1

FLECS: A Data-Driven Framework for Rapid Protocol Prototyping

Mirza Beg, Martin Karsten, and Srinivasan Keshav

David R. Cheriton School of Computer Science
University of Waterloo,Waterloo, ON, Canada
Technical Report CS-2008-17
{mbeg,mkarsten,keshav}@uwaterloo.ca

Summary. FLECS is a framework for facilitating rapid implementation of packet forwarding protocols. Forwarding functionality of communication protocols can be modeled by a combination of packet processing components called *abstract switching elements* or ASEs. Each ASE is constrained by the *axioms of communication* which enables us to formally analyze forwarding mechanisms in communication networks. ASEs can be connected in a directed graph to define complex forwarding functionality. In this paper we present FLECS, a framework that compiles meta language protocol specification into its Click implementation. It allows rapid prototyping through configuration, as well as specialized implementation of performance-critical functionality through inheritance.

1.1 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 [9, 22]. Current systems lack a solid theoretical foundation, which makes it almost impossible to formally analyze their behavior with respect to forwarding. Notable exceptions include [5, 10], 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 [17, 18]. This work was inspired by Hoare's axiomatic basis for programming [12] and is closely related to other work in the area of naming and addressing indirection [1, 11, 28].

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 describe the concepts behind the design of FLECS using *Ethernet Bridging* as an example. Figure 1.1 illustrates the configuration of a learning Ethernet bridge. The model only requires a single ASE called EthBridge. The corresponding FLECS implementation is shown in Figure 1.2.

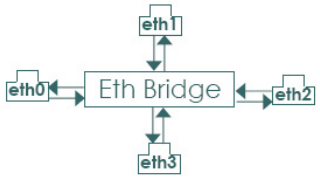

Fig. 1.1. Ethernet Bridge in FLECS.

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 1.2(b), lines 8-9). Figure 1.2(b) describes the ASE operations (patterns) on any given packet.

Packet arrival results in a sequential execution of `setup` and `forward` patterns (Figure 1.2(a), line 4). The execution of `setup` results in a new table entry that learns the reverse path towards `src_mac`. This learning action is defined by the setup operation in Figure 1.2(b), line 15. The `forward` pattern looks up the switching table for `dest_mac` Figure 1.2(b), line 13. If the path to `dest_mac` has been learnt during a previous event, then the packet is forwarded to the respective interface. If there is no specific entry for `dest_mac` in the switching table, the packet is broadcast, which is the default configuration of the switching table (Figure 1.2, line 8). Figure 1.2(a), lines 14,15 describe the connections of the EthBridge instance (`bridge`) to the network interfaces. An equivalent implementation of the Ethernet bridge takes more than 3000 lines of code in FreeBSD.

```
1  EthBridge bridge {
2
3   control {
4    [*, *] ->
5      [setup/none][forward/none];
6   }
7
8   switching {
9    [eth$i, *] -> [eth-$i, null];
10  }
11 }
12
13 config(eth0, eth1, eth2, eth3)
14 {
15     eth0 <-> bridge <->eth1;
16     eth2 <-> bridge <->eth3;
17 }
```

(a)

```
1  DEFINE ETHERNET_ADDR_LEN    6
2  DEFINE DEST_MAC_OFFSET      0
3  DEFINE SRC_MAC_OFFSET       6
4
5  ASE EthBridge {
6   peek {
7    READ {
8     dest_mac DEST_MAC_OFFSET
9              ETHERNET_ADDR_LEN
10    src_mac  SRC_MAC_OFFSET
11             ETHERNET_ADDR_LEN }
12   CONTROL { dest_mac }
13  }
14
15  forward { LOOKUP { dest_mac } }
16
17  setup {
18   UPDATE { * src_mac prev null }
19  }
20 };
```

(b)

Fig. 1.2. (a) Ethernet Bridge configuration represented in FLECS (ethbridge.flecs) (b) Definition of the EthBridge ASE in FLECS (ethbridge.ase)

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 contributions of this paper are threefold. First, it describes the design of FLECS, including its programming model. Second, it discusses the FLECS implementation using the Click modular router [20]. Third, it demonstrates the feasibility of a universal forwarding engine by building working prototypes that inter-operate with existing protocol suites.

The rest of the paper is organized as follows. Section 1.2 gives an overview of related work followed by a brief restatement of the axiomatic formulation from [18], in Section 1.3. Section 1.4 examines the FLECS framework and its core components. Section 1.5 describes the implementation of FLECS in Click. This section also gives details of our meta-language constructs with

some simple examples. Section 1.6 illustrates the practical capabilities of our framework by compactly describing some non-trivial forwarding schemes such as IP, NAT and i3. Section 1.7 evaluates the effectiveness of our approach and we end with conclusions and future work in Section 1.8.

