FLECS: A Framework for Rapidly Implementing Forwarding Protocols

Mirza Beg

David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
mbeg@cs.uwaterloo.ca

Abstract. Design, implementation and deployment of network protocols is a challenging and difficult task. Determining their correctness and feasibility for large-scale networks is even more complicated. This paper presents FLECS, a framework for fascilitating implementation of forwarding protocols for packet-switched networks. We build upon the observation that the forwarding functionality can be modeled as a combination of well-defined but customizable components, the functionality of each component is constrained by the fundamental axioms of communication. FLECS provides a protocol specification language and automatically generates the protocol implementation from the specification.

1 Introduction

Designing, implementing and deploying network software is an expensive and time-consuming process. As a result, modular network architectures have gained significant interest in the networking research community. Modular architectures are ideal vehicles to design, develop, test and optimize individual components of communication protocols.

In this paper we describe FLECS, a framework that employs modularization to quickly implement forwarding functionality of communication protocols. Existing research in protocol prototyping is generally directed towards optimization and performance enhancement techniques [15,11]. Current systems lack a solid theoretical foundation, which makes it almost impossible to formally analyze their behavior with respect to forwarding. Notable exceptions include [3,14], which study the underlying principles of connectivity in communication protocols. In contrast, our work builds on an axiomatic basis for expressing communication primitives that provides a theoretically sound framework for expressing fundamental inter-networking concepts such as deliverability of messages. In particular, we use the axiomatic basis to derive and implement a *universal forwarding engine*, constrained by the axioms of our theoretical framework. We do so by using meta-compilation techniques to rapidly generate protocol implementations for a variety of forwarding schemes. A parallel stream of research has made an attempt to define communication invariants using axioms [5,4].

J. Zhou (Ed.): Complex 2009, Part II, LNICST 5, pp. 1761–1773, 2009.

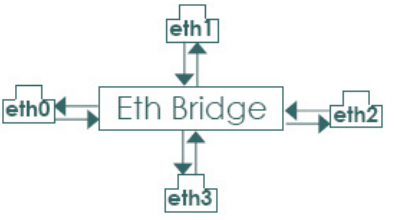

Fig. 1. Ethernet Bridge in FLECS

The axiomatic framework defines abstract components called *abstract switching elements* or ASEs. This facilitates the overall protocol design by dividing it into sub tasks and makes use of the divide-and-conquer strategy to simplify complex forwarders. The axioms in the framework help constrain the behavior of ASEs as communication protocol components in contrast to prior work, where each module can perform arbitrary processing actions.

We illustrate the concepts behind the design of FLECS using *Ethernet Bridging* as an example. Figure 1 shows the configuration of a learning Ethernet bridge. The model only requires a single ASE called EthBridge. The corresponding FLECS implementation is shown in Figure 2.

In the given model, EthBridge is directly connected to all the network interfaces (in this case four, i.e. eth0, eth1, eth2 and eth3). A packet arriving at any interface is forwarded to the EthBridge ASE which looks at the Ethernet destination (`dest_mac`) and source (`src_mac`) (Figure 2(b), lines 8-9). Each arriving packet is forwarded based on a lookup of the switching table and the ASE learns the reverse path towards `src_mac` and updates the switching table if necessary.

An equivalent implementation of the Ethernet bridge takes more than 3000 lines of code in FreeBSD. This work also presents encouraging results from our experience with implementing the universal forwarding engine. The project was undertaken with the following goals.

- Implement fundamental packet processing operations that can be used to compose complex packet forwarding schemes.
- Define a meta-language to specify packet forwarders and demonstrate its feasibility by implementing non-trivial forwarding schemes.
- Implement tools to auto-generate runnable forwarder implementations from the specifications written in our meta-language.

