

# Using FoxNet for TCP/IP Networking in ML/OS

by

Alexander Vladimirov


Submitted to the Department of Electrical Engineering and Computer Science  
in Partial Fulfillment of the Requirements for the Degree of  
Master of Engineering in Electrical Engineering and Computer Science  
at the Massachusetts Institute of Technology

May 21, 1998

[ ]

© Copyright 1998 Alexander Vladimirov. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and distribute publicly paper and electronic copies of this thesis document in whole or in part and to grant others the right to do so.

Author   
Department of Electrical Engineering and Computer Science  
May 21, 1998

Certified by   
Olin Shivers  
Research Scientist  
Thesis Supervisor

Accepted by   
Arthur C. Smith  
Chairman, Department Committee on Graduate Theses

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

JUL 14 1998

LIBRARIES

Eng

# Using FoxNet for TCP/IP Networking in ML/OS

By  
Alexander Vladimirov

Submitted to the Department of Electrical Engineering and Computer Science

May 21, 1998

In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in  
Electrical Engineering and Computer Science.

## Abstract

FoxNet is a highly modular TCP/IP stack and networking framework written in Standard ML (SML), an advanced high-level language with static type checking, polymorphic types, type inference, higher-order first-class functions, and an advanced module system. ML/OS is an operating system being developed in the Express project by extending the SML/NJ implementation of SML to run on bare hardware. ML/OS also employs the Flux OS Toolkit, a set of self-contained C libraries that implement operating system services. My experience was that SML's advanced features worked as advertised: the module system, garbage collection, static typing, first-class functions and the other features were of tremendous utility in the task of porting FoxNet over to ML/OS. However, SML has some serious shortcomings. SML/NJ is crippled by an outdated, inadequate development environment. Further, while FoxNet has many advantages over traditional TCP/IP implementations, it is too complex, the code is poorly documented, and the papers that describe FoxNet are outdated. OS Kit, on the other hand, although implemented in C, a language inferior to SML, is a well built, easy to use system. The lesson is that advanced languages features, while certainly important, are not sufficient to implement an elegant, robust, easy to understand system. They must be complemented by a good programming environment, solid system design, and thorough documentation.

Thesis Supervisor: Olin Shivers  
Title: Research Scientist

## Acknowledgments

This research would not have been possible without the help and support of many people.

First and foremost, I would like to thank Olin Shivers, my research supervisor, for his guidance, advice, and support. His ideas, technical assistance, extensive knowledge, and hacking have helped me tremendously in this work.

Albert Lin has been there since the beginning of the Express project and it was mostly his effort that made the first version of ML/OS happen. His knowledge and help with the internals of ML/OS has allowed me to make a lot of progress in my work.

Peter Szilagyi, also with the Express project, has been very helpful in all aspects of my work. He was always willing to answer the most difficult questions about anything, be that SML, UNIX, or general language design philosophy.

Kenneth Cline and Herb Derby of the Fox project at CMU have been very valuable in helping me to understand and debug FoxNet.

Peter Lee helped to arrange my visit to CMU. This visit allowed me to work closely with people of the Fox project and make a lot of progress in my work.

Lorenz Huelsbergen of Bell Labs (Lucent Technologies) has been very helpful in providing technical assistance for SML/NJ.

Roland McGrath at University of Utah has been very helpful and eager to answer any questions regarding OS Kit and UNIX in general.

Jay Lepreau at University of Utah was extremely helpful by providing me with a private release of OS Kit before it was made available to the general public.

DARPA provided some funding for this research.

I would like to thank Sonya Rikhtverchik for her love, support, and patience with me during this research. Without her I would not have the strength to finish this work.

I would like to thank Eytan Adar for listening to my complaints, joining me for snack breaks, and giving me advice, technical and otherwise.

I would like to thank Ami and Ilana (recently Katz) for being a never-ending source of fun and providing an alternative view on anything and everything.

I would like to thank my parents, Leonid and Stella Vladimirov, for their love, support, guidance, and for taking me out of a true hell on earth—the former Soviet Union.

Finally, I would like to thank my brother Gene, and my entire family. Without their love and constant support none of this work would be possible.

<b>1 INTRODUCTION .....</b>	<b>5</b>
<b>2 BACKGROUND.....</b>	<b>7</b>
2.1 STANDARD ML.....	7
2.2 STANDARD ML OF NEW JERSEY .....	10
2.3 ML/OS.....	11
2.4 FLUX OS TOOLKIT.....	12
2.5 FOXNET .....	13
2.6 RELATED WORK .....	14
<b>3 IMPLEMENTATION .....</b>	<b>16</b>
3.1 SML/NJ'S INTERFACE TO THE SYSTEM.....	16
3.2 FOXNET'S INTERFACE TO THE NETWORK HARDWARE .....	18
3.3 OS KIT'S INTERFACE TO THE NETWORK HARDWARE .....	20
3.4 USING OS KIT'S ETHERNET INTERFACE FOR FOXNET.....	20
<b>4 EVALUATION.....</b>	<b>22</b>
4.1 SML AS A SYSTEMS PROGRAMMING LANGUAGE.....	23
4.1.1 <i>Positive aspects of SML</i> .....	23
4.1.2 <i>Negative aspects of SML</i> .....	25
4.2 SML/NJ AS A PROGRAMMING ENVIRONMENT.....	27
4.2.1 <i>Positive aspects of the programming environment</i> .....	28
4.2.2 <i>Negative aspects of the programming environment</i> .....	29
4.3 INTERACTIONS BETWEEN UNIQUE FEATURES OF SML AND THE PROGRAMMING ENVIRONMENT.....	31
4.3.1 <i>Type inference</i> .....	31
4.3.2 <i>Module system</i> .....	33
4.4 FOXNET AS A GENERAL FRAMEWORK FOR NETWORK STACKS .....	34
4.4.1 <i>Positive aspects of FoxNet</i> .....	34
4.4.2 <i>Negative aspects of FoxNet</i> .....	35
4.5 FLUX OS TOOLKIT.....	38
<b>5 CONCLUSION.....</b>	<b>39</b>
5.1 FUTURE DIRECTIONS .....	40
<b>6 REFERENCES .....</b>	<b>42</b>

# 1 Introduction

In the last ten years there has been a lot of advancement in the design and implementation of advanced programming languages, compilers, and operating systems. The purpose of this project is to explore the interactions between these systems, and the feasibility of implementing operating systems in advanced programming languages. In particular, this document will describe the things which I have learned during the adaptation of FoxNet, a TCP/IP stack written in Standard ML (SML), for ML/OS, an operating system based around SML. This work was done as a part of the Express project at the MIT Artificial Intelligence Laboratory.

The porting of FoxNet to ML/OS served a number of purposes. It gave TCP/IP networking to ML/OS, so that ML/OS can provide the traditional networking services expected of modern operating systems. The porting process allowed me to evaluate the advantages and disadvantages of SML as a language for systems programming. It also allowed me to evaluate SML of New Jersey (SML/NJ) as an implementation of SML and a development environment. It also allowed me to evaluate OS Kit's interface to the networking hardware. Finally, the porting process allowed me to evaluate the design and implementation of FoxNet.

The Background section explains origins, purposes, and organization of the individual components used for this project: SML, SML/NJ, ML/OS, FoxNet, and Flux OS Toolkit (OS Kit). It also discusses previous and related work.

The Implementation section discusses the details of porting FoxNet to ML/OS. It explains how FoxNet interfaces to networking hardware in its native UNIX environment, and how that interface is changed to use OS Kit's network drivers.

The Evaluation section presents the results and observations made during the porting process. It discusses the positive and negative aspects of:

- SML as a language for systems and large project development
- SML/NJ as an SML implementation and a development environment
- Interactions between advanced features of SML and the development environment provided by SML/NJ
- FoxNet's design and implementation

- Flux OS Toolkit's network interface

This sections also presents suggestions for improvements for all of these systems.

The Conclusion summarizes the main results learned during the porting of FoxNet. It also suggests improvements for all the parts involved and outlines directions for further research.

## 2 Background

Adapting FoxNet for ML/OS involved working with a number of sophisticated systems. Most of these systems were evolving over the course of the project, both internally, and at the interface level.