1.2 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 [17, 18] 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 [20, 26], whereas other approaches like [19, 23] 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 [18]. 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) [4] 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 [30], LOTOS [2] and Esterel [9], 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. Although this eases the task of manual programming, the implementation is similar to one written in a general-purpose language.Although forwarders implemented in this way are

generally efficient, the programming task varies in difficulty.Others have augmented these techniques to design protocol prototyping systems with message sequence charts [14] and automated verification tools [15].

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.

1.3 Axioms of Communication

In this section we briefly restate the axiomatic framework [18] that forms the basis of this work. It formulates fundamental forwarding mechanisms in communication networks.

1.3.1 The Axioms

The axiomatic formulation 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.

1.3.2 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.

1.4 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 [18], 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 [8, 21]. 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.

1.4.1 Object-Oriented Design

Flecs models fundamental protocol abstractions as objects, represented by Ases. The framework predefines a Base Ase (Base) and the programmer

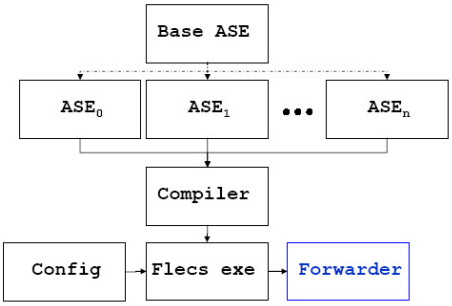

Fig. 1.3. The Design of the FLECS Framework.

can implement new ASEs by refining BASE to produce ASEs required to construct a specific protocol. Figure 1.3 illustrates the general 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 recomputation 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.

1.4.2 Packet Processing Primitives

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. We hope that these primitives are expressive enough to represent packet forwarding as well as basic control operations. Communication protocols can be represented as a sequence of these primitive operations. Since any communication protocol can be specified using a combination of these primitives, we claim that our list of primitives forms a *complete* set of packet processing operations. This set is enumerated in Table 1.1. Our primitives can be implemented in any protocol subsystem which has basic packet processing capabilities.

For packet processing and forwarding, we need to extract strings from the packet header (`peek`) and modify the header structure (`push, pop` and `swap`) as well as maintain switching tables, such as those used in NAT, i3 [28], etc. Finally, we need a few helper functions to copy a packet, create a new packet and discard ones which are not needed. In addition, we also need to send and receive packets to and from neighboring ASEs.

1.4.3 Processing Patterns

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, enumerated in Table 1.2. 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`, `resolve`, `respond` and `rupdate`.

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. The patterns are enumerated in Table 1.2. These patterns are sufficient to implement arbitrary forwarding functionality.

1.4.4 ASEs Inside Out

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

Table 1.1. Packet Processing Primitives

Primitive	Description
send(p, ase)	Sends packet p to the ASE specified by the second argument.
$\{p,\ ase\}$ = receive()	Receives a packet p from a neighboring ASE specified by ase. This information (i.e. the previous ASE) is stored and can later be used in making forwarding decisions.
$\{s\}$ = peek(p)	Returns a set of strings $\{s\}$, copied from the given packet. In its implementation it would also take a set specifying the fields to be copied by their offset and length.
p_2 = push(p_1, $\{s\}$)	Encapsulates the packet with the given set of strings. The resultant packet length increases by the cumulative length of all the strings in $\{s\}$.
p_2 = pop(p_1, l)	Removes a prefix of length l from the packet header. The length of the resultant packet decreases by l and the data in that part of the packet header is lost.
p_2 = swap(p_1, $\{s\}$)	Rewrites part of the packet header with the given set of strings. The length of the packet remains unchanged.
p = create($\{s\}$)	Creates a new packet p and populates it with the given strings $\{s\}$. The length of the new packet is the cumulative length of all the strings in $\{s\}$.
p_2 = copy(p_1)	Creates a new packet p_2 and copies the contents of p_1 into it.
drop(p)	Discards packet p. After drop is called on a p, the data in p is lost and cannot be accessed again.
v = lookup(t, k)	Returns a value object, v, with key k in the given table t. The objects represented by k and v can represent different types depending on the table t.
update(t, k, v)	Updates or inserts a table row with key k in table t with the given value object v.

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 1.2(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 1.4 shows the main processing routine for an ASE. When a packet arrives at an ASE, it is handed to its `process` routine.

