FLECS: A Data-Driven Framework for Rapid Protocol Prototyping

by

Mirza Omer Beg

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2007

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Mirza Omer Beg

Abstract

FLECS is a framework for facilitating rapid implementation of communication protocols. Forwarding functionality of protocols can be modeled as a combination of packet processing components called *abstract switching elements* or ASES. The design of ASES 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. We have developed FLECS on top of the Click modular router. The compilers in the FLECS framework translate protocol specifications into its Click implementation. We claim that the use of our framework reduces the implementation time by allowing the programmer to specify ASES and the forwarding configuration in a high-level meta-language and produces reasonably efficient implementations. It allows rapid prototyping through configuration, as well as specialized implementation of performance-critical functionality through inheritance.

Acknowledgements

I would like to thank my supervisors Martin Karsten and Keshav for their tremendous support and patience.

Dedication

Dedicated to loved ones.

Contents

List of Tables

List of Figures

Chapter 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. Components can be reused in different implementations and can be readily adapted and tuned for new environments [12]. An added advantage of a modular design is that components can be specified and verified more easily compared to a monolithic implementation. Furthermore, the dynamic nature of today's communication networks require individual components to evolve over time [13]. We intend to use concepts from modularization to reduce the complexity of network communication infrastructure and we hope to use formal analysis to understand and prove correctness even for complex protocols.

The benefits of modularization come at a cost. Modularization may lead to high performance overhead due to additional boundary crossings. There may also be an additional cost of configuring the system in terms of selection and parametrization of components.

This thesis describes the design and implementation of FLECS, a framework that employs modularization to quickly implement forwarding functionality of communication protocols. Most of the existing research in protocol prototyping is generally directed towards optimization and performance enhancement techniques [6, 26]. Most current systems lack a solid theoretical foundation, which makes it almost impossible to formally analyze their behavior with respect to communication. Notable exceptions include [7, 14], which study the underlying principles of connectivity in communication protocols. In contrast, our work builds on an axiomatic basis for expressing com-

munication primitives that provides a theoretically sound framework for expressing fundamental internetworking 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.

1.1 The Theoretical Framework

A parallel stream of research has made an attempt to define communication invariants using axioms [22, 23]. This work was inspired by Hoare's axiomatic basis for programming [16] and is closely related to other work in the area of naming and addressing indirection [2, 15, 32].

The axiomatic framework defines components called *abstract switching elements* or ASEs. This facilitates the overall protocol design by dividing it into subtasks 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 actions. These constraints on ASE design are enumerated as follows.

- Two ASEs can directly communicate by exchanging messages if and only if they are neighbors.
- ASEs choose which neighbor to forward a packet to based only on the packet header and local switching information.
- Packet forwarding from a source to a destination is modeled as a transitive closure of direct communication and local switching.

These constraints do not restrict the power of an ASE significantly, yet they are sufficient to allow formal verification. In effect, the more the user is constrained, the more the framework knows about what the user can possibly do. Stated another way, FLECS has more domain knowledge than an approach in which packet forwarders are specified in a general-purpose language. This knowledge can possibly, in the future, be harnessed to apply internal optimizations as well as automated verification of forwarder implementations.

1.2 A Simple Configuration

We describe the concepts behind the design of FLECS by showing a simple example. Figure 1.1 illustrates the configuration for an Ethernet broadcast forwarder with three interfaces called *eth*0, *eth*1 and *eth*2. It shows the complete ASE and configuration code for broadcast. In this configuration, the ASE forwards an arriving packet to all the interfaces except the one on which it arrived. Figure 1.1(b) gives the code for the Broadcast ASE, where lines 8-10 define the forward pattern. The lookup specified in the pattern indicates which field is used for switching lookup. In this case it is the `dest` field, which will always match the first line in the switching table (Figure 1.1(c), line 6). This switching entry is defined with wildcard ($*$), which matches any value to forward the packet to all the interfaces except the one on which it arrived. To indicate all the interfaces other than the source $-\$i$ is used to replace the number in the interface name. This feature assumes that the network interface prefixes are the same. Figure 1.1(c) illustrates the configuration file for broadcast. Lines 1-8 instantiate a Broadcast ASE with the respective control and switching table. The single entry in the control table (Figure 1.1(c), line 3) results in the execution of the `forward` pattern on all arriving packets. Finally, lines 9-12 give the textual representation of the configuration graph shown in Figure 1.1(a).

1.3 Objectives

The main contribution of this thesis is FLECS, a configurable, flexible universal forwarding engine for implementing packet forwarders. The FLECS framework allows a programmer to specify a protocol in a high-level meta-language. The language inherently restricts the design choices available to the programmer while making the specification simpler. The constraints imposed by our language are derived directly from our communication axioms [23]. This special-purpose programming language also provides an explicit model for concisely expressing protocols in accordance to the design principles stated in the axiomatic basis.

This work also presents encouraging results from our experience with implementing the universal forwarding engine. The project was undertaken with the following goals in mind.

- Implement fundamental packet processing operations that can be composed to form packet forwarding schemes.

(a)

```
1   DEFINE  DEST_OFFSET 0
2   DEFINE  DEST_LENGTH 6
3   ASE Broadcast {
4           peek {
5                   READ { dest DEST_OFFSET DEST_LENGTH }
6                   CONTROL { dest }
7           }
8           forward {
9                   LOOKUP { dest }
10          }
11  };
```

(b)

```
1   Broadcast bc {
2           control   {
3                   [*, *] -> [forward/none];
4           }
5           switching {
6                   [eth$i, *] -> [eth-$i, null];
7           }
8   }
9   config(eth0, eth1, eth2) {
10          eth0 <-> bc <-> eth1;
11          bc <-> eth2;
12  }
```

(c)

Figure 1.1: (a)Ethernet broadcast configuration for a three interface forwarder. Network interfaces are represented as special Ases with a single input and a single output. (b) Definition of Broadcast Ase in Flecs (broadcast.ase). (c) Broadcast configuration represented in Flecs (broadcast.flecs).

- Use object-oriented programming methodology to enforce constraints and allow re-usability of code.
- Present meta-language specifications for packet forwarders and demonstrate their feasibility by implementing some non-trivial forwarding schemes.
- Implement tools to automatically generate runnable forwarder implementations from the specifications written in our meta-language.
- Provide a framework where formal reasoning can be applied for automated protocol verification in the future.

The contributions of this thesis are threefold. First, it describes the design of FLECS, including its programming model. Second, it discusses the FLECS implementation using the Click modular router [24]. Third, it demonstrates the feasibility of a universal forwarding engine by building working prototypes that inter-operate with existing protocol suites.

1.4 Thesis Overview

The rest of the thesis is organized as follows. Chapter 2 gives an overview of related work followed by a brief restatement of the axiomatic formulation in Chapter 3. Chapter 4 examines the FLECS framework and its core components. Chapter 5 describes the implementation of FLECS in Click. This chapter also details our meta-language constructs with some simple examples. Chapter 6 illustrates the practical capabilities of our framework by compactly describing some non-trivial forwarding schemes such as an Ethernet bridge, IP router, NAT and i3 server. Chapter 7 evaluates the effectiveness of our approach and we end with conclusions and future work in Chapter 8.

