-loaded drivers.

3. RELATED WORK

Automatic code generation tools have been around for almost as long as programmers have. Two obvious reasons for the popularity of these tools are speed and correctness. It is typically much quicker to write a specification for input to a code generator than it is to write the code that would otherwise be created by the generator. Code generators excel at writing the kind of code that is repetitive; programmers find writing such code tedious, boring, and error-prone. Also, it is typical to save time and to improve code quality by debugging a code generator once rather making multiple passes through manually-written code.

Code generators are frequently used today for tasks such as creating interfaces to database systems and for creating graphical user interfaces. Discussion here is limited to tools used for language and/or application extensibility.

3.1 RPC Tools

Many code generating tools have been developed for use in inter-process communication and RPC (Remote Procedure Call) environments, including those used in applications using Sun Microsystem's RPC and the Object Management Group's CORBA. These tools generate client and server stub functions as well as the serialization code required to carry data structures across network transport. As such, they perform several tasks that writing an Erlang driver also requires, but their network-centric architecture make them difficult to adapt to a pipe driver's much simpler requirements. Furthermore, they cannot provide much assistance, beyond serialization, for linked-in driver development.

3.2 SWIG

SWIG [1] is a popular Open Source tool for generating code specifically for language extensions. Using a common specification file, SWIG generates all of the glue code required to create extensions for many languages, including Guile, Java, Perl, Python, Ruby, and Tcl.

The specification file conforms largely to ANSI C/C++ syntax, which can greatly reduce the amount of effort necessary to create a SWIG specification. For a simple library, the header file defining its data structures and function prototypes can be used without modification. SWIG is designed to be a complete glue code generator: there is rarely need to manually write any additional glue code.

SWIG's capacity for generating language extension glue code is quite remarkable. Its current release is probably capable of generating much of the code that EDTK's GSLgen templates currently create. The main reason for not using SWIG for EDTK's generator was time: it was unclear how many changes to the SWIG specification file syntax would be required to support Erlang drivers and thus difficult to estimate how many man-months of effort it would take to develop a prototype.

3.3 IG

IG is an Erlang driver generation tool originally distributed with Erlang. Its specification file also bears a strong resemblance to ANSI C data and function declaration syntax. It generates pipe driver glue code for the C side as well as serialization code for both the C and Erlang sides. Though no longer maintained by Ericsson, IG's original author has publically released its source code [15].

```
# Copy a file
def filecopy(source,target):
        f1 = fopen(source, "r")
        f2 = fopen(target, "w")
        buffer = malloc(8192)
        nbytes = fread(buffer,8192,1,f1)
        while (nbytes > 0):
                fwrite(buffer,8192,1,f2)
                nbytes = fread(buffer,8192,1,f1)
        free(buffer)
```

Figure 1: Multiple-assignment in Python using a SWIG-generated interface

```
%module fileio
FILE *fopen(char *, char *);
int fclose(FILE *);
unsigned fread(void *ptr, unsigned size,
               unsigned nobj, FILE *);
unsigned fwrite(void *ptr, unsigned size,
                unsigned nobj, FILE *);
void *malloc(int nbytes);
void free(void *);
```

Figure 2: SWIG specification for standard C library functions

4. ADDITIONAL CONSTRAINTS

Erlang and its runtime environment impose constraints on driver authors and would-be driver generator authors that many other languages do not have. These constraints are examined here in order to explain why off-the-shelf solutions to extensibility problems faced by other languages, such as the languages mentioned in section 2.1, cannot be easily applied to Erlang.

4.1 Single-Assignment Semantics

Erlang's single-assignment semantics create the single largest obstacle to a simple marriage of Erlang and C code. C does not have such a constraint, so C libraries never worry about it. A linked-in Erlang driver must avoid multiple-assignment in every C function that uses pass-by-reference arguments.

Figure 1 demonstrates the problem single-assignment poses to Erlang drivers. The Python code, taken from [14], uses an interface generated by SWIG (see specification in Figure 2) to give Python direct access to several standard C library functions. Clearly, a direct translation of what SWIG does in this example cannot be used by Erlang: the contents of `buffer` are repeatedly clobbered inside the `while` loop. A linked-in driver implementing `fopen()` has a limited number of implementation choices. First, the driver can allocate a new immutable binary for each buffer's worth of data read by `fread()`. Second, the driver can hide any mutable data structures inside the port itself: another driver function must be called to retrieve an immutable snapshot of the data.

4.2 Serialization of Data

All data sent between Erlang and a pipe driver must be serialized:[2] since the virtual machine and driver processes are connected by a pair of pipes, there are few other options. Even if the two were to communicate using a network protocol or through the file system, their internal data structures would still require serialization. An ideal Erlang driver generation tool should create the required data serialization code. As an additional feature, linked-in drivers should be able to send arbitrary terms back to Erlang instead of serializing its results like pipe drivers must.