### 2.1 Standard ML

Standard ML (SML) was first defined in 1990[4], and subsequently updated and re-defined in 1997[13]. It is an advanced, high-level language that has a number of features which make it a good candidate for a large software project. The language is strongly typed. Not only does this impose a lot of structure on the program design paradigm, it also eliminates a large class of implementation errors. Type checking is done at compile time, therefore types add very little overhead at run-time. Unlike many other strongly typed languages, such as CLU [14], or Modula 3[15], SML programmers do not have to specify the type of every new variable. SML uses a type-inference algorithm to deduce the types for most expressions. In addition, SML's type system supports parametric polymorphism. This minimizes the amount of code that needs to be written. For example, the same **length** function can be used to find out the length of any list, regardless of what type the elements of the list are.

First-class, higher-order functions are also a part of SML. Functions can be passed as arguments to other functions, they can be returned as results, and they can be created at run-time. This gives SML programs ability to package up a computation and pass it to a client who may then invoke it. For example, a connect call in a network stack can create a send function and return it to user code.

One feature of SML that makes it particularly attractive for implementing large projects is its advanced module system. This system is comprised of three constructs: structures, signatures, and functors. A *structure* is SML's term for a module of code. It provides a way to group a set of related name-value pairs. These items can be data values, functional values, types, and other structures. An SML *signature* is the type of a structure. It contains type information about all the items declared in a structure. A

*functor* is a structure that is parameterized over other structures. The body of the functor defines a structure in terms of its parameters, which are specified by a signature.

Applying a functor to actual structures instantiates it and produces a new structure.

SML's module system can be used to both control the complexity and guarantee some correctness properties of a complex software system. Anyone who has ever worked on a large system knows that one of the biggest problems with a large system is its inherent complexity. There are engineering principles that can help to control the complexity. One of them is modularity. Any well-designed complex system must be organized into separate modules, with well-defined, narrow interfaces. SML signatures can be used to define the interfaces between the modules. Structures can be used to implement specific modules. Functors can be used to implement generic modules, which can be specified by applying them to specific structures. A hierarchical design paradigm, which also helps to control complexity, can be used since structures can include other structures. When these modules are integrated, at compile time, the compiler checks whether they match the signatures that they're supposed to match. This check guarantees that in some ways the modules really do implement the interfaces they advertise.

This advanced module structure should in theory provide a novel solution to a common problem that plagues some modern micro-kernel operating systems, such as Mach [16] [17]. These operating systems achieve modularity and fault isolation by running all the operating system services as separate user-level processes. These services then use the kernel to communicate with each other. Since each service is a separate process, its bugs and crashes are completely isolated from any other service. Moreover, different modules that implement the same service can be used in the operating system, even at the same time. For example, two different file system services can be running at the same time. One can be optimized for large files and another one for frequent access. If either one of these file systems crashes, it will not take down anything else in the operating system.

Unfortunately this type of system architecture suffers from a serious performance problem. Even the simplest requests for operating system services require many context switches. For example, suppose a text editing process wants to read a line from a file on disk. It sends a request to the kernel, asking it to pass the "read" request to the file



system process. At this point the text editing process is suspended and the kernel is invoked. The kernel determines that the request is meant for the file system, and sends it to the appropriate process. Now the kernel is suspended, and the file system is invoked. The file system finds the correct line on disk and reads it into a private memory buffer. Then it sends the buffer to the kernel. Now the file system is suspended, and the kernel is invoked. The kernel copies the buffer from the file system's memory space into the text editor's memory space, and returns to the text editor. Now the kernel is suspended. Thus, a simple request to read a line of text from disk results in four context switches. A context switch is an expensive operation on most modern processors, since it requires flushing of the caches, branch predictors, translation look-aside buffers, and page tables.

This problem, however, exists only because all the services are separated as different processes during execution. Since these operating systems are implemented in C or C++, their implementors have no other way to modularize the processes, since these languages do not have good module support. Modularity can be provided by a different method, however, and one that does not incur this prohibitive run-time cost. If all the services can exist as separate modules at the source level, but then turned into one monolithic kernel by the compiler, both modularity and good performance can be achieved. In a type-safe, memory-safe language with well-defined semantics, the resulting pieces of the system should not interfere with one another, even if they do exist as one process, in one memory space at run-time. SML is just such a language, and its module system extends the guarantees enforced by its type system and safe semantics to the "programming in the large" level.

Another feature of SML that makes it a good choice for large projects is formally defined semantics. The Definition of Standard ML gives complete semantics of the languages. The meaning of every language construct can be unambiguously understood from the definition. This guarantees a couple of desirable properties. First, programs written in SML can be compiled using different SML compilers and are guaranteed to do the same things. Second, the effects of any program can be determined formally using the language definition and program text. This makes correctness proofs possible for those applications where that is necessary. And finally, mathematically defined

semantics opens up a large area of research for theoreticians who can develop various global optimizations based on high-level theoretical tools such as lambda calculus.

## ***2.2 Standard ML of New Jersey***

Standard ML of New Jersey (SML/NJ) is a compiler, a runtime system, and a set of tools for SML. It is a joint development effort at AT&T Research, Bell Laboratories (Lucent Technologies), Princeton University, and Yale University.

SML/NJ was chosen by the Express project for a number of reasons. It is one of the most mature implementations of SML. Although the last official release was in 1993, SML/NJ is constantly under development, and latest releases are always available. It is distributed together with source code, which makes modifying it for our purposes possible. The SML/NJ development team is readily available and was imperative in helping us with initial stages of the project.

SML/NJ, combined with a standard set of UNIX tools comprises a full SML development environment. The runtime system, written entirely in C, is responsible for loading the SML heap image from disk, executing it, and maintaining all interactions between the compiler and the underlying system. The runtime system is highly dependent on the operating system and the architecture. The heap image contains compiled code and SML data structures and is independent of the operating system.

SML/NJ also contains a number of development tools that are available as additional packages. The Compilation Manager (CM) is a very useful tool for managing large projects [18]. Much like the UNIX **make** utility, it can keep track of what source files in a project have changed and need to be recompiled. It also keeps track of all the dependencies and recompiles the files dependent on those that were changed. In addition, CM provides visualization capabilities for large projects. It can produce module dependency graphs that graphically illustrate all the dependencies between files in a large project.

SML/NJ also contains a lexical analyzer generator, ML-Lex, a parser generator, ML-Yacc, an extension for multi-threaded programming, CML, and a graphical user interface, eXene.

In addition, an SML mode for Emacs, a popular extensible text editor, is readily available. This mode allows the SML read-evaluate-print loop (REPL) to run in an interactive window. The source code can be edited in another window, and there is a set of commands to allow communications between the two windows. For example, it is possible to type an SML expression into the text-editing window, then invoke a command that will send the expression to the REPL window to be evaluated. Moreover, the source code mode understands SML syntax and provides correct code indentations and keyword coloring automatically.

## **2.3 ML/OS**

ML/OS is a research operating system for the x86-personal computer (PC) architecture from the Express project. It is based around, and implemented in SML. At the heart of this operating system is SML/NJ. The system is being built by extending the compiler and run-time system with services and features that make it into an operating system. These features include:

- an ability to boot up and set up processor state
- memory management
- input/output (I/O) management
- persistent storage management
- networking

The goal behind ML/OS is to explore the interactions between advanced programming languages, advanced compilers, and operating systems. It is not meant to be a production or commercial-level system usable by anyone who is computer-literate. It is supposed to prove or disprove different concepts being investigated by the Express projects. Its development will give us an opportunity to evaluate all the claims for SML proponents on a real system. It is our hope that it will provide a fun and productive development environment, orders of magnitude better than what is available today.

There is a symbiotic relationship between the UNIX operating system and the C programming language. C serves UNIX as the implementation language for the kernel, as well as a language that provides the programming interface for UNIX services. UNIX serves C by providing services that are tuned for the C runtime model, such as separate

address space, and text-based I/O. Along the same lines, the Express project is trying to develop an operating system that will exist in a symbiotic relationship with SML.

Since an important goal of ML/OS is to be an advanced operating system of the future, having a set of standard networking protocols is an important requirement. Furthermore, ML/OS must provide a networking framework that is flexible enough to be easily extended with new protocols that might become popular in the future. Moreover, it must be a convenient environment for easily developing and researching new networking protocols.

## ***2.4 Flux OS Toolkit***

The low-level operating system components, which are not the focus of the Express project, are currently implemented using the Flux OS Toolkit (OS Kit), developed at University of Utah [5]. OS Kit is a collection of libraries that are meant to ease the job of operating system developers. Each library implements some traditional operating system services, and exports clean interfaces for them.