Chapter 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 [22, 23] succinctly formalizes the design principles behind communication protocols and provides a basis for formal reasoning about their properties. This formulation forms the foundation for FLECS. We briefly describe the axioms in the next section. FLECS attempts to implement the constraints defined by the axioms, using Click [24, 29]. The design of FLECS reflects our understanding of protocol software based on experience and insight to support rapid implementation of protocols.

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. These elements are written in C++. The edges in the configuration represent possible packet paths. Click can be used to implement any forwarding engine, with few constraints on its design. It does not allow elements to share data-structures but a limited amount of data can be passed from one element to the other along with the packet. Modular design makes it easier to extend Click configurations. We differ from this approach in that we specify protocols at a higher level of abstraction rather than in a general-purpose programming language. The ASES are fairly complex compared to Click elements. Other aspects of Click architecture include push and pull connections, where packet handoff is initiated by the source end or the destination end. Pull processing makes it possible to write composable packet schedulers. FLECS does not allow the programmer to write such schedulers by restricting the ASES to use push processing and making the queuing elements implicit in the design, for simplification. In addition, our design constrains

the programmer according to the axiomatic formulation of packet forwarding [23]. We find Click to be complementary to our work and indeed we use it to build the first prototype of our system.

The *x*-Kernel [17, 30] is an early contribution towards a system for composing network protocols from user-level protocol objects. These objects can be specified in a high level programming language [1] that allows the programmer to program in an imperative style. Like a Click router, an *x*-Kernel configuration is a graph of processing nodes, and packets are passed between nodes by virtual function calls. However, it imposes constraints on the configuration that result in an acyclic and layered graph. This restricts *x*-Kernel to stack-oriented configurations and makes it difficult to construct simple cyclic configurations like an IP router. The inter-node communication protocols are fairly complex and connections between graph nodes are bi-directional. Packets travel up the graph to the user level and down the graph to the network. This is because the framework is intended for use at end nodes, where packet motion is vertical (between network and user level) rather than horizontal (between network interfaces). In contrast, our framework can handle vertical as well as horizontal packet motion.

Estelle (Extended State Transition Language) [5] 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. It borrows features from Pascal programming language and some Ada modular constructs. 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. The actual behavior of a module is is specified either as an integrated behavior of a set of interacting sub-modules or at the innermost level, as an extended finite state machine. The channels are type defined in modules according to their roles, i.e. user and provider. Operations in a module are classified as transitions, based on an input or spontaneous event. A spontaneous transition may be executed regardless of any input interaction. It should be mentioned that the Estelle state machine is nondeterministic. 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.

Approaches like SDL [34], LOTOS [3] and Esterel [6, 11], 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, subject only to the constraints imposed by the operating system and architectural environment in which the protocol will be used. Although forwarders implemented in this way are generally efficient, the programming task varies in difficulty, depending on the extent the framework is designed to accommodate new protocols. Others have augmented these techniques to design protocol prototyping systems with message sequence charts [19] and automated verification tools [20].

FLECS represents a middle ground approach compared to previous approaches to protocol design. It allows the user to define communication protocols at a higher level of abstraction using configurable protocol objects; yet it retains the clarity and simplicity in design that enables us to prove some essential properties of protocols.

Chapter 3

Axioms of Communication

In this section we briefly discuss the axiomatic framework [23] that forms the basis of this work. It formulates fundamental forwarding mechanisms in communication networks. This formulation allows us to express precisely and abstractly the concepts of naming and addressing and to specify a consistent set of control patterns and operational primitives, from which a variety of communication services can be composed.

3.1 The Axioms

The axiomatic formulation of forwarding principles describe 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 switching 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.

3.2 Forwarding and Control Primitives

Theoretically, the transformation from p to p' in LT2 in Section 3.1 is unrestricted, but in practice it is either a *push*, *pop*, *swap*, or *nop* operation. In case of *push*, a new prefix q is prepended and $p' = qp$. In case of *pop*, p is removed and $p' = \varnothing$. In case of *swap*, p is replaced by p', which usually is of equal length. With *nop*, p remains unchanged and $p' = p$. Given these transformations, it is possible to identify corresponding forwarding operations that cover a wide range of forwarding techniques used in communication protocols.

These processing operations can be translated into a set of forwarding primitives that can be used to abstractly specify ASE processing in pseudo-code. Each ASE supports the same set of operational primitives. Although all ASEs offer the same interfaces and roughly the same semantics, they differ in their specific implementations. In addition to these transformations LT1 and LT2 describe communication between ASEs and lookup of the switching table. These can be represented as the forwarding primitives which can be further used to specify the abstract functionality of a network element. These include *send*, *receive*, *copy*, *push*, *pop* and *lookup* and will be discussed in more detail later.

The operations mentioned thus far are sufficient for forwarding in a static network. However, to perform forwarding or distributed resolution each ASE's local switching

table needs to be populated. To handle the dynamic nature of the network, control primitives are introduced to update an Ase's switching table and/or create and send a reply message carrying information from the local switching table. These include *update* and *create*.

3.3 Packet Processing

It turns out that the fundamental elements for message processing can be expressed as a small number of processing *patterns*. The overall processing of an Ase can be logically partitioned into the following processing patterns. These patterns are invoked by a tight processing loop in the Ase. In an implementation this processing loop may be implemented in a 'Super-Ase' from which the rest of the Ases inherit the base functionality.

- `forward`
  This pattern is used for regular forwarding of data messages.
- `setup`
  This pattern is used for forwarding path setup requests. An Ase can be configured to implement virtual circuit switching or bridging through this pattern.
- `resolve`
  The resolve pattern carries out a resolution request.
- `respond`
  The respond pattern is used when a remote lookup request arrives. It creates a response if requested name is found in the local switching table and sends it.
- `rupdate`
  The rupdate pattern is invoked on receipt of a response to s resolution request.

It would be worth noting that these patterns are sufficient to model virtual circuit switching and packet forwarding, as well as path setup and name resolution. In addition, they can be composed from primitives outlined in Section 3.2.

3.4 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 only 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: Constraints of this type arise from the general Ase design and the patterns defined in Section 3.3.

  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.

Chapter 4

Framework

There are two main design goals behind the FLECS framework. First, our protocol specification language should comply with the axiomatic fundamentals [23], which constrain packet processing in ASES. Second, FLECS should allow programmers to specify complete protocol functionality in a simple and concise manner. The first goal emerges from experiences with traditional routers, firewalls, VLAN switches, NAT boxes and other middleboxes. As is often the case in writing software, one programs a protocol by using a similar protocol as a template, and then editing it to obtain the desired functionality. This approach derives its benefits from the fact that there are routine tasks, such as manipulating headers and demultiplexing, that are common to many protocols. This common functionality can be extracted as a super component and can be reused for different implementations instead of being re-written from scratch [10, 25]. This enables the programmer to automate the task of protocol implementation from a minimum set of specifications.