Table 1.2. Processing Patterns

Pattern	Description
forward/*subtype*	Looks up the switching table to determine the next destination Ase. It also executes push/pop/pop+push/swap if specified as *subtype* and sends the packet to the next Ase. If push or swap are specified as subtype then the forward pattern expects to get the strings to be pushed or swapped from the switching table lookup. The default subtype is *none*, meaning no modifying operation is to be performed on the packet.
setup/*subtype*	Updates the switching table using information from the packet. It also executes swap if specified as the *subtype* in the case of virtual-circuit setup. By default the subtype would be *none*.
resolve	In the case where a name needs to be remotely resolved, this pattern creates a remote lookup request message and sends it towards the relevant Ase. If a packet triggered this resolution request then it is queued until a reply is received.
respond	Handles resolve requests from other Ases. It creates a new packet containing the reply for each request and sends it to the querying Ase.
rupdate	Upon receiving a reply for a resolve request this pattern updates the local state of the Ase. It also invokes the processing of any potential packets that are waiting for this update.

```
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. 1.4. The Main Processing Routine of an Ase.

`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 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.

1.4.5 Base ASE

BASE models a generic ASE by implementing the forwarding primitives and declaring the processing patterns as virtual functions.The programmer implements a specific type of ASE by refining BASE, thereby deriving ASEs that are specific to the desired protocol. A subclass is derived from BASE by providing implementations of `peek` and other patterns required for packet processing. Additional procedures may be added to refine and add functionality not currently handled by the framework, by its post-processing features.

The framework allows the derived ASEs to override certain operations in the BASE ASE. These are defined as virtual functions in the interface defining BASE. Since the base class is predefined the instances of the other operations, including most of the forwarding primitives are fixed and cannot be overridden, understanding and using the framework becomes easier. In the implementation of FLECS the keywords and their semantics are easy to learn as they are few and correspond to meaningful units of behavior.

1.4.6 Inheritance Model

We now consider how BASE allows protocol code to be inherited and integrated within a FLECS protocol implementation. In the specification of an ASE, the programmer configures the required patterns in a specified format. This augments or overwrites the code in BASE for those patterns. If an ASE does not need a given pattern, it simply does not define the corresponding configuration. Since ASEs implemented in FLECS inherit code from a single base ASE, BASE is never instantiated directly, and consequently the programmer cannot define completely new ASEs.

FLECS has two features supported by the basic inheritance mechanism just described. The first is the ability to override the BASE behavior. This permits BASE to offer default behavior even though it might not always be desired. The second feature is the ability to intermix BASE and subASE code at a finer granularity. This results from the flexibility provided by the language model to configure the patterns.

1.5 Implementation

We have implemented FLECS using Click [20], 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 1.5 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 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 1.5.

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.

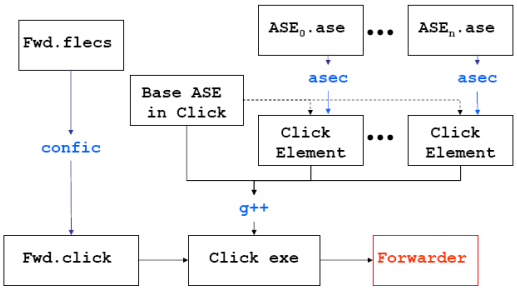

Fig. 1.5. FLECS Implementation in Click.

1.5.1 ASE Description Language (ADL)

ASEs are specified in ADL which is a formal description language subject to constraints imposed by the underlying axiomatic framework. An ASE specification contains, first, the protocol constants, pattern definitions, and references to relevant fields in the packet header (see Figure 1.2(b)). These ASE specifications are compiled by the `asec` compiler to produce the equivalent Click code.

Protocol Constants

Constant values to be used in the ASE description can be declared using the keyword `DEFINE` at the beginning of the specification file. This improves readability of the code. Usually these are protocol specific constants that specify fields in the packet header by their offset and length. Constant string values are also specified using this construct. These values are specified as hexadecimal strings using double quotes. The following examples from the EthBridge ASE define the length of the Ethernet address and the offset of the destination MAC from the beginning of an Ethernet packet respectively.

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

Extracting Header Fields

ASEs define the `peek` method to extract relevant data from the packet arriving at the ASE. The `peek` specification enumerates the set of packet fields to be used in the processing patterns, using the `READ` block. Each triple in the `READ` block of `peek` represents referencing a specific header field by a variable. The first string in the triple specifies the name of the variable by which the header field will be referenced, the second is its offset from the beginning of the packet and the third is the field length. The variable name used in the triple can be used later in the pattern definitions. An example `READ` block that reads the destination and source MAC from an Ethernet packet ASE has the following syntax:

```
READ {
  dest_mac DEST_MAC_OFFSET ETHERNET_ADDR_LEN
  src_mac SRC_MAC_OFFSET ETHERNET_ADDR_LEN
}
```

Specifying Control Values

Each ASE must specify the header fields to be used for control table lookup. This is done in the control block of `peek`. The `peek` specification would thus

contain a `READ` block and a `CONTROL` block. The `CONTROL` block specifies the fields to be looked up in the control table to determine which patterns are to be executed on the packet. The following example is from the EthBridge ASE from the introduction. The control decision on any arriving packet will be made based on the contents of its `dest_mac` field.