RPC-centric generators must address serialization because of their use of network transport. However, their serialization code is typically burdened by other assumptions and constraints, such as memory and network interface management. Using their serialization code outside of their broader frameworks can be as, or more, cumbersome and time-consuming as writing that code yourself.

4.3 Fault Tolerance

Erlang is well-suited for fault-tolerant, high availability application programming. However, a fault-tolerant design can be crippled by resource leaks, such as memory or file descriptor leaks, as surely (though not as quickly) as a divide-by-zero bug. The Python example in Figure 1 is illustrative here. If this example were instead implemented in Erlang, then depending on where the Erlang process crashed (or was killed by an outside party), the driver may leak up to two file descriptors and one hunk of memory.

Resource leak problems can be magnified by common Erlang programming practices that encourage coding only for the common case. It is not unusual to simply let an Erlang process crash when it encounters an error or exception. The OTP supervisor process behavior is used specifically for managing such crashes and to restart dead processes.

If an application must run non-stop for months or years at a time, resource leaks caused by bugs and unanticipated error conditions cannot be tolerated. An ideal Erlang driver generator should provide facilities for managing scarce system resources used by the driver and be able to release them if the owner process should die.

4.4 Thread Management

Erlang release R7B introduced an asynchronous driver mechanism. The virtual machine can maintain a pool of worker threads for use by any linked-in driver. All core virtual machine activity still takes place in the main thread, but linked-in drivers have the discretion to schedule work for execution by the virtual machine's worker thread pool. Worker threads cannot directly access any of the main thread's data structures, but they are free to block for arbitrary periods of time: the main thread remains independent and can continue executing Erlang processes.

The asynchronous driver mechanism is already used by parts of the standard Erlang distribution. For example, access to the host operating system's file system is done via the `efile_drv` driver, which can execute all potentially blocking file operations inside a worker pool thread. If the worker thread pool is not enabled, the driver will automatically execute those functions in the main (and only) thread.

An ideal driver code generator should provide the flexibility to specify whether a driver should execute a function in the main thread, utilize the thread worker pool, or use a private thread managed by the driver itself. Such flexibility can be used to achieve several goals:

- Manage libraries that contain a mix of thread-safe and non-thread-safe member functions.
- Manage unpredictable resources, such as hardware devices and lock managers.
- Take advantage of platforms with multiple CPUs without running multiple Erlang virtual machines. The virtual machine is still constrained to run in a single thread and therefore can only fully utilize a single CPU, but computation-intensive driver code can run on multiple CPUs in parallel with potentially little effect on the virtual machine's thread.

5. THE EDTK DRIVER GENERATOR

The Erlang Driver Toolkit started as a very modest project: create a collection of library functions, both in Erlang and in C, to ease the task of writing drivers for third-party C libraries. Development of those helper functions went quickly, and they were indeed significant time-savers.

But as these helpers were used for drivers for several different libraries, it became clear that even with the assistance of a capable library, writing drivers can be very tedious work. The prospect of automating most of the code generating process became extremely attractive. The new goal became writing a code generator, since a lot of a driver's C code (both pipe and linked-in drivers) and Erlang code is boilerplate, common to most drivers.

5.1 Design Goals

The EDTK code generator has grown in fits and starts, which is not surprising for a part-time programming project. It would be disingenuous to claim that its design goals and constraints were fully specified before the first line of code was written. With the clarity of hindsight, however, these are the generator's major design goals:

1. **Quickly develop a prototype.** Refactoring or even completely rewriting the tool is likely, so learn as much as possible from a usable prototype first. To paraphrase Brooks, it's worthwhile to grow a program even if you eventually throw it away [3].
2. **Create a time-saver, not a labor-eliminator.** The intent is to automate the most tedious and error-prone tasks associated with creating an Erlang driver. Generating *all* required glue code would be an extra bonus.
3. **Support both driver types.** The generator should be able to create code for pipe and linked-in drivers. Furthermore, linked-in drivers should be able to utilize the asynchronous worker thread pool.

[2]Section 2.2.2 describes an alternate method for linked-in drivers to send data back to Erlang.

Input XML file

```
<?xml version="1.0"?>
<tree>
  <entity name="World">
    <greeting type="Hello"/>
  </entity>
  <entity name="Planet">
    <greeting type="Salutations"/>
  </entity>
</tree>
```

Input schema file

```
.ignorecase = 0
#!/bin/sh

.for entity
.  for greeting
echo "$(.type), $(entity.name)!"
.  endfor
.endfor
```

Output file

```
#!/bin/sh

echo "Hello, World!"
echo "Salutations, Planet!"
```

Figure 3: Example GSLgen output

5.2 Quick Prototyping with GSLgen

iMatix Corporation has released a number of code generation tools with open licenses. One of those tools, GSLgen [6], is a general-purpose file generation tool. Given inputs of an XML file [2] and a schema template file, the schema template tells GSLgen how to traverse the XML tree and what to output.