The rest of the paper is organized as follows. Section 2 gives an overview of related work followed by a brief restatement of the axiomatic formulation from [4], in Section 3. Section 4 examines the FLECS framework and its core components. Section 5 describes the implementation of FLECS. Implementation details and detailed protocol examples have not been included due to space limitations. Section 6 illustrates the practical capabilities of our framework by compactly describing the forwarding scheme in IP. Section 7 evaluates the effectiveness of our approach and we end with conclusions and future work in Section 8.

```
EthBridge bridge {

 control {
  [*, *] ->
    [setup/none][forward/none];
 }

 switching {
  [eth$i, *] -> [eth-$i, null];
 }
}

config(eth0, eth1, eth2, eth3)
{
    eth0 <-> bridge <->eth1;
    eth2 <-> bridge <->eth3;
}
```

(a)

```
DEFINE ETHERNET_ADDR_LEN    6
DEFINE DEST_MAC_OFFSET      0
DEFINE SRC_MAC_OFFSET       6

ASE EthBridge {
 peek {
  READ {
   dest_mac DEST_MAC_OFFSET
            ETHERNET_ADDR_LEN
   src_mac  SRC_MAC_OFFSET
            ETHERNET_ADDR_LEN }
  CONTROL { dest_mac }
 }

 forward { LOOKUP { dest_mac } }

 setup {
  UPDATE { * src_mac prev null }
 }
};
```

(b)

Fig. 2. (a) Ethernet Bridge configuration represented in FLECS (ethbridge.flecs) (b) Definition of the EthBridge ASE in FLECS (ethbridge.ase)

2 Related Work

Our work is related to a handful of attempts to build engines for rapid protocol prototyping. It also relates to work in understanding the architecture of the Internet. The axiomatic framework described in [5,4] succinctly formalizes the design principles behind communication protocols and provides a basis for formal reasoning about their properties. We briefly describe the axioms in the next section. FLECS attempts to implement the constraints defined by the axioms, using Click [8,7], whereas other approaches like [1,2] fail to build upon a sound theoretical framework.

Click defines a flexible, modular architecture for building configurable routers. Click routers can be configured by connecting Click components, called *elements*, in a directed graph. Each element defines a simple packet processing operation, such as queuing, scheduling, switching, and interfacing with network devices.We differ from this approach in that we specify protocols at a higher level of abstraction rather than in a general-purpose programming language. In addition, our design constrains the programmer according to the axiomatic formulation of packet forwarding [4]. We find Click to be complementary to our work and indeed we use it to build the first prototype of our system.

Estelle (Extended State Transition Language) [6] is a format description technique to describe communication protocols and services developed within the International Standard Organization (ISO). This technique is based on an extended

finite state transition model. The Estelle framework consists of objects called *modules*. An Estelle specification is a set of cooperating modules, interacting with each other by exchanging messages through links called channels.Our approach has several similarities with Estelle. However FLECS is unlike Estelle in that it strives to present a higher level of abstraction to the programmer and constrains the design in accordance with the axiomatic principles. Approaches like SDL [12], LOTOS [13] and Esterel [15], also describe techniques to express communication protocols using formal descriptions, like Estelle. Instead of expressing protocols in completely abstract terms, they use an approach that requires protocols to be specified in an implementation oriented formal description. The code generated is generally in the form of a skeleton that must be completed by the programmer.

FLECS represents a middle ground approach compared to previous approaches to protocol design. It allows the user to define forwarding protocols in a domain specific language constrained by the axioms of communication; yet it retains the clarity and simplicity in design that enables us to prove some essential properties of protocols.

3 Background on the Axioms of Communication

The axiomatic formulation given by Karsten et al. [4] describes the properties of the "leads to" relation denoted as $\rightarrow$. In these axioms the ASEs are denoted by letters A, B and C having input and output ports for inter-ASE communication. At ASE B, the input port from predecessor A is denoted as ${}^{A}B$ and the output port to a successor C is B^{C}. A variable port is denoted as x. The unit of communication between ASEs is a message m. A message m that exists at a port x is denoted as $m@x$. An ASE maintains a private set of mappings, called the *s*witching table. The switching table at ASE B is denoted as S_B and contains mappings $\langle A, p\rangle \mapsto \{\langle C, p'\rangle\}$ from an ASE-string pair $\langle A, p\rangle$ to a set of ASE-string pairs $\{\langle C, p'\rangle\}$. The switching table can be queried through a lookup operation $S_B[A, p]$. The "leads to" relation is defined by the following four axioms:

LT1. (Direct Communication)
$\forall A, B, m : \exists A^{B}, {}^{A}B \iff m@A^{B} \rightarrow m@{}^{A}B.$