```
peek {
  READ { ··· }
  CONTROL { dest_mac }
}
```

Forwarding

The `forward` pattern relays the packet to a neighboring ASE or drops it based on the result of the switching table lookup. The lookup searches for a match in the switching table using the ASE from which the packet arrived denoted by `prev`, and the specified header fields as the key for the lookup. The lookup returns a result $\{[next\_Ase, \{s\}]\}$. The pattern sends a copy of the packet to the respective destination determined by $next\_Ase$, after making the necessary modifications to the packet. The set of strings {s} is used for possible packet maniputalion. Each string in {s} is given in a hexadecimal representation. This pattern also has advanced subtypes, `push, pop` or `swap`. The `push` subtype appends the set of strings {s} from the lookup result as a prefix to the given packet and `pop` removes a prefix of specified length from the header. The `swap` subtype replaces a given set of header fields with the given strings and is used for modeling circuit switching.

```
forward {
  LOOKUP { dest_mac }
}
```

The example shows a `forward` specification from the EthBridge ASE. It translates into a pattern, when executed on a packet looks up the `dest_mac` and forwards it to the $next\_Ase$. If `pop` or `swap` were specified then these operations would be performed before the packet is forwarded.

Encapsulation

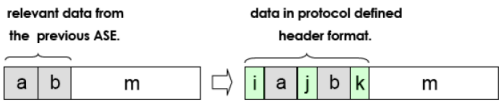

Fig. 1.6. An abstract representation of recasting data from a previous ASE into the protocol defined header format.

When packets arrive on an incoming path from the network interface, for example from a lower layer, they are decapsulated before being passed to the next higher layer. On the other hand, when the packets are passed from a higher layer to a lower layer they need to be encapsulated with the appropriate header. For example a packet arriving at an Ethernet ASE from `ARP` would need to have the correct Ethernet header added to it before being forwarded anywhere. This operation is performed by `recast`. Conceptually `recast` restructures the packet header by using the information prepended by the previous ASE and prefixing the packet with the correct header. This is illustrated in Figure 1.6. We introduce the `WRITE` block which is defined as a set of tripples. The first string in the write block is the variable containing the value and the next two strings specify the offset and length of the field to be written, respectively. For an example of recast see the IP forwarding example.

Path Setup

Control patterns are required to: 1) update local state, and 2) retrieve state information from remote ASEs and serve remote update requests. The simplest control pattern is `setup`. Upon execution, it updates the local switching table using information from the packet header. The example below shows `setup` for an EthernetBridge ASE which is invoked upon receiving any Ethernet packet. It learns the forwarding path for a packet destined to *src_mac*.

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

In this simple example, EthernetBridge would update its switching table with the entry $[*, src\_mac] \rightarrow [prev, null]$. Each pattern is aware of the interface from which the packet arrived. The identifier of that interface can be accessed from the variable *prev*. The `setup` pattern also handles virtual-circuit setup and NAT translations using its `VC` option. With this option `setup` is able to generate virtual circuit identifiers using `local_name()`, update the translation table and swap names in the packets before forwarding them. For example, in the case of NAT local_name() is expected to return an IP address and UDP/TCP port pair. This is then used to update the switching table which corresponds to the NAT translation table. These values are also used to overwrite specific values in the packet using `swap`. An example `VC` specification is given below. In this example the ASE swaps a single value which is written in the `swap` subtype.

Lookup of the switching table is specified to check whether the entry already exists. If not, then `VC` option is called and a local name is created. The `VC` block also specifies the updates to be made to the switching table. There can be multiple updates for optimization, as in the case of NAT. The `SWAP` block handles the switching of names in the packet.

```
setup {
```

```
    LOOKUP { lookup_val }
    VC {
      LOC_NAME { local_name() }
      UPDATE { * LOC_NAME prev_ase lookup_val }
    }
    SWAP { WRITE { LOOKUP_NAME OFFSET_0 LENGTH_0 } }
}
```

Remote Resolution

There are other more complex patterns that are needed for retrieving state information from remote ASEs (`resolve`), serving resolve requests (`respond`) and handling resolution replies (`rupdate`). These patterns can be defined as a combination of packet creation along with the `push` option to create the appropriate request or response.

The `resolve` and `response` patterns are comprised of `CREATE` and `PUSH` and is configured using the following template. Usually `resolve` is triggered by the arrival of a packet for which a remote name resolution is required in order to forward it correctly. In this case an appropriate request message is created and sent. The arriving message is added to the wait queue until the response is received. The `PUSH` in the case of `resolve` and `respond` is specified separately from `CREATE` as it adds the data required for the next ASE to properly format the packet.