A significant advantage that the OS Kit libraries have over traditional operating system libraries is that they are self-contained. For example, the memory management library makes very few assumptions about what kind of an environment it is being used in. That way it can be used in almost any operating system, without requiring presence of any other OS Kit libraries. This collection of self-contained libraries makes it very easy for operating system writers to concentrate on those parts of the operating system that they want to explore and improve.

ML/OS uses a number of libraries from OS Kit. It uses the kernel support library to put the processor into the correct mode, set up various processor tables, such as the Interrupt Descriptor Table.

ML/OS uses the memory management library to keep track of all the available and used memory in the system. It also gives control over what type of memory gets allocated. For example, some device drivers might require memory in the first 16 Megabytes of space. The memory management provides a facility for allocating this type of memory. This library is particularly helpful, since the PC memory architecture is complex and irregular.

ML/OS also uses the C library provided by the OS Kit. This library serves the same purpose as the standard C library found in UNIX. Among others it includes I/O functions, such as **printf**.

ML/OS also uses the device drivers library to interface to specific hardware devices. Currently it only uses the Ethernet device drivers, but as ML/OS gets further into its development it will use the drivers for permanent storage, graphics, sound and other devices.

Another feature of OS Kit that is extremely helpful is serial-line remote debugging capabilities. Since ML/OS is not an environment stable or complete enough to host its own debugger, the only one usable debugging tool (besides **print** statements) is a remote debugger. With remote debugging, gdb (or another compatible debugger) can run on a machine with a stable environment. It communicates with ML/OS, running on another machine, via a serial line. OS Kit provides this capability in its remote debugging library.

## **2.5 FoxNet**

Fortunately for the Express project, there already exists a TCP/IP stack written almost entirely in SML. This stack is FoxNet from the Fox project at Carnegie Mellon University [1] [10] [11]. FoxNet is a fully functional TCP/IP stack that works on DEC Alphas running Digital UNIX. It works in addition to the standard TCP/IP stack that is a part of Digital UNIX, and it interfaces to the networking hardware using the DEC packet filter interface [6].

FoxNet is not just a traditional TCP/IP stack however. Besides the obvious peculiarity of being implemented in SML – a high-level language not normally used for implementing networking protocols – FoxNet promises a number of advantages.

FoxNet uses SML's module system to integrate separate protocol layers into a monolithic stack. It defines a generic SML signature for all protocols. Any two or more protocol structures that match the generic protocol signature can be composed to create a protocol stack. The requirement that every layer must match the generic protocol signature is checked by the compiler, and if satisfied, guarantees that the layers will be

able to communicate. Thus, FoxNet has the ability to mix and match various standard protocols to create some standard networking stacks, such as TCP/IP, as well as some specialized protocol stacks, such as TCP over Ethernet.

FoxNet's signature framework also results in an implementation that really reflects the layered design model of a protocol stack. Usually each protocol layer in a protocol stack is designed independently of any other layer. Once all the layers are designed, the implementors do not keep this modular, layered structure. Instead, they implement all the layers together, to form one monolithic protocol stack. This is usually done for performance reasons, to minimize buffer copying between different layers in a stack, as the packets travel up and down the stack. Such integrated implementation, however, violates the basic principles of abstraction, modularity, and hierarchy in software and systems engineering. FoxNet attempts to break this trend, by keeping the layered model in the implementation. Every layer is implemented completely independently of every other layer.

Being implemented in an advanced language, FoxNet should be easy to understand and port to FreeBSD, our development environment, and ML/OS. In fact, the creators of FoxNet conclude, "...most of the information needed to understand the structure of our code can be obtained from a study of the signatures alone." [1]

FoxNet promises all these advantages without a significant sacrifice of network performance as measured by throughput and latency [1]. All these things make FoxNet an ideal candidate for networking in ML/OS.

## ***2.6 Related Work***

The project very closely related to this one is the Fox project at Carnegie Mellon University [10]. Besides being the origin of FoxNet, the Fox project is also exploring the feasibility and advantages of implementing large systems in SML. They have also attempted to port SML/NJ on bare hardware. The same task was once undertaken at Bell Labs without much success.

There are a number of existing projects that are attempting to write an operating system in an advanced language. Numerous groups, including Javasoft at Sun Microsystems are working on an operating system implemented fully in Java. The

Inferno operating system at Bell Labs is also a similar effort. It is being implemented in the Limbo language. A group at EMC Corporation is using Eifel's interfaces to provide robustness in some of the critical modules of their operating system. The Lisp Machine had special-purpose hardware and operating system written entirely in Lisp.

Besides project Fox and FoxNet, there has been another attempt to implement well-structured protocol stacks. This work was done by the x-kernel project [19]. In the x-kernel project all the layers export the same set of functions with the same set of arguments. As a result, arbitrary protocols can be combined to form a stack. Unlike FoxNet, x-kernel does not have the advantage of being able to formally specify a protocol signature. Upon combining protocols, the compiler cannot guarantee type safety of the combination.

While there has certainly existed numerous efforts to evaluate various languages, only Andrew Appel evaluated SML [20]. In his evaluation, however, he left the programming environment available for SML programmers completely unexamined.

## 3 Implementation

In order to port FoxNet to ML/OS I chose to go through an intermediate step of first porting it to FreeBSD—our development environment. Going to FreeBSD first enabled me to learn those parts of the systems which I would be modifying and using, while having all the development tools of a mature UNIX system. These parts are: SML/NJ's interface to C functions and system functions, and FoxNet's interface to the networking hardware. After the FreeBSD port was finished and FoxNet was running under FreeBSD, I proceeded to learn OS Kit's interface to networking hardware. Then I designed and wrote a layer of code that would on one side talk to the OS Kit's network interface, and on the other side to FoxNet.

### ***3.1 SML/NJ's interface to the system***

An out-of-the-box SML/NJ provides access to the underlying operating system through various SML libraries, such as the SML basis library. These libraries include a lot of the popular system functions such as file input/output (I/O), sockets, system time, and so on.

Although the libraries that are provided with a stock distribution of SML/NJ are fairly complete, they are mostly the same for all operating systems on which SML/NJ runs. As a result they include only that functionality which is present in most operating systems. Unfortunately that leaves some thing out. To circumvent this limitation SML/NJ provides a way for users to extend these libraries with arbitrary C functions.

The way that SML basic library functions are implemented internally is through C code. For almost every SML system function available to the user, there is a C function in the runtime that actually talks to the system. The way these functions are made available to SML code works as follows. In the SML/NJ runtime source tree, there is a directory that contains the sources for all the C functions callable from SML. A group of related functions is put in a separate directory and is called an SML C library. A header file inside each C library directory specifies the functions exported by the library. The functions are specified using macros provided with SML/NJ that convert C function



signatures into SML function types. Macros are also provided to extract arguments passed to the SML function. Inside the functions different macros can be used to convert between C and SML values.

This mechanism is best illustrated with the following example. This piece of C code implements a **readbuf** function that can be called from SML:

```
ml_val_t _ml_P_IO_readbuf (ml_state_t *msp, ml_val_t arg)
{
    int    fd = REC_SELINT(arg, 0);
    char   *start = REC_SELPTR(char, arg, 1) + REC_SELINT(arg, 3);
    int    nbytes = REC_SELINT(arg, 2);
    int    n;
    n = read (fd, start, nbytes);
    CHK_RETURN (msp, n)
}
```

The formal arguments specified in the C function signature are the same for all the C functions callable from SML. They are: pointer to the structure that contains SML runtime state, and the argument vector. Macros such as **REC\_SELINT** are used to extract the actual arguments from the argument vector. The **CHK\_RETURN** macro checks that the result was not an error, and then converts it to SML integer value. This function is exported in the header file by the following line:

```
CFUNC("readbuf", _ml_P_IO_readbuf, "int * Word8Array.array * int -> int")
```

This line says that **readbuf** function will be available in SML, and its SML type will be “int \* Word8Array.array \* int -> int.” When it gets called the **\_ml\_P\_IO\_readbuf** C function will be called.