GSLgen is similar to PHP [13], Microsoft Active Server Pages [7], and other "server-side parsed" World Wide Web content management tools. Instead of serving HTML files verbatim, the Web server scans HTML template files for specific tags or keywords and interprets their contents as a scripting language. The result of executing those instructions is inserted into the HTML file in place of the original tags before transmission to the client.

A GSLgen schema can be used to generate HTML files, C or C++ files, or whatever a developer wishes. Figure 3 gives a tiny example of an XML file, a GSLgen schema file, and the corresponding GSLgen output. As noted in the introduction to section 5, much of an Erlang driver's code is boilerplate. GSLgen appeared to be an excellent tool for generating driver glue code from a set of templates, so it was chosen to create the EDTK code generator.

5.3 EDTK XML File Structure Overview

This section discusses each of the element types found in an EDTK XML specification file and how attributes of one element are cross-referenced to other elements. See Figure 4, which depicts the element hierarchy of an EDTK XML file.

5.3.1 The func, arg, and return Elements

There is a `func` element for each C library function the driver is capable of calling. It may have multiple `arg` child elements that must appear in the order in which they will be called by both the Erlang and C sides of the driver glue code.

Attributes of the `arg` element specify the argument's C type, describe how it should be serialized when sent to the driver, and other characteristics of the argument. For example, an argument may be omitted from the Erlang side or the C side of the function's signature by adding the attributes `noerlcall="1"` and `noccall="1"`, respectively.

A single `return` child element is mandatory for each `func` element. Like the `func` element, it specifies the C type of the return value (even if the function's return value is void) and how that value, containing the function's results, should be returned to Erlang.

An optional pair of attributes, `expect` and `expect_errval`, can set the criteria by which the return value should be considered normal or in error; this expectation definition can affect the format of the tuple returned to Erlang. The optional `valmap_name` and `xreturn` attributes cross-reference elements of types `valmap` and `xtra_return`, respectively.

5.3.2 The valmap Element

Short for "value map," a `valmap` element is used by the generator's C template to maintain a mapping between C data values and Erlang terms. These elements are most useful for memory pointers, file descriptors, and anything else that the driver needs to deallocate, close, or otherwise clean up when the driver is closed by a `port_close/1` call or by abnormal process termination.

A value map term is opaque as far as Erlang is concerned. It is represented by a tuple such as `{valmap_NAME, Index}`. The driver uses `Index`'s value as the index into an internal value map table.

A value map makes it very difficult for Erlang to pass invalid values into the driver. When combined with expectation attributes on a `return` element, the C template can guarantee that only valid values can be inserted into a value map.

The order in which `valmap` elements appear in the XML file is important. When the driver's `stop` entry function is called, value maps will be closed and deallocation/cleanup functions will be called in the order in which the `valmap` elements appear in the XML file.

In the current implementation, only a function's return value can be inserted into a value map. For functions that do not follow this convention, a small wrapper function must be created manually to rearrange the API.

5.3.3 The xtra_return Element Hierarchy

Early versions of the EDTK generator templates could only return a single value (an atom, a binary, or an integer) to Erlang. It became apparent that more flexibility was needed.

The `xtra_return` element hierarchy defines a 1-to-1 mapping of XML elements to the values returned to Erlang in an

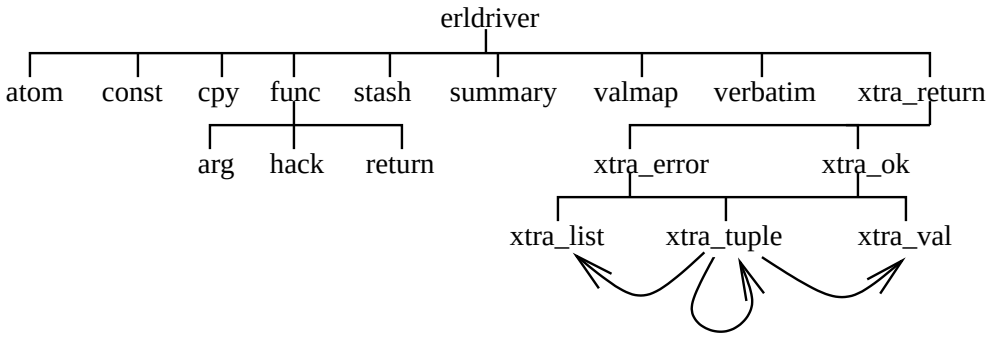

Figure 4: Diagram of EDTK XML element hierarchy.

arbitrarily-formatted tuple. Defining a 1-to-1 mapping for a variable-length list is not feasible, so the `xtra_list` element specifies the name of the C function that will create the list.

5.3.4 *The atom, const, cpy, hack, stash, summary, and verbatim Elements*

These elements play roles in initializing various driver defaults, adding copyright and license notices, and inserting verbatim code in strategic places in the C and Erlang templates.