Note that restricting the programmer to a limited specification language constrains the design choices for the protocol. An obvious benefit is that the programmer does not have to write the entire protocol code. A less obvious benefit is that the programmer is restricted from making bad design choices.

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 can implement new ASES by refining BASE to produce ASES required to construct a specific protocol. Figure 4.1 illustrates the general design of the FLECS framework. It depicts the inheritance of ASES

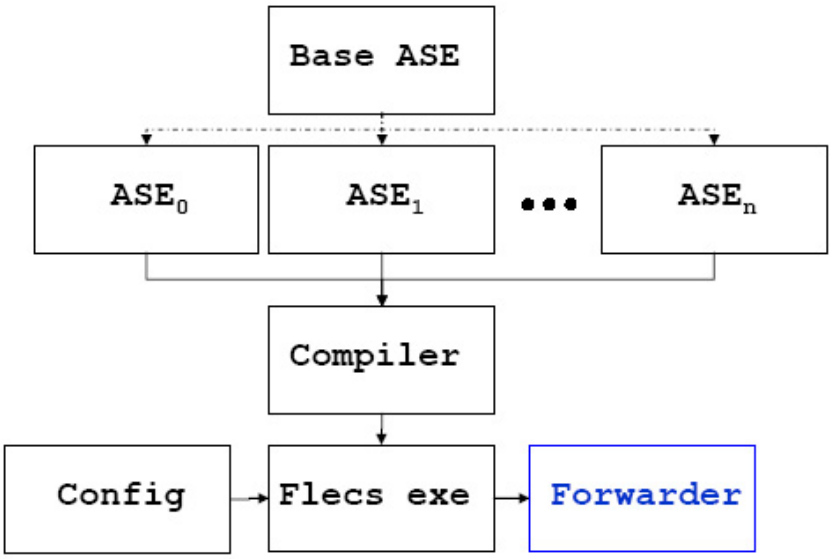

Figure 4.1: The Design of the FLECS Framework.

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. To accommodate such functionality the framework allows for inclusion of arbitrary functions in the ASEs as discussed in the Chapter 5.

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.

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 4.1. 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 [32], 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.

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 4.2. 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

Table 4.1: 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.

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 afore-mentioned 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 that can be used to define ASEs are enumerated in Table 4.2. At present, the listed patterns are sufficient to implement all of the well-known packet forwarders.

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 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 Broadcast, Figure 1.1(c)). 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 4.2 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'

Table 4.2: 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, String prev) {
2           {s} = peek(p)
3           patterns[] = ctrlLookup(prev, {s})
4
5           for (each pattern in patterns[]) {
6                   if (p) execute(pattern, p)
7           }
8   }
```

Figure 4.2: The Main Processing Routine of an ASE.

is a set of strings; the pair forms the *key* for that entry. The key maps onto a set of patterns. 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.

4.5 Base ASE

BASE models a generic ASE by implementing the forwarding primitives and declaring the processing patterns as virtual functions as shown in the BASE interface, Figure 4.3. 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  interface BASE {
2    // The two tables which contain the local state.
3    Table  control;
4    Table  switching;
5
6    // Base class methods which cannot be overridden in
7    //  sub classes. These consist of most of the main
8    //  processing method and most forwarding primitives.
9    void     process(Pkt p, Str prev);
10   void     send(Pkt p, Str next);
11   void     receive(Pkt p, Str prev);
12   Pkt      push(Pkt p, int pushlen, Triple[] d);
13   Pkt      swap(Pkt p, Triple[] d);
14   Pkt      pop(Pkt p, int length);
15   Pkt      create(int createlen, Triple[] d);
16   void     drop(Pkt p);
17   Pattern[]ctrlLookup(Str prev, Str n);
18   {Pair[]} fwdLookup(Str prev, Str n);
19   void     fwdUpdate(Str prev, Str n, Pair[] v);
20
21   // Processing operations that
22   //  can be overridden in sub classes.
23   virtual Str[] peek(Pkt p);
24   virtual Pkt recast(Pkt p, Str prev);
25   virtual Pkt forward(Pkt p, Str sub, Str prev);
26   virtual Pkt setup(Pkt p, Str sub, Str prev);
27   virtual Pkt resolve(Pkt p, Str prev);
28   virtual Pkt respond(Pkt p, Str prev);
29   virtual Pkt rupdate(Pkt p, Str prev);
30 };
```

Figure 4.3: The Interface for BASE ASE.

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.

4.7 Message Flow

Message flow between Ases is constrained by the axioms as outlined in Chapter 3. Messages flow between Ases which are directly connected to each other. Our framework assumes that messages do not get modified during direct communication, and hence our model does not handle any corruption or loss of messages. Thus we cannot model many interesting phenomena driven by packet loss or corruption.

Flecs also assumes that messages that arrive at an Ase do always find room in the local buffer and the control table contains the respective entry to process the packet. This also means that we assume the completeness of the control table in the sense that it has entries to handle all the possible packets that can arrive at the Ase.

Chapter 5

Implementation

We have implemented FLECS using Click [24], a framework for building flexible, configurable routers. A Click router configuration is specified by a directed graph of *click elements* where each element performs a specific and simple router function such as queuing, scheduling or updating a field of a packet. Click elements are implemented in C++ and form the core of the Click runtime system. They are configured at runtime by interpreting a configuration file, which specifies how and which elements are connected to each other.

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 5.1 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. However, unlike most traditional Click elements, an ASE represents a more complex unit of processing. Given the ASE design, it can easily be noticed that a traditional protocol layer can be modeled 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 5.1.

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.

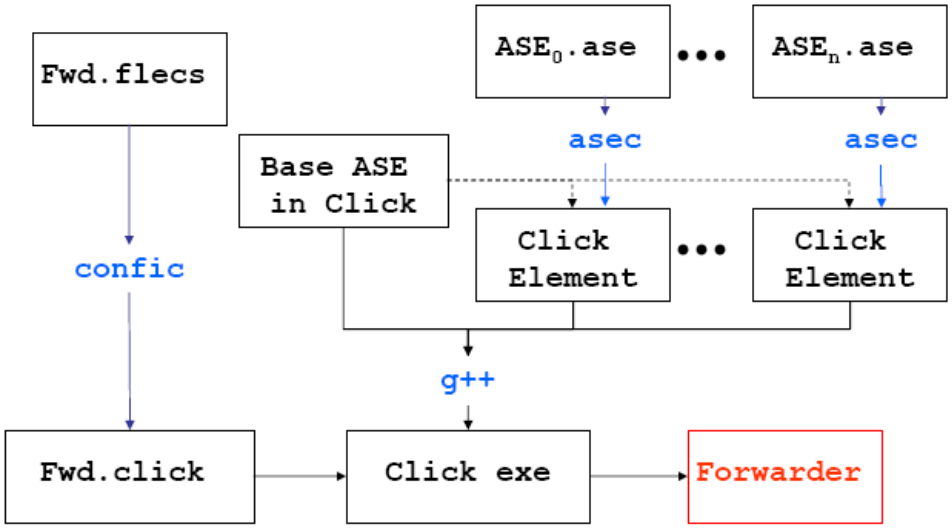

Figure 5.1: FLECS Implementation in Click.