Once the runtime is compiled with the new functions, all the available C functions can be invoked by calling a special SML function **c\_function**. This function takes the name of the library and the name of the C function in the library, and returns an SML function that can now be called. Evaluating the following expression inside SML can use the **readbuf** function:

```
val readbuf: int * Word8Array.array * int -> int = c_function “posix-io” “readbuf”;
```

The library name “posix-io” comes from the fact that the **readbuf** function is a part of the posix-io library, contained in the posix-io directory. After this expression is evaluated **readbuf** can be used just as any other SML function.

### ***3.2 FoxNet's interface to the network hardware***

As explained in section 2.3, FoxNet uses the SML module system to implement protocol layers and combine them into a protocol stack. Every layer is implemented as a functor. One of the functor's parameters is a structure that must match the generic PROTOCOL signature and implements a lower layer of the stack. The result of the functor application is a stack that now contains a new layer. For example, given a structure that implements the Ethernet layer, an Arp functor can be applied to it. The result of the application will be a structure that implements the Address Resolution Protocol (ARP) layer, on top of the Ethernet layer [21]. Now this structure can be passed as a parameter to the Ip functor. This application will produce a structure that implements the Internet Protocol (IP) layer on top of the Arp/Ethernet stack [3]. Now the Tcp functor can be applied to this structure. This application will create a structure that implements the Transmission Control Protocol layer on top of the Ip/Arp/Ethernet stack [2]. This structure will implement a more or less standard TCP/IP stack that can now be used for reliable communications between any two hosts on the Internet.

To build such a stack, however, one needs the lowest layer protocol to start building up the stack. In the above explanation I assumed that this layer—the Ethernet layer—already existed in the environment. In reality this layer needed to be built and it had to provide a way to send and receive raw packets from the Ethernet hardware. This is the layer where all of the porting work took place. Only minor modifications had to be made to layers above it.

In the original CMU FoxNet implementation for DEC Alphas running Digital UNIX, FoxNet provides a protocol stack that co-exists with the standard TCP/IP stack present in Digital UNIX. The way the stack gets to the raw Ethernet packets is through the DEC packet filter interface [6]. The packet filter exports an interface that allows the user to read all the incoming packets and send arbitrary packets to the network. It also provides a filter language, which can be used to read only packet of certain types. For example, the filter language can be used to filter out all packets except broadcast ARP packets. FoxNet does not use the filter language, it simply uses the packet filter as a way

to receive all the incoming packets. Higher layers in FoxNet discard the packets that are not meant for it, such as broadcast ARP packets querying another host.

FoxNet extends the stock SML/NJ runtime with a new C library, called “foxnet.”

This library exports the following functions:

- **open** – this function opens the packet filter
- **writenv** – this function writes a packet scattered around various buffers to the network
- **read** – this blocking function reads the next packet from the network
- **get\_address** – this function returns the hardware address of the Ethernet card. This address is used in some of the TCP/IP layers, such as the ARP layer.

In addition the **poll** function from the stock SML/NJ basis library is used to check if new packets have arrived from the network. A **device** structure is implemented using these functions. This structure matches the PROTOCOL signature, and so now it can be passed into other functors to create protocol stacks as described above.

When FoxNet was ported to FreeBSD the foxnet library in the SML/NJ runtime had be slightly changed. The reason for the change was that FreeBSD does not provide the DEC packet filter interface. Instead FreeBSD provides Berkeley packet filter interface [8] [9]. While there is nothing that is fundamentally different between the two packet filters, there are some differences in the way you access them.

Other than differences in the names and arguments of functions that access the packet filter, the most significant difference was in the **read** call. In the DEC packet filter, the **read** call always returns just one packet without any extraneous data. The SML code relies on this, and when it calls the packet filter **read** function from SML it only expects one packet, without anything appended to it.

The FreeBSD packet filter, on the other hand, copies as many packets as there are currently available in the kernel buffer into a user-supplied buffer. Moreover, a special header is pre-pended to each packet. The header contains such information as the packet length, the header length, and so on. FreeBSD also provides macros to parse the header and locate the boundaries of each packet within the buffer. Thus, to make the Berkeley packet filter work with FoxNet a layer of code was written to parse the buffer into

packets and put them into a special queue. Whenever FoxNet called **read** from SML the queue was checked. If there was at least one packet there, it was returned to SML code, otherwise a call to the packet filter was made to get more packets. One of these packets was then returned to SML, while other ones were put on the queue.

### ***3.3 OS Kit's interface to the network hardware***

As mentioned in section 2.4, ML/OS uses OS Kit's driver library to interface to the hardware devices. In particular I used the Ethernet drivers and the **netio** interface to send and receive Ethernet packets. The interface provided for these purposes by the OS Kit is fundamentally different from the packet filter.

OS Kit uses an object-oriented model for most of the interfaces. An OS Kit **netio** object is an abstract packet consumer. It has only one method, **push**, which accepts a packet. What the **netio** does with the packet depends on the specific **netio** object. When an Ethernet device is open, a **netio** object exchange takes place. A module that opens the devices passes in a **netio** object, which is then called by the Ethernet driver whenever a packet arrives from the network. This object is effectively a handler for the incoming packets. Since the packets arrive from the network asynchronously, this handler will be called asynchronously. The **open** call also returns a **netio** object that can now be used to send packets to the network.

Thus this interface is different from the packet filter interface in the way the incoming packets are handled. With a packet filter, a synchronous read can be issued, which will return the next packet. The actual packet queuing and buffering is done by the operating system. With the OS Kit interface to the network card, the incoming packet handler is called asynchronously, whenever a packet arrives from the network. In addition, there is no way to simply poll the device to see if new packets have arrived.

### ***3.4 Using OS Kit's Ethernet interface for FoxNet***

Because the Ethernet interface provided by the OS Kit is different in nature to the packet filter interface used by FoxNet, the SML/NJ runtime foxnet library had to be rewritten almost entirely to work with ML/OS. The new foxnet library had to talk to the

OS Kit's Ethernet interface on one side. To keep the SML FoxNet code the same, the new foxnet library had to provide the same interface as before on the SML side.

To accomplish this task, a traditional producer-consumer model was used. The foxnet library now incorporated a packet queue. The **netio** object that is passed to the Ethernet interface during an **open** call does the following. When a packet arrives, and this **netio** object gets called, it checks if the queue has not reached the maximum size. If it hasn't, the packet is added to the queue. Otherwise the packet is simply dropped. Whenever FoxNet calls **read** from within SML, if the queue is not empty, the oldest packet is returned. If the queue is empty, the call will busy-wait for a packet to arrive. Busy-waiting, however, never occurs, because FoxNet only calls **read** after calling **poll** to make sure there is a packet. **poll** works by simply checking whether the queue is non-empty.

The implementation of **writew** is fairly straightforward. First, all the buffers passed into **writew** are copied into a single contiguous buffer. The resulting packet is then sent to the network by using the **netio** object returned by the Ethernet interface during the **open** call. The implementation of **get\_address** is also straightforward, because OS Kit provides a call that does exactly the same thing—returns the hardware address of the Ethernet card.

## 4 Evaluation

When large software systems, such as networking stacks, are designed and implemented, one of the key design goals is usually to control the complexity of the system. When this design goal is ignored, the resulting system is usually too complex to keep under control even for the people who designed and built it. Such a system is usually completely impenetrable for someone who was not involved with the design and building of it. If he or she has to maintain this system, many months or even years are needed to first learn and understand the system. Therefore, a system is usually considered successful from the designer's point of view if it contains all the required features and is easy to understand. Ease of understanding brings with it many other desirable properties. These include robustness, straightforward implementation with relatively few bugs, ease of debugging and maintenance, and extensibility.

Proponents of advanced programming languages, such as SML, argue that there are many benefits in implementing large systems in these languages. One of the benefits they always bring is that the inherent elegance of the language, coupled with careful system design, will result in an elegant, easy-to-understand system. SML advocates, specifically argue that SML's advanced module system based on signatures, structures, and functors, together with static typing, type inference, and polymorphism can be used to build a highly modular, and therefore easy-to-understand system.

What makes SML's module system especially useful is that the interfaces between modules can be specified, and checked by the compiler. System implementors do not have to rely on conventions that have to be observed at the interfaces. This eliminates an important class of programming-in-the-large errors—errors that occur when integrating modules together.

Another advantage claimed by SML's advocates is easy portability. Since everything in the language has well-defined semantics, any program should have the same meaning regardless of which platform it's being compiled and run on. Therefore porting any system written in SML to another platform should theoretically be as simple as re-compiling the system on that platform.