```
resolve {
   CREATE { CREATE_LENGTH
      WRITE { data_0 OFFSET_0 LENGTH_0
              data_1 OFFSET_1 LENGTH_1 }
   }
   PUSH { PUSH_LENGTH
      WRITE { data_2 OFFSET_2 LENGTH_2
              data_3 OFFSET_3 LENGTH_3 }
   }
}
```

The syntax for `rupdate` is similar to a simple `setup` and can be specified as follows. It handles the updates to the switching table and its implementation discards the packet which is actually a response to the request sent to be resolved. It also handles the packets which are waiting for the remote resolution response by looking up the wait queue against the changes made to the switching table. The following is an example of `rupdate` specified in an address resolution protocol.

```
rupdate {
   UPDATE { * src_proto#ip prev src_hwadd }
}
```

The `respond` pattern is triggered by the arrival of a resolution request packet from a remote ASE. In response it creates an appropriate reply to the request and sends it towards the concerning ASE.

Pre-Forwarding Hooks

There are also certain operations which are specific to forwarders and have not been modeled in our framework as they do not affect forwarding decisions. These include specialized mathematical operations on certain fields of the header or the entire packet, such as TTL decrements, checksum computations, etc. Our framework allows the programmer to inline ASE methods directly into the Click implementation of an ASE. This can be done using the character `%` at the beginning of a line. For example the TTL decrement code in IP would read the TTL value from the packet, decrement it and write it back.

```
void localTTLUpdate(Packet *p) {
%   unsigned char ttl =
%     (unsigned char) p->data()[TTL_OFFSET];
%   --ttl;
%   write(p->uniqueify(), TTL_OFFSET, TTL_LENGTH,
%         (unsigned char *)&ttl);
}
```

1.5.2 FLECS Configuration Language (FCL)

FLECS configuration consists of ASE initializations and forwarding graph layout. These are specified using a formal definition language called FLECS Configuration Language or FCL. The FLECS configuration can be compiled into Click configuration using the `confic` compiler in the framework.

ASE Initialization

Initialization involves creating an instance of an ASE which includes naming the instance and specifying initial entries for the local state (`control` and `switching` tables). The following is the syntax for ASE initialization.

```
AseType AseName {
  control { /* Control Table Entries */}
  switching { /* Switching Table Entries */}
}
```

Multiple symmetric ASEs can be instantiated using the same declaration by using `$i` at the end of its name. The `$i` permits the instances to range from 1 to n, depending on the configuration of the graph. For example, in an IP router configuration, an Arp ASE instance is required for each interface and can be declared using one declaration named as `Arp$i`. `$i` can then be used in the initial control and switching table entries.

Another use of `$i` is in switching `Ase`s. If `Ase$i` is used in combination with `Ase−$i`, as in EthBridge (Figure 1.2(a)), it corresponds to multiple entries in the table resulting in forwarding packets to all the ASEs whose names have a prefix `Ase` except from the one it arrived from i.e. `prev`.

Control Table

The theoretical model described in the axiomatic basis [18] assumes complete tables. This means that all possible lookup values are handled by the control table. In reality, this assumption is not very practical for an implementation. Thus wildcard matching is introduced to handle arbitrary lookups and hence reduce the size of the tables.

The control table specifies the patterns to be executed on different packets. The control table row from the Broadcast example is

```
[*, *] -> [forward/none];
```

This entry matches all packets arriving from any Ase. It specifies that the `forward` pattern (with no options) is to be executed for all packets. The wildcard(`*`) is used to match all possible lookup values. In general, specific string patterns are used in the control table keys and a sequence of processing patterns executed on each packet. The patterns which can be used in this table are listed in Table 1.2. In addition `recast` and `drop` can also be used just like patterns in the control table.

Switching Table

Theoretically the switching table should also be complete. But in practice the same argument that applies to the control table also applies to the switching table, and we use wildcards and partial matching for feasibility of implementation. Each switching table entry maps an Ase-string pair to a set of Ase-string pairs. As in the `control` table wildcard(`*`) can be used in the key for arbitrary matching. An example from the EthSwitch Ase switching table which maps the destination MAC address to the forwarding interface is shown below. Multi-string entries are separated by `#`.