6. EXAMINATION OF A SAMPLE DRIVER

The XML file shown in Appendix A is a complete EDTK XML specification file for creating a driver for a very small portion of the standard C library.[3]

The EDTK driver generator exposes all of the underlying C function's arguments and return value (if non-void). If a traditional Erlang interface for the driver is desired, for example to use a list of atoms to specify options to a C function that utilizes a bitmask for those options, it must be written separately.

The following subsections will focus on linked-in drivers. Significant differences with the pipe driver will be addressed in section 6.6.

6.1 Step 1: Erlang side calls the port

All data passed from Erlang to the driver port must be serialized, regardless of whether the port interfaces with a pipe or linked-in driver. Like the data passed to other Erlang I/O functions, this data must be in the form of what is hereafter called an "I/O list."

An I/O list is a list comprised of binary terms, integer terms with values between 0 and 255, and other I/O lists. For example, `[0, 1, [[2, []], <<254, 255>>]]` is an I/O list that is five bytes long. A single binary, such as `<<1>>`, is also considered a valid I/O list: there is no need to enclose it in a list like `[<<1>>]` to make it a valid I/O list.

The Erlang function for each library function serializes the function's arguments into an I/O list. The first byte of the I/O list encodes the C library function number. Function arguments follow, encoded following these rules:

- Integers, including value map indexes, are serialized according to the XML attributes on the argument element which specify whether the integer is signed and how many bits to use. If unspecified, the integer is assumed to be a 32-bit unsigned value.
- I/O lists first have their total length in bytes calculated. This size is encoded as a 32-bit unsigned integer. The lists's contents follow immediately afterward.
- Integers greater than 32 bits and all floating point values are not currently supported.

Pattern matching on the atom member of {`valmap_NAME, Index`} verifies that the term is of the proper type for a particular function argument. The `Index` integer is what is actually serialized and sent to the driver.

By definition, C arrays of any data type are stored in a single contiguous piece of memory. The underlying data in an I/O list is not contiguous, unless it is a single binary term. Since something must make arrays contiguous to match the driver C function's expectation, the Erlang side performs this task using `list_to_binary/1`.

The resulting I/O list is then sent to the port using `port_command/2`, which will send the data to the pipe driver's external process using the `writev()` system call or invoke the linked-in driver's `outputv` entry function. For linked-in drivers, using the `outputv` entry function avoids making an extra data copy that `port_control/3` and `port_call/3` require.

6.1.1 *Items of note in* `sample.xml`

The `malloc_int/2` function calls the same C function as `malloc/2` does, but the handling of the return value is quite different. `malloc/2` returns a value map, whereas `malloc_int/2` demonstrates returning a pointer as an unsigned integer. This pointer-as-integer return style is how pointers are handled by Perl's XS, Python, and extensions generated by SWIG. The `malloc_int/2` function demonstrates this method can also be supported.

[3]Although a symbol for `lstat()` is often found in libc, it is not officially part of the standard C library.

Both `fopen()` and `lstat()` have arguments that expected to be NUL-terminated. The `nulterm="1"` attribute will cause the Erlang template to append a NUL byte to these arguments before serializing them.

The `size` argument has been removed from `fwrite`'s signature to demonstrate the use of a `hack` element to hard-code `size`'s value to be 1.

The `lstat` definition uses the attribute `noerlcall="1"` to remove the `struct stat` argument from the Erlang version of the function. The C driver code provides the necessary `struct stat` buffer; there is no need for Erlang to have knowledge about it.

6.2 Step 2: C side deserializes arguments

The C template's implementation of the driver's `outputv` entry function is responsible for three things:

1. Deserialize the arguments and copy them into a `callstate_t` structure.
2. Determine which library function should be called.
3. Determine if that function should be executed by the virtual machine's main thread or by some other thread.

The deserialization task is straightforward, except in the case of value maps. The value map index, passed in serialized form, is decoded and used an index into the appropriate value map table. The value stored there is checked for validity and, if valid, the real data value is copied to the appropriate C data structure.

The library function to be called is specified by the first byte in the serialized call data. The choice of which thread to use is made according to the driver's XML specification.

If any arguments are improperly formatted, `outputv` will return the message {`Port, error, badarg`} to the caller.

6.2.1 *Items of note in* `sample.xml`

A special XML element, called `hack`, is used by `fwrite`'s description to initialize the argument that was intentionally omitted from the Erlang function's signature. A `hack`'s code may be arbitrary C code.

The variable `c` is a pointer to a `callstate_t` structure used to store all data used by the C extension function. Two substructures, `i` and `o`, are used input and output values, respectively. This is one of several examples that demonstrates that some knowledge is necessary of how EDTK creates driver glue code.

6.3 Step 3: Executing the extension function

Step two has already prepared all input arguments and storage for output data. Executing the C extension function is straightforward. The other work done in this step is evaluating the return value expectation condition, if one is defined.