FoxNet is clearly a complex software system, because of the difficulty and the scope of its goals, as explained above, and because of its sheer size, which is 40,000+ lines of SML code. Undertaking the porting FoxNet to ML/OS, has forced me to develop a solid understanding of all the systems involved, both as the end-user and as a system engineer. In the process of gaining that understanding I have made observations about various aspects of all the systems. I have noted those aspects that made my job easier, as well as those that made it harder.

## ***4.1 SML as a systems programming language***

As already mentioned before, SML is a very advanced language, with many interesting features. Some of these features turned out to be very helpful, others not so much. I also felt that some features were missing.

### **4.1.1 Positive aspects of SML**

#### **4.1.1.1 Garbage Collection**

While certainly not a feature unique to SML, garbage collection is definitely the most useful and welcome feature. Having worked with languages that require explicit memory management, such as C and C++, I can truly appreciate the benefits of garbage collection.

I can imagine what kind of a nightmare my job of porting FoxNet would have been if I had to worry about allocating and freeing memory. Keeping a clear invariant for which modules are responsible for freeing the memory passed to them is a task that is difficult enough for the people building the system. It would have been impossible for me, as someone who did not build the system, but was trying to port it.

It is unpleasant enough to chase down memory bugs in user-level programs. It gets a lot worse, however, when you have to find memory bugs in the kernel. If a user-level program references an invalid pointer, it gets a signal from the operating system and usually dumps the memory state (core). This core can then be used to find the bug. In a kernel, on the other hand, an invalid pointer reference does not dump core. It crashes the kernel and reboots the machine.

#### 4.1.1.2 Static typing and type checking

Most people think of type checking as simply a tool that eliminates a large class of implementation errors. While that is certainly true, there is another benefit of types that is often overlooked. Types impose a design paradigm, discipline, and structure onto the programmer.

During the process of learning FoxNet I found the type information extremely helpful. A lot of times by just knowing the type of a variable I was able to determine what its purpose was. Also, the structure imposed by the type system is clearly seen in FoxNet, and understanding the structure was very helpful in gaining insight into the system. Moreover, type errors in my modification to FoxNet have been imperative for pointing to sources of more serious errors.

These advantages of the type system are exactly those touted by the designers of SML. It is clear that they have delivered on their promises here.

#### 4.1.1.3 Module system

As has been mentioned a couple of times above, SML has a unique module system. Structures can group name-value pairs together, signatures specify the types of everything within a structure, and functors are structures parameterized over other structures.

Because FoxNet uses the module system extensively, I had a lot of exposure to it during the porting process. While there are some negative aspects of the module system, which will be discussed below, for the most part it did what a module system is supposed to do. I could understand a lot about a particular structure by studying the signature and the code. On very few occasions did I have to look at the implementation of one module while attempting to understand another. Thus, the module systems kept the separation between modules reasonably well.

When I had to change the structure that implements the Ethernet device, the compiler complained until my implementation matched the DEVICE signature. Once that was done, very few bugs were left to work out. The signature framework ensured



that structures indeed implemented what they were supposed to. Here, again, SML designers have clearly delivered on their promises.

#### 4.1.1.4 Call-with-current-continuation

SML/NJ adds a few things to the SML language. One of the more peculiar things it adds is a call-with-current-continuation (call/cc) function. This function captures the current control state of the execution, and packages it up as a first-class value. This value can be used later to resume the execution at the point where call/cc was invoked. Further details on this powerful and sometimes confusing operation can be found in programming language books [23].

FoxNet uses call/cc to implement a threading package. This package is completely self-contained, which makes it robust and independent of the operating system. I had to change only a small part of it in order to get it running in ML/OS.

### 4.1.2 Negative aspects of SML

While the positive aspects of SML were extremely helpful to me during the porting process, SML is far from perfect. Some things could be improved on and some things could be added.

#### 4.1.2.1 Module system

Since my preferred style of programming is an object-oriented approach such as that of CLU or Java, the SML module system took some getting used to. It is more complex than the object system of CLU. A more powerful system can be easily misused and even abused. A powerful tool can be a dangerous weapon in the hands of unskilled people. There is a reason that industrial-strength power sanders are not available to the general public. In the hands of an average person such a power sander can do more harm than good. I believe in the design philosophy advocated by MIT's software engineering classes, which states that things should be kept as simple as possible and there should be very little opportunity for potential misuse.

The thing that I like the most about the module system of a language such as CLU is the simplicity of the interface it dictates. In CLU, a module is a CLU cluster, and it has

a CLU type. This type simply specifies all the function that the cluster provides— the cluster's methods.

An SML structure, on the other hand, does not have a single type that specifies it. The specification of a structure is an SML signature. An SML signature, however, can be much more complex than a CLU type. In addition to functions, a signature can contain specifications for data members, other SML structures, and types. This generality allows the programmer to make a signature amazingly complex.

SML's structure should theoretically provide a way to group *related* items together. It does not, however, restrict the programmer from putting *unrelated* items together. It is not even clear if enforcing the item-relatedness in a structure is at all possible at the language level. If the structure construct is used as intended, the resulting system is modular and modules are easy to understand. If the construct is misused, however, the resulting system can be a bunch of huge modules which by themselves do not make a whole lot of sense.

Murphy's law states that if there is a way to misuse some feature of a system, sooner or later somebody will misuse it. SML's module system is a feature that can be easily misused, with a lot of potential for harm. I believe that if one has to choose between power and safety, safety is most often the right choice.

#### 4.1.2.2 Lack of access to low-level programming in SML

FoxNet sends and receives raw Ethernet packets using a C library at the lowest level. That library is where most of the porting work actually happened. The reason this is done in C code is quite simple: there is no way to do that in SML.

Whenever a part of the system must operate on machine level, SML programmer is forced to leave SML and drop into C. Obviously none of the benefits of SML are relevant when you are programming in C. SML advocates commend to the public the portability of SML. Recall that most of the porting work was done in C, changing the foxnet library for SML/NJ runtime. Very little was changed in SML. If there were no need to drop into C, SML would be 100% portable. I would have to do very little work to port FoxNet first to FreeBSD, and then to ML/OS. The need to drop into C sometimes makes SML code not 100% portable.

I think there are some reasonable ways to improve the situation. One thing that SML creators are working on already is the ability to manipulate C data structures from within SML. Having this ability could make it possible to implement such low-level things as device drivers almost entirely in SML. Little pieces of C code would be necessary to actually map data structures to physical device registers for example. Although that could be done in SML as well by employing constructs that would put certain data structures into specified memory locations.

Modula 3 has a solution for this problem [15]. Modula 3's modules can be declared as safe or unsafe. Those declared safe cannot do anything that violates the semantics of the language, such as converting an integer into a pointer. With modules that are declared unsafe, however, all bets are off. Unsafe modules can exploit "loophole" constructs in the language that are designed to enable low-level manipulation of the machine. Device driver code can be written by putting it into an unsafe module.

The fact that Modula 3 has support for low-level programming while SML does not can be explained by the history of both languages. Designers of Modula 3 came from a systems background. Therefore they carefully designed into the language features to enable low-level systems programming. SML comes from the theorem-proving community—which may explain its lack of these features.

## ***4.2 SML/NJ as a programming environment***

SML as a programming language does not exist in a vacuum. There is no way to work with pure SML. A real SML implementation and programming environment must be used. As mentioned above, SML/NJ under FreeBSD comprises our SML programming environment. While some tools in the environment were helpful, as a whole the environment is inadequate. Some essential tools are missing, and others do not scale to projects the size of FoxNet.

## **4.2.1 Positive aspects of the programming environment**

### **4.2.1.1 SML mode for Emacs**

Most of my interactions with SML code were from within Emacs. The SML mode for Emacs provides a number of simple but surprisingly helpful features. In the text-editing mode Emacs presents different parts of the code in different color. For example, SML keywords are yellow, comments are green, strings are blue, and types are red. This colorization serves as the first line of defense against syntax errors. For example, if a keyword is misspelled, it will not be interpreted as such by Emacs, and presented in a color other than yellow. Since I am used to seeing all the keyword in yellow, this would immediately alert me, and I would correct the mistake before even attempting to compile the code.