LT2. (Local Switching)
$\forall A, B, C, m, p, p' : \exists {}^{A}B, B^{C} \wedge \langle C, p'\rangle \in S_B[A, p] \Longrightarrow pm@{}^{A}B \rightarrow p'm@B^{C}.$

LT3. (Transitivity)
$\forall x, y, z, m, m', m'' : (m@x \rightarrow m'@y) \wedge (m'@y \rightarrow m''@z) \Longrightarrow m@x \rightarrow m''@z.$

LT4. (Reflexivity) $\forall m, x : m@x \rightarrow m@x$

These axioms constrain ASE packet processing. LT1 denotes direct communication between ASEs A and B. This is possible if and only if A and B are connected to each other by a link. Axiom LT2 expresses the lookup and switching capability of an ASE. Note that in the theoretical model a packet pm is logically split into a header prefix p and the opaque message m during each local switching step. LT2 also covers any form of multi-destination forwarding, such as multicast, since the set $S_B[A, b]$ may have multiple elements. LT3 describes

transitivity over direct communication and local switching to splice the individual forwarding steps together. These three axioms naturally express the simplex forwarding process in a communication network, where, potentially, at each forwarding step, a forwarding label is swapped. Axiom LT4 specifies reflexivity for simplification of certain formal proofs.

3.1 Constraints Imposed by the Axiomatic Basis

The axiomatic basis imposes stringent constraints on the behavior of an ASE. These constraints apply to two main aspects of ASE design.

Inter-ASE Communication: These constraints arise directly from the axioms themselves. LT1 restricts each ASE by only allowing direct communication between neighbors. Two Ases are neighbors if and only if they are directly connected to each other. The second constraint arises from LT3. This bounds the overall connectivity of an ASE by the transitive closure of direct communication and local switching.

Processing within an ASE: The first constraint is that the ASE is not allowed to overwrite or redefine the main loop which forms the core of ASE processing. This prohibits the user from defining completely new ASEs in the framework. The second constraint is imposed by the processing patterns. The ASE is restricted to a small well-defined set of patterns. Any ASE specific processing must be defined by specialization and configuration of the patterns.

4 Framework

There are two main considerations which drive the design of the meta-language in FLECS. First, our protocol specification language should comply with the axiomatic fundamentals [4], which constrain packet processing in ASEs. Second, FLECS should allow programmers to specify complete protocol functionality. The routine tasks of packet manipulation can be extracted as a super component and can be reused for different implementations instead of being re-written from scratch [9,10]. This enables the programmer to automate the task of protocol composition from a minimum set of specifications.

Restricting the programmer to a limited domain specific language constrains the design choices for the protocol. An obvious benefit of using Flecs is that the programmer does not have to bother with the intrinsic details of networking which is common in protocol implementations. A less obvious benefit is that the programmer is restricted from making bad design choices.

4.1 Object-Oriented Design of ASEs

FLECS models fundamental protocol abstractions as objects, represented by ASEs. The framework predefines a Base ASE (BASE) and the programmer can implement new ASEs by refining BASE to produce ASEs required to construct a specific protocol. Figure 3 illustrates the general design of the FLECS framework.

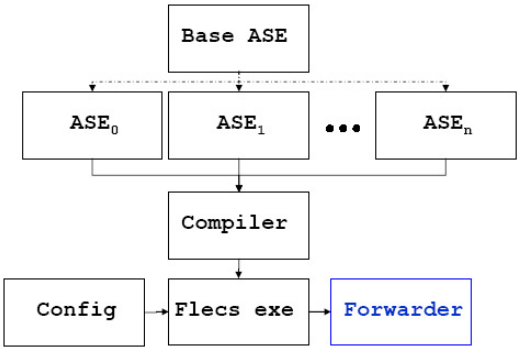

Fig. 3. The Design of the FLECS Framework

It depicts the inheritance of ASEs from BASE to compose the final forwarder. A protocol instance is made up of ASE instances, connected together to form a configuration graph. Representing protocol abstractions this way not only achieves our goal of constraining ASEs using our axiomatic formulation, but it also supports our secondary goal of dividing the functionality into smaller components, hence making the specifications simpler and easier to write.