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 (if needed), pattern definitions, and references to relevant fields in the packet header (see Figure 1.1(c)). The constituents of the ASE description are defined in more detail below.

5.1.1 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. The main objective of this language construct is to improve readability of code. The broadcast example declares the `OFFSET` and `LENGTH`. 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 DEFINE. These values are specified as hexadecimal strings using double quotes. Following are examples from the Ethernet ASE.

```
DEFINE ETHERNET_HEADER_LEN  14
DEFINE SOURCE_MAC_OFFSET    6
DEFINE BROADCAST_ADDRESS    "FFFFFFFFFFFF"
```

The first defines the length of the Ethernet header which is used to pop the Ethernet header from the packet. BROADCAST_ADDRESS is used as data to be written in the destination field of the Ethernet header if the packet is to be broadcasted.

5.1.2 Extracting Header Fields

ASEs define the peek method to extract relevant data from the packet arriving at the ASE. Peek defines the set of packet fields to be used in the processing patterns. 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 three values has the following syntax:

```
READ {
  var0 OFFSET_0 LENGTH_0
  var1 OFFSET_1 LENGTH_1
  var2 OFFSET_2 LENGTH_2
}
```

5.1.3 Specifying Control Values

Each ASE must specify the header fields to be used for control table lookup. This is specified 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 is an example of peek where the relevant offsets and lengths are assumed to have been declared using DEFINE. In this example, the control table lookup is based on address and port fields of the arriving packet.

```
peek {
  READ {
   address ADDRESS_OFFSET ADDRESS_LENGTH
   port    PORT_OFFSET PORT_LENGTH
  }
  CONTROL { address port }
}
```

5.1.4 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 in the format $\{[next\_Ase, \{s\}]\}$ and sends a copy of the packet to the respective destination determined by *next_Ase* and a set of strings {s}. Each string in {s} is given in a hexadecimal representation. This pattern also has advanced subtypes, `push`, `pop` or `swap`. `Push` 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. Swap replaces a given set of header fields with the given strings. `Pop` and `swap` subtypes can be specified after lookup.

```
forward {
  LOOKUP { address port }
  POP { POP_LENGTH }
  SWAP { WRITE { DATA_w0 OFFSET_w0 LENGTH_w0 } }
}
```

The example shows a `forward` specification for a hypothetical ASE. It translates into a pattern, when executed on a packet with subtype `pop`, looks up the `address` and `port` combination, removes `POP_LENGTH` bytes from the beginning of the packet and forwards it to the *next_Ase*. Note that the push subtype is not specified in the forward specification as it simply prepends the strings from the lookup result to the packet.

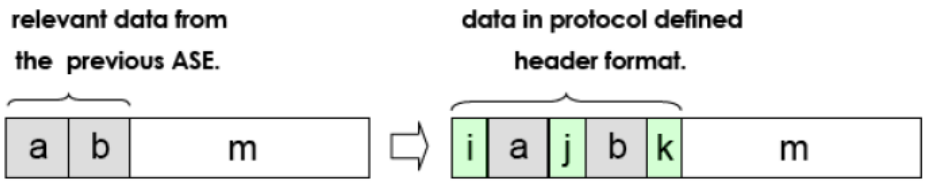

Figure 5.2: An abstract representation of recasting data from a previous Ase into the protocol defined header format.

5.1.5 Encapsulation

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. For example an Ethernet Ase would pop off the Ethernet header before forwarding an ARP packet to the ARP Ase. 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 5.2. It can be generalized into the following format, where ANNOTAION_LENGTH is the length of data from the previous Ase and CAST_LENGTH is the length of the properly formed header. The following example shows the recast specification which eventually writes two fields in the reconstructed packet.

```
recast {
  CAST { ANNOTAION_LENGTH CAST_LENGTH
       WRITE { data_0 OFFSET_0 LENGTH_0
               data_1 OFFSET_1 LENGTH_1 }
  }
}
```

In Figure 5.2 the ANNOTAION_LENGTH is the combined length of a and b whereas CAST_LENGTH is the combined length for i, a, j, b and k. The WRITE block in recast is responsible for populating these fields with the correct values.

5.1.6 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 { val }
  VC {
   LOC_NAME { local_name() }
   UPDATE { * LOC_NAME prev val }
  }
  SWAP { WRITE { data_0 OFFSET_0 LENGTH_0 } }
}
```

5.1.7 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` pattern is 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.

```
respond {
  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 }
  }
}
```

5.1.8 Post-Forwarding Hooks

There are also certain operations which are specific to forwarders and have not yet been modeled in our framework. 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. This piece of code is passed directly into the output file without any syntactic or semantic interpretation. 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);
}
```

Note that the declaration line of the method and the last line are not preceded by the % character. The method name should have a prefix '`local`'. Also, this code should be a valid and complete function as the compiler will not check the inlined code for consistency, but the errors will be reported when the Click elements are compiled by the C++ compiler.

This feature is used to make modifications to the packets which do not affect the switching decisions, but are required for correctness of the overall forwarding protocol. These include error detection and error correction functions such as checksum computations etc. This feature cannot be used to extend the core functionality of an ASE as the main process loop cannot be overwritten to directly call these user-defined functions. These functions can only be called from within `recast`, `forward` and `setup`.

5.1.9 Meta-Compilation of ASEs

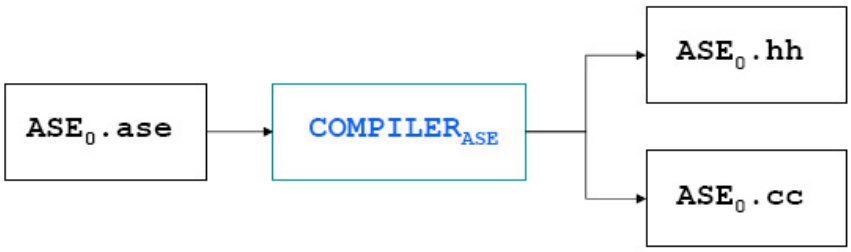

Figure 5.3: Generating Click Implementation of an ASE.

Conventionally ASEs are specified in `.ase` files. For example the Broadcast ASE is defined in `broadcast.ase`. It can be compiled using the following command:

```
asec [-l] broadcast.ase
```

This produces Click element code for user-level Click for Broadcast. For generating code for kernel-module Click, the same command can be used with `-l` option. The `asec` compiler generates two files. They are named `broadcast.hh` and `broadcast.cc` for the given example. The `asec` compiler transformation is shown in Figure 5.3. These files must be copied over to the root Click folder where they can be compiled with the rest of the Click elements.