```
[eth1, 0005A23B45FF#0800] -> [IP, null];
```

Configuration Graph

Ase instances and network interfaces are connected together to form the configuration graph.This is done in the `config` block as shown in Figure 1.2(a). This can be accomplished using multiple statements. Each statement ends with a semi-colon. Each network interfaces is identified by the device name allocated to it by the system. In Linux this tends to have a prefix `eth` followed by one or more numeric characters. These must be specified as arguments to `config`. Once specified as arguments to `config`, these interfaces can be used in the configuration like any other Ase with the limitation that they can have only one input and one output. Both unidirectional and bidirectional links between Ases can be used to specify the graph. These are represented by arrows: `->`, `<-` and `<->`.

1.6 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 few protocol implementations with diverse compositions. The framework can be used to implement forwarding in DNS [24, 25], Mobile IP [27], Dynamic Source Routing [16] and other multicast and anycast protocols with little effort. We discuss the implementation details of an IP forwarder, NAT (as an example of virtual-circuit setup) and forwarding in an i3 server (as an example of an overlay routing mechanism). These examples also give some intuition behind code reuse and the amount of code that FLECS programmer is spared from writing.

1.6.1 IP Forwarding

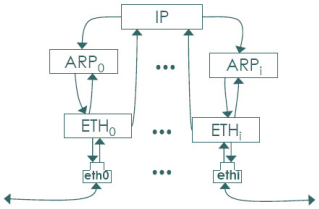

Fig. 1.7. IP Router in FLECS.

A simple IP forwarder can be modeled in FLECS as shown in Figure 1.7. A specific configuration is shown in Figure 1.8. 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 the respective `ETH` ASE, the packet is dropped, otherwise the Ethernet header is popped off and the packet forwarded to `IP` ASE (Figure 1.8, lines 8-10).

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. Figure 1.9 shows the IP ASE specifications in ACL. Constatnt definitions and pre-forwarding hooks are not shown 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

```
Eth eth_$i {
 control {
  [eth$i, mac]         ->  [forward/pop];
  [eth$i, FFFFFFFFFFFF]->  [forward/pop];
  [arp_$i, *]          ->  [recast][forward/none];
 }
 switching {
  [eth$i, mac#0806]         ->  [arp_$i, null];
  [eth$i, FFFFFFFFFFFF#0806]->  [arp_$i, null];
  [eth$i, mac#0800]         ->  [ipswitch, null];
  [arp_$i, *]               ->  [eth$i, null];
 }
}

Arp arp_$i {
 control {
  [eth_$i, ip#0001]-> [rupdate][respond];
  [eth_$i, ip#0002]-> [rupdate];
  [ipswitch, *]    -> [resolve][forward/pop+push];
 }
}

Ip ipswitch {
 control {
  [*, *]       -> [forward/push];
 }
 switching {
  [*, C0A80303] -> [arp_0, C0A80303];
  [*, C0A80707] -> [arp_1, C0A80707];
 }
}

config(eth0, eth1) {
 eth0 <-> eth_0(mac=00055DE6265D)
   <-> arp_0(ip=C0A80305, mac=00055DE6265D);
 eth1 <-> eth_1(mac=00055DE6265E)
   <-> arp_1(ip=C0A80705, mac=00055DE6265E);
 eth_0 -> ipswitch <- eth_1;
 arp_0 <- ipswitch -> arp_1;
}
```

Fig. 1.8. Sample IP Router Configuration in FCL.

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.

1.6.2 Network Address Translation

Figure 1.10 shows the NAT model in FLECS for a NAT box having two internal and two external interfaces. In addition to forwarding packets, the `NAT` ASE also performs path setup for outgoing packets and the `NAT` forwarding entries act as a filter for all incoming packets. The following snippet shows the pattern specifications in the `NAT` ASE. The `setup` specification illustrates the virtual circuit setup for packets coming from the internal subnet.