Object-oriented programming is well-suited for representing the ASEs. One characteristic of FLECS is that it partitions protocol state such that each ASE operates on its own local state information. Object-oriented design fosters this way of thinking by packaging related meta-data and procedures together within an ASE. Another benefit is that object-orientation provides inheritance as an in-built language discipline for supplying packet processing functionality and data structures from the BASE. It should be noted here that there are certain protocol specific functions, such as TTL decrement or checksum re-computation in an IP Router, which are difficult to generalize. The framework allows the programmer to include arbitrary functions in the ASEs to make the implementations interoperable within the existing architecture.

It should be noted that FLECS is object-oriented only with respect to the protocol abstractions built in the BASE. FLECS programmers cannot define arbitrary, new and unconstrained ASEs. The language specifications only allow the programmer to create specializations of the BASE. This makes FLECS specific for packet processing, and unlike a general purpose, object-oriented language, it does not explicitly provide the programmer with language-level constructs to optimize protocol software. This restriction allows us to exploit the knowledge of common patterns in protocol operations for internal optimizations. This gives additional power to FLECS over hand-coded optimizations by reducing per-layer overhead, even though the protocol graph is not determined until run time.

FLECS represents fundamental tasks in protocols as *packet processing primitives*. It predefines a collection of primitives, using which any arbitrary network protocol can be easily composed. These primitives include `pop` (to remove a prefix of the packet header), `push` (appends a prefix to the packet header), `send`,

`receive`, `lookup` (looksup the switching table) and `update` (inserts entries to the switching table).

4.2 Internals of an ASE

It turns out that the forwarding functionality of an ASE can be specified through a small number of *processing patterns*, using the primitives described above. We use patterns and primitives to abstractly describe the design of ASEs. We logically partition overall ASE processing into several processing patterns. Each pattern defines either a *forwarding* or *control* procedure. Forwarding includes manipulation of the packet header as well as packet switching based on a *switching table* lookup. This forwarding operation is along with the necessary modifications to the packet is defined by the `forward` pattern. Control patterns are designed to update local or remote ASE state. These include `setup` (updates ASE state), `resolve` (performs remote lookup), `respond` (responds to a remote lookup request) and `rupdate` (updates ASE state based the reply for `resolve`).

Patterns model complex operations of packet processing than the aforementioned primitives. In fact, each pattern can be composed from a set of primitives arranged in a block of code using regular programming constructs. For different ASEs the same pattern can be configured differently, possibly with different options, to yield different functionality. Essentially, it is the processing patterns that implement the constraints imposed by the axiomatic formulation.

ASEs are a particularly novel aspect of FLECS. Each ASE operates on a specific prefix of the packet header. It extracts the relevant information from this header prefix and uses it for processing the packet and forwarding. An ASE can be instantiated multiple times in the same configuration. An active instance of an ASE in a particular forwarder configuration can emulate a *protocol layer* such as IP.

ASEs make processing and switching decisions based on values retrieved from the packet header. They can carry out complex operations such as swapping header fields, encapsulating a message with a new header or removing header prefixes as required by the specific protocol. The functionality of an ASE is defined by the processing patterns it implements (e.g. `forward` pattern in EthBridge, Figure 2(b)). At runtime, the behavior of an ASE is determined by its local state. ASEs maintain their local state in *control* and *switching* tables. These are initialized for each instance of an ASE in the configuration.

The pseudo-code in Figure 4 shows the main processing routine for an ASE. When a packet arrives at an ASE, it is handed to its `process` routine. `Process` extracts the relevant fields from the packet header and looks up the control table to determine which patterns are to be executed on the packet. If there is no matching entry for a particular packet in the control table, the packet is discarded. Otherwise, the patterns returned by the lookup are sequentially executed on the packet.

The control table determines the patterns to be executed on different packets received by the ASE. Entries in the control table specify mappings as $[Ase_x,\ p'] \rightarrow \{[pattern/subtype]\}$, where Ase_x is the ASE from which the packet was sent and p' is a set of strings; the pair forms the *key* for that entry. The

```
1   process(Packet *p, AseRef prevAse) {
2     s = peek(p)
3     patterns[] = lookup(control, {prev, s})
4
5     for (each pattern in patterns[]) {
6           if (p) execute(pattern, p)
7     }
8   }
```

Fig. 4. The Main Processing Routine of an ASE

key maps onto a set of patterns. As can be noted from the table structure, the loop enforces an order on pattern execution. This is an additional constraint not captured by the axioms. Switching table entries are mappings of the form $[Ase_x,\ p'] \rightarrow \{[Ase_{y_i},\ p'']\}$. In the forward pattern, a packets forwarding path is determined by using previous ASE and a set of header fields as the lookup value. The lookup returns a set of ASE and string pairs, and copies of the packet are then forwarded to each of those ASEs along with the string p'' which is used as a name for the destination ASE of this packet. Note that this gives us the ability to handle broadcast, multicast as well as anycast packets.