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.

5.2.1 ASE Initialization

The type of an ASE is the name with which it is defined in the `.ase` file. An instance can be named using an unused string identifier. In general, initialization involves creating an instance of an ASE defined in a `.ase` file, 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 broadcast (Figure 1.1(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`.

5.2.2 Control Table

The theoretical model described in the axiomatic basis [23] 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 4.2. In addition the pseudo-patterns are listed in Table 5.1. These operations are elevated to a special status of pseudo-patterns because they can be used in the control table just like our regular patterns.

Table 5.1: Additional Pseudo Processing Patterns

Pattern	Description
recast	Encapsulates the packet in proper header format by removing a specified prefix from the packet header and reconstructing the header into a specified format.
drop	Discards the packet.

5.2.3 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];
```

5.2.4 Configuration Graph

The configuration graph can be created using ASE instances, network interfaces and connectors. This is done in the `config` block as shown in Figure 1.1(a). The graph can be specified in multiple statements. Each statement ends with a semi-colon.

5.2.5 Implicit Queuing

The specification language does not allow explicit queuing. Queuing is implicit in the framework. The generated configuration adds the queues at the end of each outgoing path towards the network interfaces. This restricts the programmer from defining composable packet schedulers, which require that queues be specified explicitly.

5.2.6 Network Interfaces

FLECS identifies network interfaces 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`. Figure 1.1 shows how a forwarder handling three network interfaces can be specified. 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.

5.2.7 Connectors

FLECS allows both unidirectional and bidirectional links between ASES. These are represented by arrows: `->`, `<-` and `<->`. The Ethernet broadcast configuration example shows a typical use of the double arrow in Figure 1.1.

5.2.8 Meta-Compilation of FLECS Configuration

Figure 5.4: Generating Click Configuration.

The FLECS configuration can be translated into Click configuration by using the command

```
confic broadcast.flecs
```

This generates `broadcast.click` which can be run on the click interpreter to produce the desired forwarding functionality. The transformation of the configuration compiler is shown in Figure 5.4.

Chapter 6

Examples

In this section, we discuss how the FLECS framework can be used to implement some well-known and non-trivial communication protocols. In the following sections, we discuss a few protocol implementations with diverse compositions. The framework can be used to implement DNS [27, 28], Mobile IP [31], Dynamic Source Routing [21] and others with minimum effort. We discuss the implementation details of a learning Ethernet bridge, an IP router, 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 and complexity code that FLECS programmer is spared from writing, that is, the boilerplate code generated by the compiler.

6.1 Ethernet Bridging

We can model Ethernet bridging in FLECS with a single ASE called EthBridge, as shown in Figure 6.1. Figure 6.2 shows the EthBridge specification in ADL and Figure 6.3 defines the bridge configuration in FCL.

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 EthBridge which peeks at the Ethernet destination(`dest_mac`) and source (`src_mac`) (Figure 6.2, lines 14-15).

The control table lookup results in the sequential execution of `setup` and `forward` (Figure 6.3, line 7). The execution of `setup` results in a switching table entry that learns the path towards `src_mac`. The `forward` pattern looks up the switching table

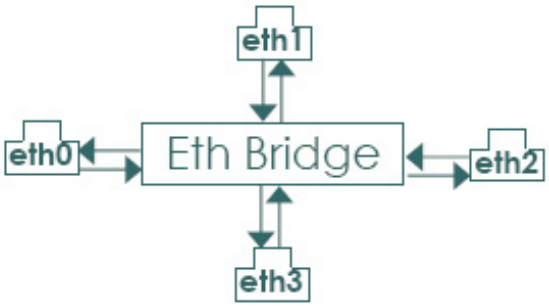

Figure 6.1: Ethernet Bridge in FLECS.

```
1   //
2   //  This is an implementation of a learning Ethernet bridge
3   //  which learns the reverse path from the source address
4   //
5
6   DEFINE ETHERNET_HEADER_LENGTH  14
7   DEFINE ETHERNET_ADDRESS_LENGTH 6
8   DEFINE DEST_MAC_OFFSET         0
9   DEFINE SRC_MAC_OFFSET          6
10
11  ASE EthBridge {
12
13   peek {
14     READ { dest_mac DEST_MAC_OFFSET ETHERNET_ADDRESS_LENGTH
15            src_mac  SRC_MAC_OFFSET  ETHERNET_ADDRESS_LENGTH }
16     CONTROL { dest_mac }
17   }
18
19   forward { // forwarding is based on destionation MAC lookup
20     LOOKUP { dest_mac }
21   }
22
23   setup {  // switching table is updated with the entry
24            //      [*,src_mac]->[prev,null];
25     UPDATE { * src_mac prev null }
26   }
27  };
```

Figure 6.2: EthBridge Specification in ADL.

```
1   //
2   // Ethernet Bridge configuration for two NICs
3   //      eth0 <-> bridge <-> eth1;
4   //
5   EthBridge bridge {
6    control { // standard bridge operations
7           [*, *] -> [setup/none][forward/none];
8    }
9    switching { // broadcast is the default entry
10          [eth$i, *] -> [eth-$i, null];
11   }
12  }
13
14  config(eth0, eth1, eth2, eth3) {
15      eth0 <-> bridge <->eth1;
16      eth2 <-> bridge <->eth3;
17  }
```

Figure 6.3: Ethernet Bridge Configuration in FCL.

for `dest_mac`. If the path to `dest_mac` is known, 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 broadcasted, which is the default configuration of the switching table (Figure 6.3, line 10). Figure 6.3 (lines 15-16) describes the connections of the EthBridge instance (`bridge`) to the network interfaces.

6.2 IP Routing

A simple IP router can be modeled in FLECS as shown in Figure 6.4. The corresponding configuration is shown in Figure 6.5. 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 6.5, 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

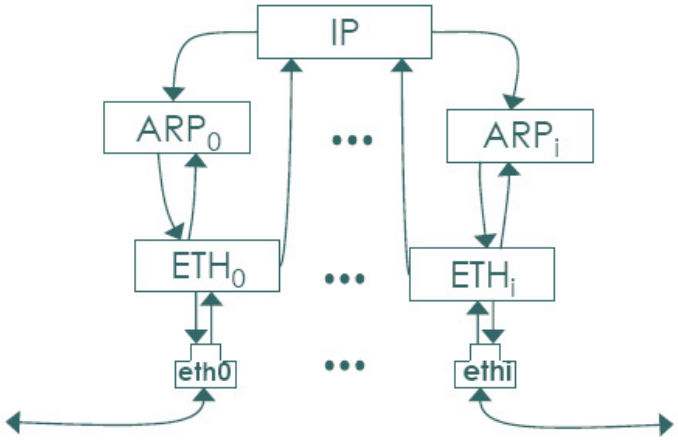

Figure 6.4: IP Router in FLECS.

IP address given in the switching table entry of IP ASE. 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. The example below shows how an ARP ASE is initialized in an IP Router configuration. A complete sample configuration is given in Figure 6.5.