Features of the interactive mode were also extremely helpful. If an SML/NJ REPL is run from within a UNIX shell, there is no way to do things as simple as repeating your previous command. So if you have typed in a long expression, but made a syntax error, you will have to type the whole expression all over again. Emacs on the other hand stores a buffer of all the previous commands entered into the REPL. Any one of them can be recalled, edited, and re-evaluated. In addition, expressions can be typed into a text-editing buffer, where colorization is applied. Then an Emacs command can be invoked to send the expression to the REPL buffer for evaluation.

### **4.2.1.2 Compilation Manager**

The compilation manager (CM) that comes standard with SML/NJ was extremely helpful during the porting effort. CM's ability to recompile only those parts of the system that changed saved me many hours.

Before I started using CM, all of FoxNet had to be recompiled after the smallest code change. This recompilation took around 40 minutes, which was completely unacceptable. When I started to use CM, an average re-compilation took about 5 minutes.

## 4.2.2 Negative aspects of the programming environment

While Emacs and CM are helpful, the general state of the SML/NJ programming environment leaves a lot to be desired. It is missing some fundamental tools that have existed in other environment for many years. After working with SML/NJ for over a year, I have developed a firm belief that it can never gain widespread acceptance unless the environment is significantly improved. There is hope for SML as a language, however. Recently a commercial implementation of SML, Harlequin MLWorks, has been released. It is reported that this package provides a full and productive, commercial-level development environment.

### 4.2.2.1 Lack of a debugger

Anybody who has been involved in building, debugging, or maintaining a large system knows the benefits of a good debugger. Without it the only way to see where things go wrong is to insert output statements everywhere in the program. Obviously that is far from the best way of doing things. It has been my experience that debugging FoxNet without a debugger was very similar to driving a car with no brakes or doors, with a windshield painted black, during rush-hour traffic in Boston. Once a bug occurs, there is no way to stop the system and examine its state. There is no way to see how a variable gets updated while the code is running, other than to insert print statements whenever the variable is mentioned. FoxNet is a sophisticated system with concurrency; many things are happening at the same time. It is hard to reason about it without being able to see what it is actually doing at run-time.

It is true that the type system of SML is great for catching errors. It does not, however, catch all of them. After the program compiles successfully, there is no support for debugging. In fact, during the porting, I found it much easier to debug those parts of the system written in C, because I could use gdb. This is a very unfortunate state of things, especially since some earlier versions of SML/NJ included a very sophisticated debugger [22].

#### 4.2.2.2 Where's the source?

It is a common situation to be reading through code and encounter a call to a function that does not appear in the file being examined. Most programming environments provide an easy way to find the definition of the function. In some environments all it takes is a double-click of a mouse on the function name.

With SML/NJ there is no easy way to do that. The only tool available for that is an ugly combination of **find** and **grep** UNIX commands that searches through all the source files for the name of the function. This method of searching has a number of disadvantages. First, it'll not only turn up the definition of the function, it will also turn up all the calls to the function. For a popular function, that can produce a list of matches almost as big as the entire source. Obviously that list would not be very useful. Another disadvantage is that all files have to be searched. When these files reside on a remote file system that has to be accessed over the network, this search can take a very long time. Thus, a tool that would index all the function definitions and provide an easy way to search for any one of them would be extremely helpful. A simple tool such as **etags** would not do the job because the module system complicates things further, as will be explained later.

#### 4.2.2.3 Lack of a call-graph tool

Sometimes one wants to do just the opposite of the scenario described above. If a function definition is being examined and the meaning is not clear, sometimes it helps to see how the function is being used. Moreover, in a higher-order language sometimes it is necessary to see all the invocations of one function, to find all the sources of a different one. Suppose function **bar** is a higher-order function that takes some arguments, and returns a new function **foo**. To see all the definitions of **foo**, I need to find all the invocations of **bar**.

Most programming environments provide a way to find all instances where a particular function is being called. In SML/NJ, however, you are stuck with **grep** and **find**, with the same problems as above.

#### 4.2.2.4 CM's visualization tool does not scale

To be fair to SML/NJ, CM does come with a tool that can provide a graph illustrating all the dependencies in a project. They say that a picture is worth a thousand words, and I believe that definitely applies to the module-dependency diagram (MDD) which CM produced for FoxNet (figure 1).

It is possible that for some simple projects CM produces MDDs that convey useful information. It is obvious, however, that this tool does not scale to a project the size of FoxNet.

One way I see to improve this tool is to provide control over the level of detail. For example, one could start by looking at the major groups of files that represent somewhat self-contained modules. A single node can represent such a group of files. After the highest-level structure is clear, one can then zoom into the details of any particular group of files.

### **4.3 *Interactions between unique features of SML and the programming environment***

The shortcomings of the SML/NJ development environment described above would apply to almost any language. There are some features of SML, however, that are unique. To be fully useful, these features require some tools that, if present, would also be unique to SML. Without these tools, I found that these SML features create a lot of problems.

#### **4.3.1 Type inference**

One of the most spoken-about features of SML is type inference. Opponents of type systems often argue that type information present in strongly typed languages such as CLU and Java makes programs too wordy for understanding. Type inference seems to offer a perfect answer for these people—you get all the benefits of strong typing, but your program is not full of type specifications.

Personally, I have never found type information in programs to be an unnecessary annoyance. In fact, I have always found types absolutely essential for my understanding of code. I think that code without types is too terse for understanding. Unfortunately,

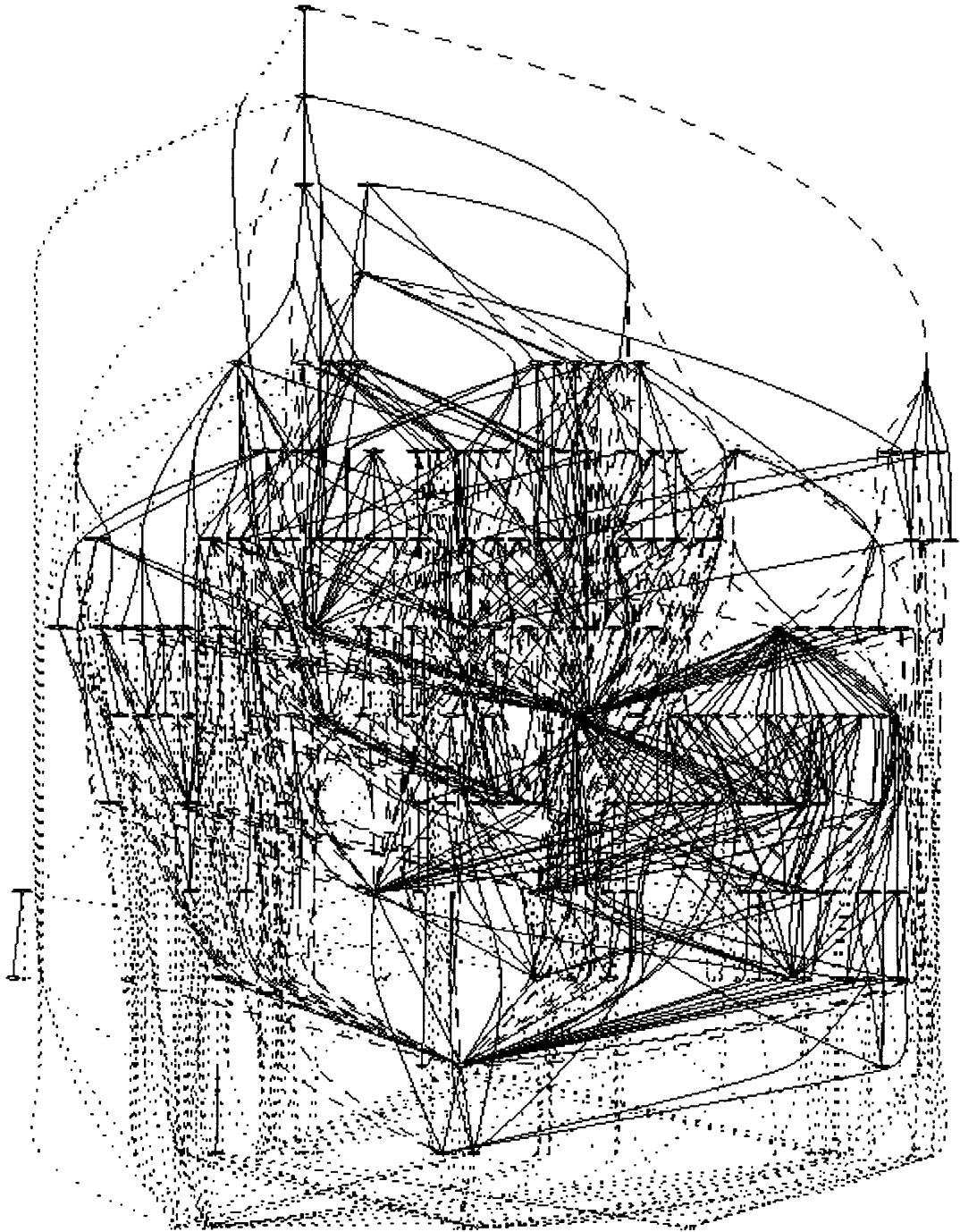


Figure 1: FoxNet Module-Dependency Diagram. Nodes represent files, labels show file names, and edges represent dependencies.



poorly documented SML code has very little type information in it. Moreover, unless the code in question is a short, self-contained expression, there is no way to make SML/NJ evaluate it and report the types that it inferred.