If the expectation test evaluates true, the basic status atom `ok` is returned to Erlang; otherwise, `error` is returned. If the expectation is false, the expectation error value is captured immediately after executing the C function by the same thread that called the function, ensuring that the correct thread-specific value, such as `errno`, is returned to Erlang.

Expectation tests should be used whenever value maps are used. Without an expectation condition, an error value (or worse, garbage) may be stored in the value map table and then used by subsequent driver calls.

6.3.1 *Items of note in* `sample.xml`

The `malloc_int` description has no expectation, so the Erlang side must determine if the C function succeeded or failed.

The calculation of the error status value is complicated by `ferror()`'s limited utility. It is possible, though not likely, that `fread()` or `fwrite()` failed for reason other than a failed system call, so `errno`'s value may not explain what went wrong. So a value of 0 is returned when end of file is reached and -1 otherwise.

6.4 Step 4: Driver returns data to Erlang

All return values are sent to Erlang by the driver's `ready_async` entry function, regardless of the driver's type or thread usage. If a return value more complicated than an atom, binary, or integer is desired, a `xtra_return` element must be used to define the structure of the return term.

6.4.1 *Items of note in* `sample.xml`

The Erlang return value for `lstat` returns a handful of the data actually found in a `struct stat` structure: `st_mode`, `st_mtimespec`, and `st_size`. Returning only these structure members avoids problems of structure members not found in all operating systems as well as problems returning very large integers (see also section 6.5).

6.5 Step 5: Erlang receives result term from port

The current `receive` clause used to fetch the return value from the driver is intentionally very simple:

1. The driver always returns a 2-tuple or 3-tuple.
2. The first element is always the port's identifier.
3. The second element is either the atom `ok` or the atom `error`.
4. The third element, if present, is an arbitrary Erlang term.

There is no support currently for returning true unsigned 32-bit integers and "bignum" values (including floating point): they are not supported by `driver_output_term()`. One possible work-around is to return those values to Erlang as binaries using agreed-upon formatting, and they can be deserialized by the Erlang side after receipt.

6.6 EDTK Pipe Drivers

There are only two differences between linked-in drivers and pipe drivers that driver writers care about. First, pipe driver code executes in an operating system process separate from the virtual machine. Second, pipe drivers *must* serialize their output: linked-in drivers have an option to create terms and send them directly to an Erlang process's mailbox.

EDTK uses a program called `pipe-main` that hides these differences from the EDTK-generated dynamically-loadable library created for linked-in driver usage. It implements all the virtual machine's functions that an EDTK driver uses (primarily memory management functions).

An EDTK-generated driver pushes its output term onto a stack structure, then calls `driver_output_term()` to deliver the term. The `pipe-main` executable's version of this function pops values off this stack and serializes the data instead. Template code on the Erlang side deserializes the data before returning the original output term to its caller.

7. EXPERIENCE WITH REAL LIBRARIES AND APPLICATIONS

The EDTK driver generator has been used to create fully-functional drivers for several libraries. Each has triggered new EDTK features because of their variety of APIs and the data structures they manipulate. Libpcap and Berkeley DB have been particularly challenging to support, but they were chosen specifically for their complexity: if EDTK can successfully generate drivers for them, it will have the capability to support many other "real-world" libraries.

7.1 First Driver: simple1_drv

The `simple1_drv` driver is the first one developed using EDTK. It was created to discover the basic tasks that EDTK ought to do. As such, it is much more useful for finding bugs and acting as the target for most of EDTK's regression tests than something that a developer would use to do real work.

The functions implemented by the driver are a mix of functions written specifically for unit and regression testing, functions from the standard C library, and several UNIX system calls.

7.2 Libnet

Libnet [9] is a collection of platform-independent functions for constructing network protocol packets and transmitting them. Half of the library provides a platform-independent API for transmitting network data at ISO network layers 2 and 3 for Ethernet and IP, respectively. Libnet's encapsulation of operating system-specific minutiae of using so-called "raw" sockets is extremely useful.

The other half of the library contains functions for creating various types of network packets and headers. The power of Erlang's bit syntax makes this part of the library less appealing, so it is not currently implemented by the driver.

7.3 libpcap

Libpcap [10] provides a platform-independent interface for capturing network packets (also often called "packet sniffing"). Broadcast media, point-to-point interfaces, and loopback interfaces are all accessible via libpcap.

Libpcap's API glosses over many platform-specific details, but it cannot hide run-time differences that various UNIX platforms exhibit when running libpcap's code. The differences alone between platforms running Linux 2.2, which uses the "packet socket" mechanism, and FreeBSD 4.5, which uses the Berkeley Packet Filter (BPF) [11], are large and vexing. It is no surprise that this library creates several problems for a linked-in driver. A few examples include:

- Some platforms cannot guarantee that the libpcap's packet read timeout will occur until after at least one packet has been received, and some cannot guarantee that the packet read timeout will happen at all.
- FreeBSD's timeout behavior is further dependent on whether or not the application uses Pthreads or not.
- Several BSD platforms cannot use the `select()` system call on a BPF file descriptor and get meaningful results.