```
1  DEFINE  ...
2
3  ASE Ip {
4   peek {
5    READ { dest_anno DEST_ANNO_OFFSET IP_ADDRESS_LEN
6     proto_anno PROTO_ANNO_OFFSET IP_PROTO_LENGTH
7     dest_ip      DEST_IP_OFFSET  IP_ADDRESS_LEN
8     src_ip       SRC_IP_OFFSET   IP_ADDRESS_LEN
9     protocol IP_PROTO_OFFSET IP_PROTO_LENGTH
10   }
11   CONTROL { protocol }
12  }
13
14  recast {
15   CAST { IP_ANNOTATION_LEN IP_HEADER_LEN
16    WRITE{VER_IHL VER_IHL_OFFSET VER_IHL_LENGTH
17         TLEN    TLEN_OFFSET     TLEN_LENGTH
18         TTL     TTL_OFFSET TTL_LENGTH
19         proto_anno IP_PROTO_OFFSET IP_PROTO_LENGTH
20         local.ip SRC_IP_OFFSET IP_ADDRESS_LEN
21         dest_anno DEST_IP_OFFSET IP_ADDRESS_LEN }
22    }
23    %localRecalculateChecksum(p);
24  }
25
26  forward {
27    LOOKUP       { dest_ip }
28    POP  { IP_HEADER_LEN }
29    %localTTLDecrement(p);
30  }
31
32 void local...(Packet*p) {
33  % ...
34 }
35 };
```

Fig. 1.9. IP ASE in FCL.

The `setup` pattern first looks up the switching table to see if the entry for the source IP and port exists. If not, then it executes the virtual circuit (`VC`) block. VC acquires a local name and creates the virtual circuit entries in the switching table. It rewrites the source IP and port in the `SWAP` block before calling `localRecalculateChecksums` on the packet.

The other ASEs, which are common in both NAT and IP router configurations perform forwarding operations as described for the IP router, with the exception of a few minor changes to route the packets through `NAT` ASEs.

1.6.3 i3 Forwarding

Using the FLECS framework, i3 [28] becomes a straightforward implementation of forwarding using Chord [29] as the routing process. This can be modeled by extending the IP router design by adding a `I3Switch` and `I3TriggerHandler` ASEs as shown in Figure 1.11. In our model the i3 overlay sits on top of UDP. When a UDP packet arrives at `IP` for the i3 server it is relayed to `I3Switch` after the IP and UDP headers are removed in the respective ASEs. `I3Switch`, which implements the Chord protocol, determines whether the top-

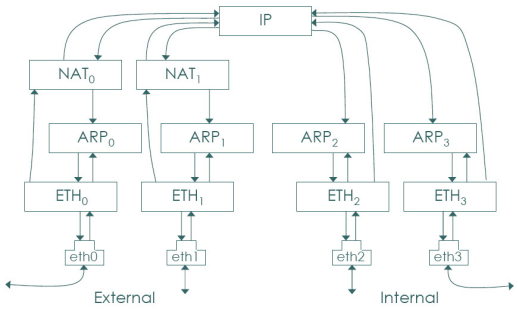

Fig. 1.10. NAT Configuration in FLECS.

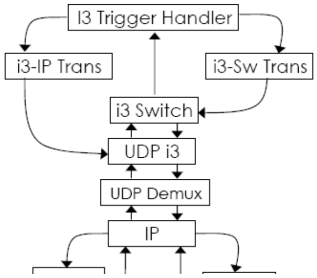

Fig. 1.11. I3 Forwarding in FLECS.

most i3 id can be locally resolved or not through switching table lookup. If so, then the message is forwarded to the `I3TriggerHandler`, otherwise it is forwarded to the Chord neighbor as determined by the switching table. `I3TriggerHandler` lookup on the topmost i3 id in the i3 id stack determines the number of packet copies made and forwarded, each one with either a new i3 id added on top of the id stack or a specific IP/UDP destination. The packet or packets are then forwarded to the respective translators (`I3IP-Trans` or `I3SW-Trans`) for proper recasting depending upon the switching table lookup in `I3TriggerHandler`. If `I3TriggerHandler` does not have an entry for the topmost i3 id and the i3 id stack in the packet is empty, the packet is dropped.

1.7 Evaluation

Figure 1.12 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

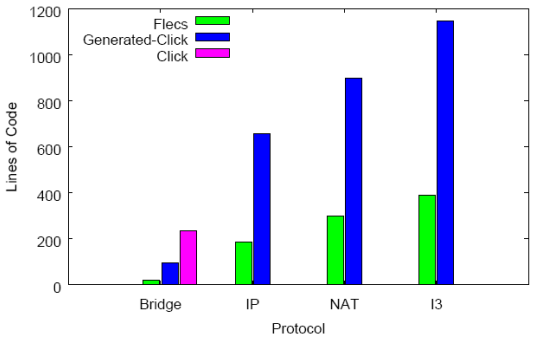

Fig. 1.12. Comparison of the number of lines of code in the .ase file with the number of lines generated by the *asec* compiler. Click implementation of Etheret bridge is in 236 lines of code.

interfaces in less than thirty lines (1.2). 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 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 1.13 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

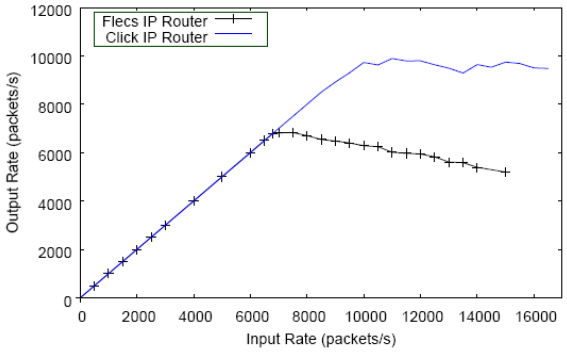

Fig. 1.13. Forwarding Rates of an IP Router in FLECS.

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.

1.8 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.

It should be noted that FLECS has been developed as a proof of concept for the axiomatic basis for communication [18] and is limited by the same set of limitations as the model, such as obliviousness to time, error and loss. These limitations, restrict us from implementing protocol mechanisms such as handling congestion and retransmissions. we intend to extend our framework to address these limitations in future work.