I have found this inability to get the types of expressions extremely hindering during the porting process. There has been a number of times when I would stare at an expression for a long time without realizing what it means simply because I did not know its type.

This problem should be fairly easy to fix. The compiler infers the types of all the variables in all of the code. It can then dump this information in some indexed, textual form. A tool, such as Emacs can then parse this information, so that when asked, it could provide the inferred type of any variable. Why this functionality was not put into SML/NJ is beyond me.

### 4.3.2 Module system

Another unique feature of SML that requires a special tool to be truly useful is the module system. I have found two separate aspects of the module system that would benefit considerably from special tools.

When specifying signatures in SML, it is possible to include other signatures in the specification. For example, this signature specification actually comes from FoxNet:

```
Signature ETH_NUMBER =  
  sig  
    include KEY  
    val new: Word48.word -> T  
    val convert: T -> Word48.word  
  end
```

Besides specifying **new** and **convert** it also includes the **KEY** signature. To fully understand what the **ETH\_NUMBER** signature means, I have to understand the **KEY** signature. To understand the **KEY** signature, I have to find its definition. SML/NJ does not provide an easy way of doing that. The **KEY** signature might include some other signature, and so on. To find the definition of these signatures, once again I am stuck with **find** and **grep**, and all the disadvantages explained above.

A similar problem exists with functor definitions. Functors are structures parameterized over other structures. A functor application results in a structure with a specific signature. This signature can be specified by the programmer and checked by the compiler. It can also be left unspecified and the compiler will deduce a signature of the structure that the functor implements. In this case the only way to learn the meaning of the functor is to read the code.

Even when the signature of the functor is specified, it does not contain all of the semantic information. A signature gives me 3 kinds of semantics:

- Whatever is expressed in the types.
- Whatever I can informally infer from the names of the exported identifiers.
- Whatever the programmer might have put in the comments—if I am lucky.

All this information is frequently not enough to understand the functor fully, especially if the signatures are poorly documented. The only way to learn it is to see the code that implements it. In the current programming environment there is no tool to do that besides **find** and **grep**.

## **4.4 *FoxNet as a general framework for network stacks***

As already explained above the designers of FoxNet did not simply set out to build a regular TCP/IP stack. Among other things, they promised that FoxNet is a framework that is elegant and easy to understand, where most of the understanding can be gained by studying the signatures alone. This claim has never been verified since I am the first person to use FoxNet outside of the Fox project at CMU.

### **4.4.1 Positive aspects of FoxNet**

During the process of porting FoxNet to ML/OS I had the chance to evaluate the validity of the FoxNet developers' claims. Having some familiarity with standard UNIX protocol stacks, I did find that in terms of modularity of design, FoxNet is superior.

The modularity promised by FoxNet developers is very well manifested. Having to change the device layer required very little understanding of any other parts of the system. Moreover, once the device layer was working, all the other layers began to work, requiring only very small modifications.

## 4.4.2 Negative aspects of FoxNet

Although an improvement over standard network stacks, FoxNet is far from being perfect. During the porting process I visited CMU to work with some of the people responsible for FoxNet. At one point during my visit I got a very insightful answer to a very difficult question that from Herb Derby. I was very impressed because Herb was not involved with FoxNet from day one. I asked how he was able to gain such a deep understanding of the system. He said, "I've been staring at this code for 2 years now."

I think there is something wrong with FoxNet if it requires 2 years for a very competent developer to understand it. A full TCP/IP stack can be implemented in C in a couple of months. Something is wrong if a TCP/IP stack written in an advanced language require 2 years of learning to be understood.

### 4.4.2.1 The FoxNet papers are outdated

The way FoxNet is described in the original publication, it seems very clean-cut and simple [1]. Their definition of the generic PROTOCOL signature is fairly straightforward. Unfortunately FoxNet has changed significantly since that time. The PROTOCOL signature looks almost nothing like it did in the original FoxNet paper. This is the signature as it appears in the FoxNet paper:

```
signature PROTOCOL =
  sig
    eqtype address
    eqtype address_pattern
    eqtype connection
    type incoming_message
    type outgoing_message

    val initialize: unit -> int
    val finalize: unit -> int

    val active_open: address * (incoming_message -> unit) -> connection
    val passive_open: address_pattern * (incoming_message -> unit)
        -> (connection * address)
    val close: connection -> unit
    val abort: connection -> unit
    val send: connection -> outgoing_message -> unit

    type control
```

```

type info
val control: control -> unit
val query: unit -> info

exception Initialization_Failed of string
exception Protocol_Not_Initialized of string
exception Invalid_Connection of connection * address option * string
exception Bad_Address of address * string
exception Open_Failed of address * string
exception Packet_size of int
end

```

This signature is elegant and fairly straightforward. A connection is opened actively by calling **active\_open** with the address of the host and the handler for incoming packets. Connections from addresses that match a certain pattern can be accepted by calling **passive\_open** with the address pattern and a handler for incoming messages. Packets are sent using **send** and connection is closed using **close**.

This is the PROTOCOL signature as it actually appears in the code:  
signature PROTOCOL =  
sig

```

structure Setup: KEY
structure Address: KEY
structure Pattern: KEY
structure Connection_Key: KEY
structure Incoming: EXTERNAL
structure Outgoing: EXTERNAL
structure Status: PRINTABLE
structure Count: COUNT
structure X: PROTOCOL_EXCEPTIONS

exception Already_Open of Connection_Key.T
type connection_extension
type listen_extension
type session_extension

datatype connection = C of {send: Outgoing.T -> unit,
                             abort: unit -> unit,
                             extension: connection_extension}

datatype listen = L of {stop: unit -> unit, extension: listen_extension}
datatype handler = H of Connection_Key.T
-> {connection_handler: connection -> unit,
    data_handler: connection * Incoming.T -> unit,

```

```

        status_handler: connection * Status.T -> unit}

datatype session = S of {connect: Address.T * handler -> unit,
                        listen: Pattern.T * handler * Count.T -> listen,
                        extension: session_extension}

val session: Setup.T * (session -> 'a) -> 'a

end

```

Obviously this signatures has very little in common with the original one. There are three data types, **connection**, **handler**, and **session**, which are somehow used to start a connection. One cannot initiate a network connection by just calling **active\_open** anymore.

#### 4.4.2.2 Lack of documentation

It is my firm belief that all software should be fully documented. In all the Software Engineering classes I have taken at MIT I was taught that code should read like a document. Apparently FoxNet developers do not share this point of view, as there are very few comments in the code. Even in places where comments exist, they are hard to understand and not very useful. This lack of useful comments, in addition to all the problems with the SML/NJ development environment, made learning FoxNet a very long, painful, and humbling process.

#### 4.4.2.3 Signatures do not substitute for documentation

One of the claims made by FoxNet developers is that most of their code can be understood by studying the signatures alone. There is certainly some truth in that statement, but it's not entirely accurate. First, even reading the signatures is not as easy as it sounds, for reasons outlined in section 4.3.2. Putting that aside and assuming that after some time all the definitions of relevant signatures are found and read, very limited understanding is achieved.

What FoxNet people are trying to do here is substitute signatures for documentation. Unfortunately that does not really work well. Recall that signatures only contain type information. That makes reliance on signatures alone equivalent to reliance

on the type information to gain most of the understanding about the system. Types simply do not contain this much information. In my opinion type information is absolutely necessary, but by no mean sufficient to gain full understanding of the system.