To be as portable as possible, the EDTK libpcap driver compromises by performing all potentially-blocking operations in a separate thread by using the virtual machine's worker thread pool. Unfortunately, this compromise in turn causes problems with the virtual machine's asynchronous work scheduling algorithm. For example, the `efile_drv` driver is used for local file system I/O. A file I/O operation may scheduled for execution by the same thread that is currently blocked on a libpcap driver operation, even if other threads in the work pool are idle.

7.4 ethbridge

Ethbridge is an Erlang application that uses both the libnet and libpcap drivers to turn an Erlang node with two or more Ethernet interfaces into an Ethernet bridge. ETS tables are used to keep track of which MAC addresses are reachable on each side of the bridge. Ethernet broadcast and multicast packets are automatically forwarded to the other side(s) of the bridge.

A small amount of testing has been performed using FTP to measure Ethbridge's speed when transferring a single large file across the bridge. The machine running Ethbridge is a PC with an AMD K6 233MHz processor and two Netgear Fast Ethernet cards running FreeBSD 4.5. Ethbridge can forward 920 packets/second using linked-in libnet and libpcap drivers and 540 packets/second using both as pipe drivers.

The `ping` program was used to measure single packet round-trip latency. Using linked-in drivers, round-trip times ranged from 2.11–4.41 milliseconds, averaging 2.57 ms ($\sigma = 0.32$ ms). Using pipe drivers, round-trip times ranged from 3.09–13.6 ms, averaging 3.89 ms ($\sigma = 2.6$ ms). Both tests leave the CPU 0% idle. Using FreeBSD's kernel-based bridging for comparison, round trip times ranged from 0.436–1.06 ms, averaging 0.868 ms ($\sigma = 0.11$ ms) while leaving the CPU 93% idle.

7.5 Berkeley DB

Berkeley DB [12] is an embedded, key-value database now developed and supported by Sleepycat Software. It is a complex library containing over 160 member functions, making it the biggest "real-world" driver yet attempted with EDTK. In addition to key-value tables, Berkeley DB also supports cursors, secondary key indexes, transactions (including nested transactions), deadlock detection, and a replication service.

The author's first attempt to create an Erlang driver for Berkeley DB took approximately 60 man-hours of effort to interface 36 members of the API. The Berkeley DB driver developed with EDTK is still under development. However, it currently supports 49 API member functions after approximately 24 man-hours of effort. The majority of that time was spent fixing generator template bugs and adding functionality necessary to support Berkeley DB's API, rather than actually crafting and debugging the XML specification.

Although Berkeley DB can be used safely in multi-threaded environments, it restricts which threads can use cursors and operate within transactions. This complexity is an excellent motivator for expanding and debugging EDTK's support for asynchronous linked-in drivers.

8. FUTURE WORK

At the time of this writing, EDTK's GSLgen-based code generator is still very young. Sometime in the not-so-distant future, it will be worth analyzing whether to continue using GSLgen or to shift development effort to SWIG or perhaps another open-source tool.

An EDTK-generated driver assumes that the Erlang process calling the driver will block waiting for the C extension function to finish. A general driver framework would permit Erlang to make multiple simultaneous calls to the driver and retrieve the results asynchronously. Also, several of the linked-in driver's API entry functions are not yet implemented, making it infeasible for use by drivers that wish to perform I/O managed by the virtual machine's I/O infrastructure.

EDTK does not yet provide any explicit support for extensions based on C++ libraries or support for drivers on non-UNIX platforms. Neither are a priority for this author, but support for C++ libraries and Microsoft Windows platforms undoubtedly are priorities for other EDTK users.

It would be wonderful if EDTK could support linked-in driver code that in turn call Erlang functions. As an example from the Berkeley DB library, the driver could use an Erlang function as a sorting callback function to determine if two keys are identical.

9. CONCLUSION

The EDTK started as a collection of useful functions and macros to assist the development of Erlang drivers. It quickly turned into a full-fledged code generator and has surpassed the design goals listed in section 5.1.

1. The GSLgen file generation tool is largely responsible for the rapid development of the project. Including debugging time, the EDTK code generator was capable of supporting simple call-by-value C functions after less than 30 man-hours of effort, most call-by-reference C functions after 70 man-hours, and using worker threads after 90 man-hours.

2. EDTK can now automatically take care of driver implementation details such as data serialization, output term creation, preserving single-assignment semantics, and utilizing asynchronous work threads. Drivers for simple C libraries require no additional code to be written manually. The Berkeley DB driver, with 49 of its approximately 160 API functions supported, requires an additional 150 lines of code to support the 5,150 line driver module created from a 1,000 line XML specification.[4]

3. The same shared library created by EDTK can be used directly as a linked-in driver or as a pipe driver run by the `pipe-main` program in an external process.