Our conformance to the axiomatic basis not only allows us to discover different patterns in packet forwarders but also makes the design of our framework independent of any specific software or hardware architecture. It would be reasonable to state that FLECS can be implemented on other packet processing engines and network processors [7, 6]. Future implementations of FLECS may perhaps be able to generate validated protocol implementations for programmable hardware devices such as FPGA [3, 13]. This would demonstrate the potential of automatically building validated protocol implementations.

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. H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A Layered Naming Architecture for the Internet. In *SIGCOMM '04*, pages 343–352, Portland, OR, USA, aug 2004. ACM Press.
2. T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(14):25–59, 1987.
3. S. Brown and J. Rose. Architecture of fpgas and cplds: A tutorial, 1996.
4. S. Budkowski and P. Dembinski. An Introduction to Estelle: A Specification Language for Distributed Systems. *Computer Networks and ISDN Systems*, 14(1):3–24, 1988.
5. D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *SIGCOMM '88*, pages 106–114, August 1988.
6. D. Comer. *Network Systems Design with Network Processors*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.
7. D. Comer and L. Peterson. *Network Systems Design Using Network Processors*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.
8. T. Condie, J. M. Hellerstein, P. Maniatis, S. Rhea, and T. Roscoe. Finally, a Use for Componentized Transport Protocols. In *HotNets IV*, 2005.
9. W. Dabbous, S. O'Malley, and C. Castelluccia. Generating Efficient Protocol Code from an Abstract Specification. In *SIGCOMM '96*, pages 60–72, New York, NY, USA, 1996. ACM Press.
10. T. G. Griffin and J. L. Sobrinho. Metarouting. In *Proceedings of SIGCOMM '05, Philadelphia, PA, USA*, pages 1–12, August 2005.

11. M. Gritter and D. R. Cheriton. An Architecture for Content Routing Support in the Internet. In *USITS 2001*, pages 37–48, San Fransisco, CA, USA, March 2001.
12. C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969.
13. M. Hutton. Architecture and cad for fpgas. In *SBCCI '04: Proceedings of the 17th symposium on Integrated circuits and system design*, pages 3–3, New York, NY, USA, 2004. ACM Press.
14. K. Ishikawa and T. Hoshino. Rapid Protocol Prototyping from Message Sequence Chart Based Specification. In *Seventh IEEE International Workshop on Rapid System Prototyping*, pages 61–64, jun 1996.
15. A. Jirachiefpattana and R. Lai. A Rapid Protocol Prototyping Development System. In *Sixth IEEE International Workshop on Rapid System Prototyping*, pages 118–126, jun 1995.
16. D. B. Johnson and D. A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. *Mobile Computing*, 353:153–181, 1996.
17. M. Karsten, S. Keshav, and S. Prasad. An Axiomatic Basis for Communication. In *HotNets V*, pages 19–24, Irvine, CA, USA, nov 2006. ACM Press.
18. M. Karsten, S. Keshav, S. Prasad, and M. Beg. An Axiomatic Basis for Communication. In *SIGCOMM '07*, Kyoto, Japan, aug 2007. ACM Press.
19. E. Kohler, M. F. Kaashoek, and D. R. Montgomery. A Readable TCP in the Prolac Protocol Language. In *SIGCOMM '99*, pages 3–13, August 1999.
20. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.
21. B. Krupczak, K. Calvert, and M. Ammar. Increasing the Portability and Reusability of Protocol Code. *IEEE/ACM Transactions on Networking*, 5(4):445–459, August 1997.
22. X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. P. Birman, and R. L. Constable. Building reliable, high-performance communication systems from components. In *SOSP '99*, pages 80–92, December 1999.
23. A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: Creating a "Functional" Internet. In *EuroSys '07: Proceedings of the 2007 conference on EuroSys*, pages 101–114, New York, NY, USA, March 2007. ACM Press.
24. P. Mockapetris. RFC 1034 - Domain Names - Concepts and Facilities, November 1987.
25. P. Mockapetris. RFC 1035 - Domain Names - Implementation and Specification, November 1987.
26. R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. In *SOSP '99*, pages 217–231, Kiawah Island Resort, near Charleston, SC, USA, December 1999.
27. C. Perkins. RFC 3344 - IP Mobility Support for IPv4, August 2002.
28. I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet Indirection Infrastructure. *IEEE/ACM Transactions on Networking*, 12(2):205–218, April 2004.
29. I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. *IEEE Transactions on Networking*, 11:17–32, February 2003.
30. D. L. Tennenhouse. Layered Multiplexing Considered Harmful. In *First International Workshop on High Speed Networking*, nov 1989.