5 Implementation

We have implemented FLECS using Click [8], a framework for building flexible, configurable routers. We use a hybrid approach of class inheritance and meta compilation to produce the desired Click implementation and configuration. The complete protocol development process in FLECS is shown in Figure 5 where BASE is implemented as a Click element. ASE specifications are compiled by the `asec` compiler to generate code for the corresponding Click elements. ASEs are implemented as complex Click elements, extending BASE to inherit the generic functionality. Given the ASE design, it can easily be noticed that a traditional protocol layer can be modelled as an ASE. A particular protocol configuration

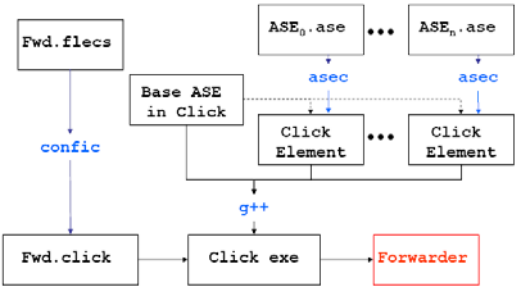

Fig. 5. FLECS Implementation in Click

might require multiple instances of the same ASE to simulate a single layer. A specific FLECS configuration can be translated into the corresponding Click configuration using the `confic` compiler. The elements are compiled to form the Click executable which interprets the configuration file to produce the desired forwarding functionality represented by Forwarder in Figure 5.

Succinctly stated, the FLECS framework is comprised of two meta-compilers and the respective meta-language specifications. The ASE compiler, called `asec`, compiles ASE specifications written in ASE Description Language (ADL) to generate Click element code representing the ASE. The configuration compiler, namely `confic`, compiles configurations specified in FLECS Configuration Language (FCL) to produce a Click configuration. It should be noted here that FLECS does not depend on any specific functionality of Click, rather we can implement the FLECS compilers in any reasonable packet processing engine.

6 Examples

In this section, we discuss how the FLECS framework can be used to implement some well-known and non-trivial forwarding protocols. In the following sections, we discuss a couple of protocol implementations with diverse compositions. In general, the framework can be used to implement forwarding in DNS [18,19], Mobile IP [17], Dynamic Source Routing [16] and other multicast and anycast protocols with little effort. We discuss the implementation details of an IP forwarder.

6.1 IP Forwarding

A simple IP forwarder can be modeled in FLECS as shown in Figure 6. An IP packet arriving at a network interface is forwarded to the corresponding `ETH` ASE. `ETH` ASE's switching lookup on the Ethernet destination and protocol determines whether to forward the packet to the `IP` ASE, `ARP` ASE or drop it. If the intended Ethernet destination of the packet differs from the Ethernet address assigned to

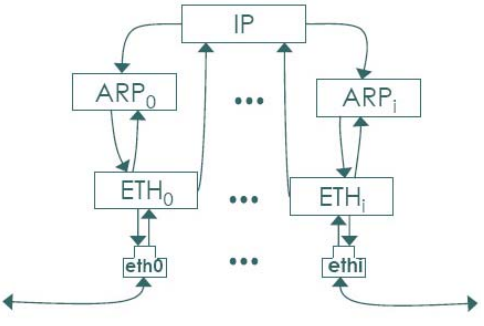

Fig. 6. IP Router in FLECS

the respective ETH ASE, the packet is dropped, otherwise the Ethernet header is popped off and the packet forwarded to IP ASE.