The value map mechanism has been particularly useful. A value map's automatic resource cleanup is vital to avoid long-term resource starvation, but it is also very convenient from the programmer's perspective: all cleanup tasks are taken care of by a one-line statement to close the port. In addition, it is very difficult to pass an invalid data value into a driver, greatly reducing the number of crashes caused by dereferencing invalid pointers given to the driver by Erlang.

10. AVAILABILITY

The Erlang Driver Toolkit is freely available under a BSD-style open source license. Drivers generated by EDTK have been tested on FreeBSD and Linux platforms; porting to other UNIX platforms that support Erlang is not expected to be difficult. Further documentation, full source code, and development community information is available at [5].

11. ACKNOWLEDGMENTS

I owe many thanks to everyone who reviewed this paper, especially in its early, ugly drafts: the workshop review committee, Francesco Cesarini, Chris Halverson, Mark Henning, Lennart Öhman, and Torbjörn Törnkvist. Nick Christenson went far beyond the call of duty with his helpful comments. Finally, thanks to Carolyn Lystig and Louise Lystig Fritchie for their copyediting finesse; any lingering errors are mine alone.

12. REFERENCES

[1] D. M. Beazley. SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++. In *4th Annual Tcl/Tk Workshop Conference Proceedings*. The USENIX Association, July 1996. See also: `http://www.swig.org/`.

[2] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0, 10 Feburary 1998. See: `http://www.w3.org/TR/REC-xml`.

[4] These figures include all comment lines.

[3] F. P. Brooks, Jr. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 20^{th} anniversary edition, 1995.

[4] The Erlang Questions electronic mailing list. See: `http://www.erlang.org/faq.html` for subscription information and access to list archives.

[5] S. L. Fritchie. EDTK: The Erlang Driver Toolkit. See: `http://www.snookles.com/erlang/`.

[6] GSLgen: a general-purpose file generator. See: `http://www.imatix.com/html/gslgen/`.

[7] S. Hillier and D. Mezick. *Programming Active Server Pages*. Microsoft Press, Redmond, Washington, 1997.

[8] S. Hinde. Personal correspondence.

[9] Libnet: a library constructing and injecting network packets. See: `http://www.packetfactory.net/projects/libnet/`.

[10] libpcap: a packet capture and filtering library. See: `http://www.tcpdump.org/`.

[11] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX Winter 1993 Conference Proceedings*. The USENIX Association, January 1993.

[12] M. Olson, K. Bostic, and M. Seltzer. Berkeley DB. In *USENIX Annual Technical Conference*. The USENIX Association, June 1999.

[13] PHP: an HTML-embedded scripting language. See: `http://www.php.net/`.

[14] SWIG 1.1 User Manual. See: `http://www.swig.org/doc.html`.

[15] Torbjörn Törnkvist. IG: The Interface Generator. See: `http://www.bluetail.com/~tobbe/ig/`.