#### ***4.5 Flux OS Toolkit***

As explained before ML/OS uses OS Kit extensively. It was absolutely essential for implementing the device layer of FoxNet. My experience with OS Kit has been only a pleasant one. (Although it did take a long time to get the Flux project to send us a release). All the interfaces were clearly documented and well designed. The entire OS Kit compiled out of the box, with no modification necessary. The network driver interfaced worked exactly as explained in the documentation, with no glitches or surprises.

OS Kit's clean interfaces and flawless functionality and ease of use are somewhat of a surprise. OS Kit is implemented in C and assembly. Neither one of these languages provides tools for building modular systems with narrow and clean interfaces. The people behind OS Kit achieve this modularity by carefully designing conventions for interfaces and carefully designing the system as a whole. More importantly, though, OS Kit comes with documentation which clearly explains all the interfaces, and design rationale behind them. Moreover, the code itself is commented thoroughly, and provides plenty of examples of how to use various components.

The ease with which OS Kit was integrated with ML/OS and the amount of effort it took to port FoxNet goes to illustrate an important idea. Language tools for modularity, such as strong typing, and SML's module system, are neither necessary, nor sufficient for building robust, elegant systems. Careful design, conventions, and thorough documentation can compensate for the deficiencies of the language. The opposite is also true.

## 5 Conclusion

During my work of adapting FoxNet to ML/OS I worked with five complicated systems. I had to understand most of SML, parts of SML/NJ internals, most of FoxNet, most of ML/OS, and parts of Flux OS Toolkit. To make things more complicated, all these were moving targets. I started working with SML/NJ version 109.16 and the finished with version 109.30. FoxNet tracked the SML/NJ versions in small jumps over every few versions. OS Kit also went through at least three different versions.

The process of understanding FoxNet was the hardest, but it gave me an opportunity to evaluate those things about all the systems involved which made it hard, as well as those things which made it easy. I have also thought about features that, if present, would have made the process easier.

SML as a language could benefit from a simpler module system. Having an ability to write low-level code in SML would also improve it. It would eliminate the need to drop into C code, and make SML more portable.

SML/NJ as a development environment could benefit from a number of tools:

- A source-level debugger
- A tool to find a specific function definition in a large body of code
- A tool to find all instances of a function application in a large body of code
- An improved visualization tool for module-dependency diagrams
- A tool that will report the inferred type of a variable
- A tool to find a specific signature definition in a large body of code
- A tool to find all instances of a functor application in a large body of code

Adding these tools to SML/NJ would make it into a much more pleasant and productive development environment.

FoxNet has grown from an elegant system described by its authors into a huge and complicated behemoth. It could benefit from thorough documentation, which would make it easier to understand and use.

The most important thing that I have learned during this work is that the implementation language is not the silver bullet. The modularity and abstraction tools present in SML, such as strong typing, higher-order functions, and advanced module system only go so far to build an elegant, modular system. It is true that SML's module system can enforce modularity by checking the interfaces. That does not, however, help to modularize systems where the module interfaces are not sufficiently narrow. Neither does it guarantee full understanding of the interfaces if no documentation is present. If the system is well designed and well built, however, the module system can go a long way to help the designers and implementors stick to their design. It can enforce the interfaces and raise warning signs when they are not implemented.

## ***5.1 Future directions***

Having a working TCP/IP stack in ML/OS creates a number of opportunities for further work. It is an ideal application for another idea under development in the Express project. This idea is addition of data-flow analysis to the SML/NJ compiler to enable it to optimize across computational transducers [7].

*A computation transducer* is a structure similar to a traditional electrical transducer. It simply receives some input from whatever structures happen to be upstream. It then processes the input in some way, and sends the results downstream. A good example of computational transducers at work is a simple UNIX text processing command line such as: **gunzip | spell | sort | uniq | lpr**. This command line first uses **gunzip** to decompress text from a file, with a **gunzip** being the first transducer whose input comes from a file, and whose output goes to the **spell** transducer. The **spell** program checks the spelling of the text and sends the output to the **sort** transducer, which sorts the uncompressed and spell-checked text. Next, the **uniq** transducer removes duplicate lines from the text and finally sends it to the **lpr** transducer which converts the text to a format understood by the printer, and sends it to the printer.

A network protocol stack, such as TCP/IP can also be viewed as a transducer pipeline. In this pipeline, every layer is a transducer. This is easy to see since every



layer gets a packet of input from the layer upstream, processes the packet according to the protocol, and then sends the packet downstream.

Unfortunately, FoxNet's current implementation structure does not easily follow this transducer pipeline model. To make FoxNet follow this model more closely it must be re-implemented in such a way that every layer is a functional co-routine. Then, if data-flow analysis optimizations are added to the SML/NJ compiler, it is likely that the compiler will be able to optimize across these co-routines, resulting in a tightly integrated protocol stack. This will theoretically provide the best of both worlds: every layer in FoxNet will be implemented separately, as designed. Yet, the resulting stack will be tightly integrated and will not suffer from the performance losses which are present in the current FoxNet implementation.

Having a TCP/IP stack also creates an opportunity for experimenting with various operating system services that use networks. These include web servers, network file servers, and so on.

## 6 References

- [1]. Edoardo Biagioni, Robert Harper, Peter Lee, Brian G. Milnes. Signature for a Network Protocol Stack: A Systems Application of Standard ML. *In 1994 ACM conference on Lisp and Functional Programming*, Orlando, FGV, June 27-29 1994.
- [2] Postel, J. (ed.), Transmission Control Protocol - DARPA Internet Program Protocol Specification, RFC 793, USC/Information Sciences Institute, September 1981.
- [3] Postel, J. (ed.), Internet Protocol - DARPA Internet Program Protocol Specification, RFC 791, USC/Information Sciences Institute, September 1981.
- [4] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.
- [5] Bryan Ford, Keven Van Maren, Jay Lepreau, Stephen Clawson, Bart Robinson, and Jeff Turner. *The Flux OS Toolkit: Reusable Components for OS Implementation*. Department of Computer Science, University of Utah, 1997
- [6] Packetfilter – Ethernet packet filter. DEC/OSF1 manual page.
- [7] Olin Shivers. Control-Flow Analysis of Higher-Order Languages. Ph.D. dissertation, Carnegie Mellon University, May 1991.
- [8] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. *USENIX conference*, San Diego, CA, January 25-29, 1993.
- [9] BPF – Berkeley Packet Filter. FreeBSD manual page.
- [10]. Eric Cooper, Robert Harper, and Peter Lee. The Fox Project: Advanced Development of Systems Software. School of Computer Science, Carnegie Mellon University, 1991.
- [11] Edoardo S. Biagioni. A Structured TCP in Standard ML. School of Computer Science, Carnegie Mellon University, July 1994.
- [12] L.C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
- [13] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. The MIT Press, 1997.
- [14] Barbara Liskov, Alan Snyder, Russell R. Atkinson, Craig Schaffert: Abstraction Mechanisms in CLU. *In ACM Conference on Language Design for Reliable Software, March 28-30, 1977*.

- [15] Samuel P. Harbison. *Modula-3*. Prentice Hall 1992.
- [16] Richard Rashid, Daniel Julin, Douglas Orr, Richard Sanzi, Robert Baron, Alessandro Forin, David Golub, Michael Jones. Mach: A System Software kernel. *In the 34th Computer Society International Conference*, February 1989.
- [17] Richard Rashid, Robert Baron, Alessandro Forin, David Golub, Michael Jones, Daniel Julin, Douglas Orr, Richard Sanzi. Mach: A Foundation for Open Systems. *In the Second Workshop on Workstation Operating Systems*, September 1989.
- [18] Matthias Blume. A Compilation Manager for SML/NJ. Department of Computer Science, Princeton University, February, 1996.
- [19] Sean W. O'Malley and Larry L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2), May 1992.
- [20] Andrew W. Appel. A Critique of Standard ML. Princeton University, November, 1992.
- [21] David C. Plummer. An Ethernet Address Resolution Protocol. RFC 826, November, 1982.
- [22] Andrew Tolmach, Andrew Appel. A Debugger for Standard ML. *Proceedings of the 1990 ACM conference on Lisp and Functional Programming*, Nice France, June 1990.
- [23] George Springer and Daniel P Friedman. *Scheme and the Art of Programming*. McGraw-Hill Book company, December 1990.