```
ARP_1(ip=C0A80705, mac=00055DE6265E);
```

An interesting feature in the IP forward specification is the TTL decrement which is invoked on packets forwarded by the IP ASE.

```
forward {
  LOOKUP { dest_ip }
  %localTTLDecrement(p);
}
```

```
1  //      eth0 <-> Eth <-> Arp <-> IP <-> Arp <-> Eth <-> eth1;
2  //
3  Eth eth_$i {
4   control {
5    [eth$i, mac] ->  [forward/pop]; // packets directed at this MAC
6    [eth$i, FFFFFFFFFFFF] ->  [forward/pop]; // broadcast packets
7    [arp_$i, *]  ->  [recast][forward/none]; // ip or arp packets
8   }
9   switching {
10   [eth$i, mac#0806] ->  [arp_$i, null];   // arp packets
11   [eth$i, FFFFFFFFFFFF#0806] ->  [arp_$i, null]; // arp requests
12   [eth$i, mac#0800] ->  [ipswitch, null]; // ip packets
13   [arp_$i, *] ->  [eth$i, null];          // outgoing packets
14  }
15 }
16 Arp arp_$i {
17  control {
18   [eth_$i, 0001#ip] -> [rupdate][respond]; // ARP_REQUEST opcode = 0001
19   [eth_$i, 0002#ip] -> [rupdate];          // ARP_REPLY opcode = 0002
20   [ipswitch, *] -> [resolve][forward/pop+push]; // ip packets
21  }
22 }
23 Ip ipswitch {
24  control {
25   [*, *]       -> [forward/push]; // push appends the gateway information
26  }
27  switching {
28   [*, C0A80303] -> [arp_0, C0A80303]; // specific routing table entries for
29   [*, C0A80707] -> [arp_1, C0A80707]; //     this dual NIC configuration
30  }
31 }
32 // constant values to be used in each ASE instance are specified as
33 //   parameters along with an idenifier which can be used in the
34 //   initialization (above) and also in the ASE specs as local.<name>
35 //      for e.g. in Eth spec we can access 'mac' value as 'local.mac'.
36 config(eth0, eth1) {
37   eth0 <-> eth_0(mac=00055DE62C09) <-> arp_0(ip=C0A80305, mac=00055DE62C09);
38   eth1 <-> eth_1(mac=00055DE62C0A) <-> arp_1(ip=C0A80705, mac=00055DE62C0A);
39   eth_0 -> ipswitch <- eth_1;
40   arp_0 <- ipswitch -> arp_1;
41 }
```

Figure 6.5: Sample IP Router Configuration in FCL.

6.3 Network Address Translation

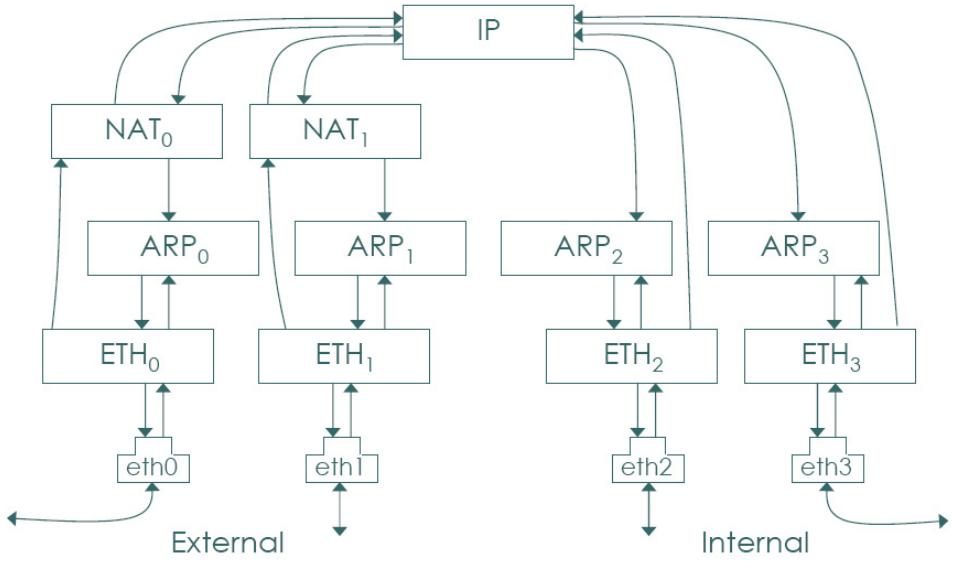

Figure 6.6: NAT Configuration in Flecs.

Figure 6.6 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.

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 (Figure 6.7 lines 21-26). VC acquires a local name and creates the virtual circuit entries in the switching table (Figure 6.7 lines 23-24). It rewrites the source IP and port in the `SWAP` block (Figure 6.7 lines 27-30) before calling `localRecalculateChecksums` on the packet. The `setup` pattern is called on packets from the internal network.

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   peek {
2     READ { src_ip IP_ADDRESS_LEN+SRC_IP_OFFSET IP_ADDRESS_LEN
3            dest_ip DEST_IP_OFFSET IP_ADDRESS_LEN
4            src_port IP_ADDRESS_LEN+SRC_PORT_OFFSET PORT_LENGTH
5            dest_port DEST_PORT_OFFSET PORT_LENGTH
6            protocol TRANS_PROTO_OFFSET TRANS_PROTO_LENGTH }
7     CONTROL { dest_ip dest_port }
8   }
9
10  forward {
11    LOOKUP { dest_ip dest_port }
12    SWAP {
13         WRITE {LOOKUP_NAME[0] DEST_IP_OFFSET IP_ADDRESS_LEN
14                LOOKUP_NAME[1] DEST_PORT_OFFSET PORT_LENGTH }
15    }
16    %localRecalculateChecksums(p);
17  }
18
19  setup {
20   LOOKUP { src_ip src_port }
21   VC {
22    LOC_NAME { local_name() }
23    UPDATE {* LOC_NAME prev src_ip#src_port
24            * src_ip#src_port local.next LOC_NAME
25    }
26   }
27   SWAP {
28      WRITE{LOOKUP_NAME[0] IP_ADDRESS_LEN+SRC_IP_OFFSET IP_ADDRESS_LEN
29            LOOKUP_NAME[1] IP_ADDRESS_LEN+SRC_PORT_OFFSET PORT_LENGTH }
30   }
31   %localRecalculateChecksums(p);
32  }
```

Figure 6.7: Pattern Specifications in NAT.