APPENDIX

A. SAMPLE EDTK XML SPECIFICATION FILE

This appendix contains the contents of the file `sample.xml`, an EDTK XML specification file that describes the interface for several well-known standard C library functions and UNIX systems calls. Together with the definition for the `peek()` function (see Appendix B), the EDTK driver generator creates all of the code required to implement these functions as an Erlang driver. Both pipe and linked-in drivers are fully supported.

```
<?xml version="1.0"?>

<erldriver name="sample_drv" abbrev=""
           default_async_calls="1">

<summary>Examples for PLI2002 paper</summary>

<atom name="mini_stat"/>

<verbatim place="top_cpp_stuff">
#include &lt;stdio.h>
#include &lt;stdlib.h>
#include &lt;errno.h>
#include &lt;sys/stat.h>
</verbatim>

<func name="malloc" async_op="0">
 <arg name="size" ctype="size_t"/>
 <return name="ret_ptr" ctype="void *"
         valmap_name="ptr" valmap_type="start"
         expect="!= NULL" expect_errval="errno"/>
</func>

<func name="malloc_int" async_op="0" cname="malloc">
 <arg name="size" ctype="size_t"/>
 <return name="ret_ptr" ctype="void *"
         etype="integer" casttoint="1"/>
</func>

<func name="free" async_op="0">
 <arg name="ptr" ctype="void *"
      valmap_name="ptr"/>
 <return ctype="void"
         valmap_name="ptr" valmap_type="stop"/>
</func>

<func name="peek" async_op="0">
 <arg name="ptr" ctype="void *"
      valmap_name="ptr"/>
 <arg name="offset" ctype="int"/>
 <arg name="len" ctype="int" noccall="1"/>
 <return name="ret_char_p" ctype="char *"
         etype="binary" valtype="static_buf"
         val="c->o.ret_char_p"
         offset="0" length="c->i.len"/>
</func>

<func name="fopen">
 <arg name="path" ctype="char *"
      ser_type="binary" nulterm="1"/>
 <arg name="fmode" ctype="char *"
      ser_type="binary" nulterm="1"/>
 <return name="stream" ctype="FILE *"
         valmap_name="FILE" valmap_type="start"
         expect="!= NULL" expect_errval="errno"/>
</func>

<func name="fread">
 <arg name="ptr" ctype="void *"
      valmap_name="ptr"/>
 <arg name="size" ctype="size_t"/>
 <arg name="nmemb" ctype="size_t"/>
 <arg name="stream" ctype="FILE *"
      valmap_name="FILE"/>
 <!-- ferror() not very helpful, -1 good enough? -->
 <return name="ret_size_t" ctype="size_t"
         expect="> 0"
         expect_errval="feof(c->i.stream) ? 0 : -1"/>
</func>

<!-- Drop size arg, force it = 1 via hack element -->
<func name="fwrite">
```

```
 <arg name="ptr" ctype="void *"
      valmap_name="ptr"/>
 <arg name="size" ctype="size_t" noerlcall="1"/>
 <arg name="nmemb" ctype="size_t"/>
 <arg name="stream" ctype="FILE *"
      valmap_name="FILE"/>
 <return name="ret_size_t" ctype="size_t"
         expect="== (c->i.size * c->i.nmemb)"
         expect_errval="feof(c->i.stream) ? 0 : -1"/>
 <hack place="post-deserialize" type="verbatim">
        c->i.size = 1;
 </hack>
</func>

<func name="fclose">
 <arg name="stream" ctype="FILE *"
      valmap_name="FILE"/>
 <return name="ret_int" ctype="int"
         valmap_name="FILE" valmap_type="stop"/>
</func>

<func name="lstat">
 <arg name="path" ctype="char *"
      ser_type="binary" nulterm="1"/>
 <arg name="sb" ctype="struct stat"
      noerlcall="1" argtype="out"/>
 <return name="ret_int" ctype="int"
         xreturn="mini_stat"/>
</func>

<valmap name="ptr" ctype="void *"
        maxsize="32" initial_val="NULL"
        cleanup_func="free"/>

<valmap name="FILE" ctype="FILE *"
        maxsize="32" initial_val="NULL"
        cleanup_func="fclose"/>

<xtra_return name="mini_stat">
  <xtra_ok>
   <!-- Create {A, {B, C}, D} -->
   <xtra_val etype="tuple">
     <xtra_val etype="integer"
               val="c->o.sb.st_mode"/>
     <xtra_val etype="tuple">
       <xtra_val etype="integer"
             val="c->o.sb.st_mtimespec.tv_sec"/>
       <xtra_val etype="integer"
             val="c->o.sb.st_mtimespec.tv_nsec"/>
       </xtra_val>
     <xtra_val etype="integer"
               val="c->o.sb.st_size"/>
   </xtra_val>
  </xtra_ok>
  <xtra_error>
    <!-- EDTK provides __expect_errval -->
    <xtra_val etype="integer"
              val="c->o.__expect_errval"/>
  </xtra_error>
</xtra_return>

</erldriver>
```

B. ADDITIONAL C CODE FOR SAMPLE DRIVER

This appendix contains the contents of the file `sample-additional.c`, which defines the one function used by `sample.xml` that is not found in the standard C library.

```
char *
peek(void *ptr, int offset)
{
    return ((char *) ptr) + offset;
}
```

C. FILECOPY FUNCTION IN ERLANG

This appendix contains the contents of the file `filecopy.erl`. It uses the driver described by `sample.xml` in Appendix A to implement a file copying function in the style of the Python function shown in Figure 1. Despite using `malloc()` and `fread()`, the use of value maps preserves single-assignment semantics.

```
-module(filecopy).

-define(DRV, sample_drv).
-define(BUFSIZ, 8192).

-export([copy/2]).

copy(Src, Dst) ->
    {ok, Port} = ?DRV:start(),
    {ok, SrcF} = ?DRV:fopen(Port, Src, "r"),
    {ok, DstF} = ?DRV:fopen(Port, Dst, "w"),
    {ok, Buf} = ?DRV:malloc(Port, ?BUFSIZ),
    RFun = fun () ->
      ?DRV:fread(Port, Buf, 1, ?BUFSIZ, SrcF) end,
    WFun = fun (N) ->
      ?DRV:fwrite(Port, Buf, N, DstF) end,
    Val = copy2(RFun, WFun),
    %% Shutdown will automatically close
    %% files and free the buffer.
    ?DRV:shutdown(Port),
    Val.

copy2(RFun, WFun) ->
    copy2(RFun, WFun, RFun()).
copy2(RFun, WFun, {ok, N}) ->
    WFun(N),
    copy2(RFun, WFun);
copy2(RFun, WFun, {error, 0}) ->
    ok;                 % End of file
copy2(RFun, WFun, Error) ->
    Error.
```