The IP switching table lookup determines the interface to forward the packet and passes it on to the corresponding ARP ASE, annotating the packet with the next hop IP address given in the switching table entry of IP ASE. Protocol specifications are not given due to space constraints. The IP switching table is the routing table of the IP Router. This can be configured manually during initialization or rupdate in the IP ASE can be defined for handling routing table updates. ARP looks up its switching table to resolve the next hop IP address, pushes the resolved Ethernet address and forwards the packet to the ETH ASE which recasts the packet in the correct Ethernet header and relays it to the respective interface.

ETH and ARP ASEs are also configured to handle ARP requests and ARP replies, hence the extra arrows between them. The ARP ASEs are configured with the local ip and the corresponding Ethernet address.

7 Evaluation

Figure 7 demonstrates the feasibility of using the FLECS framework to prototype forwarding functionality of communication protocols. It shows the difference between the lines of code written by the programmer in FLECS compared to the number of lines of code generated by the asec compiler for different protocol implementations. For example an Ethernet bridge configuration can be specified in FLECS along with its configuration for a two network interfaces in less than thirty lines (2). The same implementation in Click results in more than two hundred lines of code. FLECS produces the Click implementation from the specification in less than a hundred lines of code. This does not include the generic code inherited from BASE. A comparable Ethernet bridge written for FreeBSD is more than 3K lines of code. This difference between implementations in different

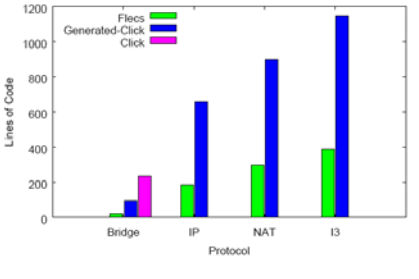

Fig. 7. Comparison of the number of lines of code in the .ase file with the number of lines generated by the *asec* compiler. Click implementation of Etheret bridge is in 236 lines of code.

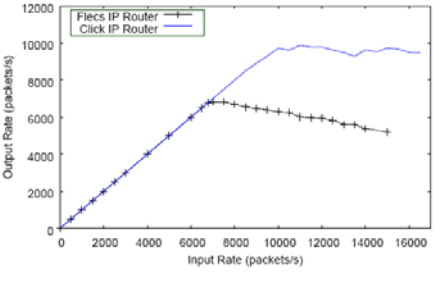

Fig. 8. Forwarding Rates of an IP Router in FLECS

environments results partly because of our generalized nature of the framework, reusable code base and inheritance model and partly because other implementations have a big chunk of error handling and optimization code. This includes optimizations such as the spanning tree protocol implementation and network interfacing with the LAN driver in FreeBSD. We specify the IP forwarder in 187 lines of ACL. The ASE compiler produces 657 lines of Click code for IP. A comparable Click implementation for IP forwarding (not using the Base class functionality provided by FLECS) takes more than 2K lines of code. A similar implementation in Linux would probably consist of several thousand lines of code. FLECS generalizations not only reduces the amount of work the programmer has to put in to prototype a specific forwarder, but also makes it easier to locate bugs which might be difficult to find due to the complexity of a code base.

We also evaluate the cost of adhering to the axiomatic constraints and the generalizations implemented in FLECS. Figure 8 characterizes the performance of an IP forwarder in FLECS by measuring the rate at which it can forward 64 byte packets, when compared to a Click implementation of a comparable IP forwarding configuration. This analysis presents the router behavior under different workloads. The experiments were conducted by running the implementations in user-level Click, on the same machine. The FLECS-generated implementation peaks at 7,000 packets where as the Click implementation peaks at 10,000. The resulting ASEs from FLECS were modified to use an optimized data structure to hold extracted values from the packets and table lookups.

We expected to see some performance degradation due to the nature of generalization enforced on the ASE processing. The results show a performance hit of 30%. This is an encouraging result considering that we have not yet incorporated any optimization techniques into our compilers and we are performing at 70% of a protocol specific implementation. We observe that each IP packet passes through five complex elements, each performing at least two lookup operations, compared to thirteen simple elements in the Click implementation with a single lookup amongst them. Optimized data-structures for holding the control and switching tables would probably result in regaining s significant portion of the performance loss. Furthermore the `asec` compiler can utilize domain knowledge to produce optimized forwarding code.

8 Conclusions

This paper describes FLECS, a framework for rapid protocol prototyping. FLECS applies a divide-and-conquer strategy to decompose complex protocols into a combination of ASEs. ASEs can support a wide variety of complex packet forwarding tasks through composition.