6.4 i3 Forwarding

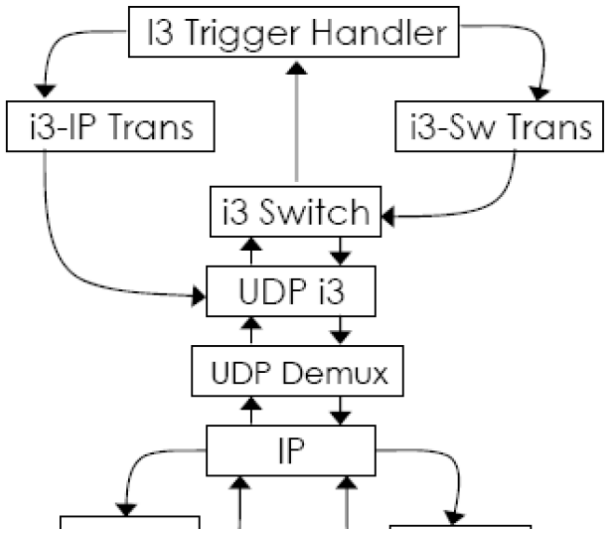

Figure 6.8: I3 Forwarding in FLECS.

Using the FLECS framework, i3 [32] becomes a straightforward implementation of forwarding using Chord [33] 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 6.8. 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 topmost 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.

Chapter 7

Evaluation

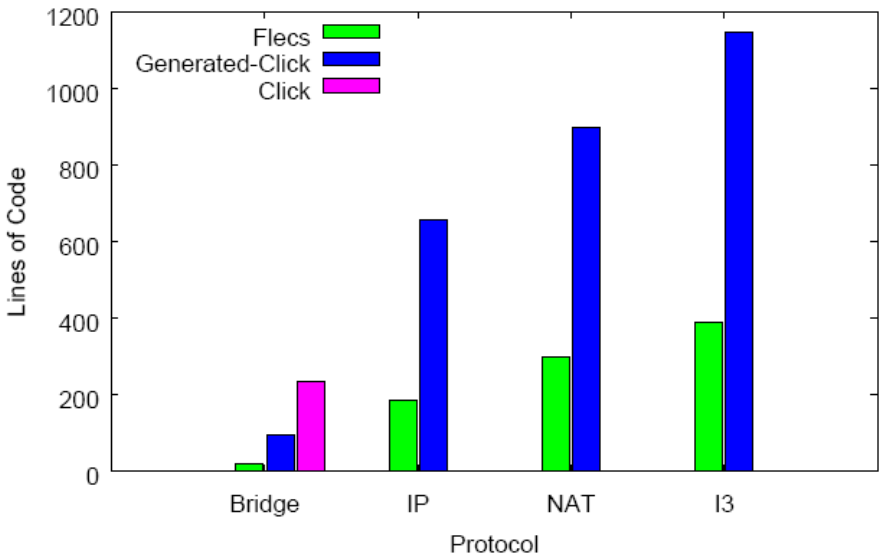

Figure 7.1: Comparison of the number of lines of code in the .ase file with the number of lines generated by the *asec* compiler.

To evaluate the effectiveness of the framework, we compare the number of lines of code written by the programmer and the ones generated by the compilers, `asec` and `confic`. This determines the gain in terms of the amount of code written to implement a protocol in Flecs compared to developing the corresponding implementation in Click. We have performed some experiments to compare the performance of Flecs

generated implementation of the IP router with the comparable Click implementation using regular Click elements.

7.1 Automated Code Generation

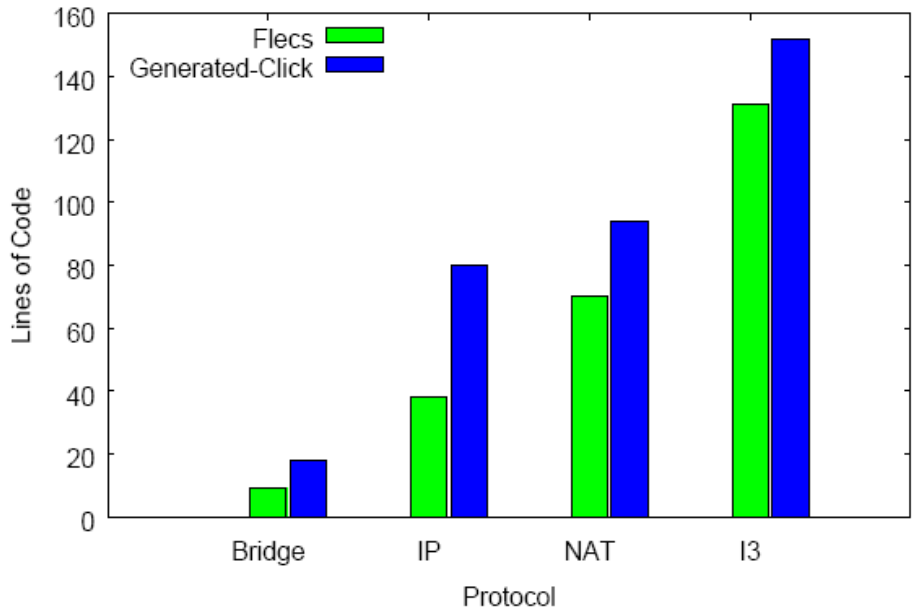

Figure 7.2: Comparison of the number of lines of code in the .ase file with the number of lines generated by the *confic* compiler.

Figures 7.1 and 7.2 demonstrate 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. The same implementation in Click results in around 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 around three thousand 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.

Figure 7.2 presents the comparison for the configuration files written in FCL and those generated by the `confic` compiler. The small difference in this case does not seem to justify the corresponding compiler implementation. However, it disregards the complexity and thus the difficulty of hand-coding the target protocol configuration, where the input and output ports of Ases have to be manually specified using Click configuration language.

7.2 Performance Measurements

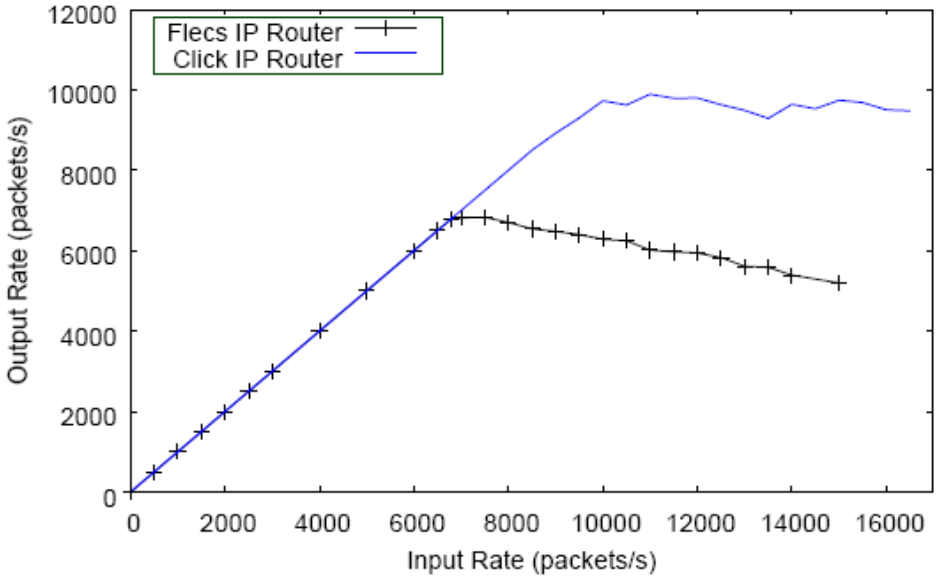

Figure 7.3: Forwarding Rates of an IP Router in Flecs.