There are a three main advantages of using FLECS for implementing packet forwarders. The first is that by using the FLECS framework the time to design and implement communication protocols can be drastically reduced. The second advantage is that by adhering to the *axiomatic basis*, the generalized proofs of

correctness of patterns can eventually be used in augmentation with automated theorem provers to prove correctness of protocol implementations. The third advantage emerges from our use of the object-oriented inheritance model to extract the generic functionality and the main processing loop in the Base. This not only constrains design choices but also reduce the protocol specifications to mere data-oriented specializations of the Base.

Given the current status of our work, we can implement optimization techniques available to a domain specific framework to generate very efficient implementations.

References

1. Kohler, E., Kaashoek, M.F., Montgomery, D.R.: A Readable TCP in the Prolac Protocol Language. In: SIGCOMM 1999, Cambridge, Massachusetts, USA, pp. 3–13 (1999)
2. Madhavapeddy, A., Ho, A., Deegan, T., Scott, D., Sohan, R.: Melange: Creating a 'Functional' Internet. In: EuroSys 2007: Proceedings of the 2007 conference on EuroSys., Lisbon, Portugal, pp. 101–114 (2007)
3. Clark, D.: The Design Philosophy of the DARPA Internet Protocols. In: SIGCOMM 1988, Stanford, California, pp. 106–114 (1988)
4. Karsten, M., Keshav, S., Prasad, S., Beg, M.: An Axiomatic Basis for Communication. In: SIGCOMM 2007, Kyoto, Japan, pp. 217–228 (2007)
5. Karsten, M., Keshav, S., Prasad, S.: An Axiomatic Basis for Communication. In: HotNets V, Irvine, CA, USA, pp. 19–24 (2006)
6. Budkowski, S., Dembenski, P.: An Introduction to Estelle: A Specification Language for Distributed Systems. Computer Networks and ISDN Systems 14(1), 3–24 (1988)
7. Morris, R., Kohler, E., Jannotti, J., Kaashoek, M.F.: The Click Modular Router. In: SOSP 1999, Kiawah Island Resort, near Charleston, SC, USA, pp. 217–231 (1999)
8. Kohler, E., Morris, R., Jannotti, J., Kaashoek, M.F.: The Click Modular Router. ACM Transactions on Computer Systems 18(3), 263–297 (2000)
9. Condie, T., Hellerstein, J.M., Maniatis, P., Rhea, S., Roscoe, T.: Finally, a Use for Componentized Transport Protocols. In: HotNets IV (2005)
10. Krupczak, B., Calvert, K., Ammar, M.: Increasing the Portability and Re-usability of Protocol Code. IEEE/ACM Transactions on Networking 5(4), 445–459 (1997)
11. Liu, X., Kreitz, C., Renesse, R., Hickey, J., Hayden, M., Birman, K.P., Constable, R.L.: Building Reliable, High-performance Communication Systems from Components. In: SOSP 1999, Kiawah Island Resort, near Charleston, SC, USA, pp. 80–92 (1999)
12. Tennenhouse, D.L.: Layered Multiplexing Considered Harmful. In: First International Workshop on High Speed Networking (1989)
13. Bolognesi, T., Brinksma, E.: Introduction to the ISO Specification Language LOTOS. Computer Networks and ISDN Systems 14(14), 25–59 (1987)
14. Griffin, T.G., Sobrinho, J.L.: Metarouting. In: SIGCOMM 2005, Philadelphia, Pennsylvania, USA, pp. 1–12 (2005)
15. Dabbous, W., O'Malley, S., Castelluccia, C.: Generating Efficient Protocol Code from an Abstract Specification. In: SIGCOMM 1996, Palo Alto, California, USA, pp. 60–72 (1996)

16. Bolognesi, T., Brinksma, E.: Dynamic Source Routing in Ad Hoc Wireless Networks. Mobile Computing 353, 153–181 (1996)
17. Perkins, C.: RFC 3344 - IP Mobility Support for IPv4. IETF (2002)
18. Mockapetris, P.: RFC 1034 - Domain Names - Concepts and Facilities. IETF (1987)
19. Mockapetris, P.: RFC 1035 - Domain Names - Implementation and Specification. IETF (1987)