The purpose of this evaluation is also to determine the cost of adhering to the axiomatic constraints and increased complexity of the Ases as opposed to the regular Click elements. Figure 7.3 characterizes the performance of an IP router in Flecs by measuring the rate at which it can forward 64 byte packets, when compared to a Click implementation of an IP router. 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. These changes have not been merged in the framework.

This is not a significant hit considering 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. With that said, these results are encouraging, considering that we have not yet incorporated any optimization techniques into our compilers. For this purpose, the `asec` compiler can utilize domain knowledge to apply optimizations and significantly improve the forwarding performance.

Chapter 8

Conclusions

This thesis 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.

The compilers in our framework translate meta-language programs specified by simple and easy to program data-structures. Complete forwarding functionality of complex communication protocols can be specified in a fraction of the code required to produce equivalent implementations. This not only saves the programmer from writing numerous lines of error checking code but also reduces the chances of bugs to be introduced in these implementations.

We demonstrate how non-trivial forwarders can be quickly expressed in the meta-languages defined in the FLECS framework. These specifications can then be used to generate protocol implementations. Careful language design constrains ASE processing to predefined axiomatic fundamentals to achieve conceptual clarity. This combined with auto-generation of protocol code eliminates error-prone human coding of complex protocols.

8.1 Discussion

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.

8.2 Contributions

In the context of implementing networking software, our objective is to explore the design space that lies between implementing protocols by hand and automatically generating protocol implementations from axiomatically constrained specifications. We claim that the following has been achieved.

- FLECS identifies and implements fundamental primitive operations of packet processing which are concise, making it easy to reason about protocols and yet powerful enough to implement a variety of forwarding schemes.
- Our design employs object-oriented techniques to implement the building blocks of network protocols. This greatly reduces protocol complexity by decomposing them into small components.
- We discovered some well-defined packet processing patterns which fit nicely into our axiomatic framework to support componentization of protocols and give the design a formal meaning.
- We describe a meta-language with formal semantics to define the core components (ASES) of protocol design. This allows us to concisely specify complex protocols in a readable format.
- FLECS supports a modular design of ASES, making them suitable for formal analysis and reasoning about fundamental properties of protocols.
- The framework uses an inheritance model, allowing packet processing code to be re-used for different implementations while making the framework easily configurable and flexible.

It should be noted that FLECS has been developed as a proof of concept for the axiomatic basis for communication [23] and is limited by the same set of limitations as the model, such as obliviousness to time, error and loss.

8.3 Future Work

Our conformance to the axiomatic basis not only allows us to discover different patterns in protocol implementations but also makes the design of our framework independent of any software or hardware architectures. It would be reasonable to state that FLECS can be implemented on other packet processing engines and network processors [9, 8]. Future implementations of FLECS may perhaps be able to generate validated protocol implementations for programmable hardware devices such as FPGA [4, 18]. 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. Using our inheritance model and defining more efficient data-structures in the BASE we can reduce per ASE processing overhead. Latency of the individual ASEs is not the only performance issue for network software implemented using FLECS. Maximizing throughput is also a pressing problem. We can focus on optimizations which actually contribute to increased throughput. These include use of more efficient data structures for ASE tables and manipulation of packet data.

Other aspects which still need some further work include exploration of alternate ways of representing protocol specifications more concisely which is also easier to understand and modify. Automated theorem provers can also be incorporated into the framework for automated generation of correctness proofs for specific implementations.

List of References

[1] M. B. Abbott and L. L. Peterson. A Language-Based Approach to Protocol Implementation. In *SIGCOMM '92*, pages 27–38, New York, NY, USA, 1992. ACM Press. 7

[2] 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

[3] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(14):25–59, 1987. 7

[4] S. Brown and J. Rose. Architecture of fpgas and cplds: A tutorial, 1996. 49

[5] 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. 7

[6] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating Efficient Protocol Code from an Abstract Specification. *IEEE/ACM Transactions on Networking*, 5(4):514–524, 1997. 1, 7

[7] D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *SIGCOMM '88*, pages 106–114, August 1988. 1

[8] D. Comer. *Network Systems Design with Network Processors*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004. 49

[9] D. Comer and L. Peterson. *Network Systems Design Using Network Processors*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003. 49

[10] T. Condie, J. M. Hellerstein, P. Maniatis, S. Rhea, and T. Roscoe. Finally, a Use for Componentized Transport Protocols. In *HotNets IV*, 2005. 13

[11] 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. 7

[12] D. Decasper, Z. Dittia, G. Parulkar, and B. Plattner. Router plugins: a Software Architecture for Next Generation Routers. In *SIGCOMM '98*, pages 229–240, New York, NY, USA, 1998. ACM Press. 1

[13] T. GenSSler and B. Schulz. Tool-Supported Component Evolution. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 175–182, London, UK, 1999. Springer-Verlag. 1

[14] T. G. Griffin and J. L. Sobrinho. Metarouting. In *Proceedings of SIGCOMM '05, Philadelphia, PA, USA*, pages 1–12, August 2005. 1

[15] 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. 2

[16] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969. 2

[17] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991. 7

[18] 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. 49

[19] 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. 8

[20] 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. 8

[21] D. B. Johnson and D. A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. *Mobile Computing*, 353:153–181, 1996. 35

[22] 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. 2, 6

[23] M. Karsten, S. Keshav, S. Prasad, and M. Beg. An Axiomatic Basis for Communication. In *SIGCOMM '07*, Kyoto, Japan, aug 2007. ACM Press. 2, 3, 6, 7, 9, 13, 31, 49

[24] 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. 5, 6, 22

[25] 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. 13

[26] 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. 1

[27] P. Mockapetris. RFC 1034 - Domain Names - Concepts and Facilities, November 1987. 35

[28] P. Mockapetris. RFC 1035 - Domain Names - Implementation and Specification, November 1987. 35

[29] 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. 6

[30] S. O'Malley and L. L. Peterson. A Dynamic Network Architecture. *ACM Transactions Computer Systems*, 10(2):110–143, 1992. 7

[31] C. Perkins. RFC 3344 - IP Mobility Support for IPv4, August 2002. 35

[32] 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. 2, 15, 42

[33] 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. 42

[34] D. L. Tennenhouse. Layered Multiplexing Considered Harmful. In *First International Workshop on High Speed Networking*, nov 1989. 7