Sequence Abstractions for Flexible, Line-Rate Network Monitoring

Andrew Johnson
Princeton University

Ryan Beckett
Microsoft Research

Xiaoqi Chen
Princeton University

Ratul Mahajan
University of Washington, Amazon

David Walker
Princeton University

Abstract

We develop FLM, a high-level language that enables network operators to write programs that recognize and react to specific packet sequences. To be able to examine every packet, our compilation procedure can transform FLM programs into P4 code that can run on programmable switch ASICs. It first splits FLM programs into a state management component and a classical regular expression, then generates an efficient implementation of the regular expression using SMT-based program synthesis. Our experiments find that FLM can express 15 sequence monitoring tasks drawn from prior literature. Our compiler can convert all of these programs to run on switch hardware in way that fit within available pipeline stages and consume less than 15% additional header fields and instruction words when run alongside switch programs.

1 Introduction

Many network management tasks involve recognizing and reacting to a user-defined sequence of packets. Such *sequence monitors* can enforce security policies, prioritize traffic, mitigate attacks, ensure protocol compliance, and more. For example, they can identify and de-prioritize video flows using a fingerprint based on successive packets to improve the network for other traffic [23] or verify that network clients faithfully implement protocols such as the Dynamic Host Configuration Protocol (DHCP) by observing the protocol exchange [27].

Ideal sequence monitors are *i*) *flexible*: can express a broad range of monitoring tasks; and *ii*) *line-rate*: can perform all processing directly in the data plane (hardware). Being line-rate allows sequence monitors to analyze all traffic passing through the switch, without needing the switch CPU or a remote server. Switch CPUs cannot process all packets at line rate, and using remote servers incurs high network overhead and reaction delays.

Existing sequence monitors sacrifice either flexibility or line-rate processing. Systems such as Aragog [27] can express most sequence monitoring tasks, but they run entirely in software (i.e., are not line rate). Programmable switches based on Protocol Independent Switch Architecture (PISA) [5] enable hardware-based sequence monitoring, but are programmed in languages such as P4 [4, 25] that are too low level, making it hard to express and debug sophisticated tasks [15, 28, 31]. Hybrid systems like Marple and Sonata [11, 21] run only partially on switches. Their core abstractions focus on data transformations such as filter, map, and fold operations rather than packet sequences. They are line-rate only if strong restrictions are placed on the allowed functions or if hardware could be redesigned [21].

In this paper, we present an abstraction of a packet sequence *pattern* that is both flexible and compiled directly to PISA-based hardware. It enables line-rate sequence monitoring without mirroring traffic to the switch CPU or a central controller. Recognizing patterns for sequence monitors directly in hardware is difficult due to stringent data access constraints in current programmable switches. PISA-based switches process packets using a series of *stages*. Each stage contains its own local memory and a number of arithmetic and logic units (ALUs) to perform computation. While some sequence monitors such as packet counting fit this architecture naturally, others that require tracking state across multiple packets are significantly more challenging to realize.

Our system, called FLM, allows programmers to write sequence monitors using a high-level, pattern-based language. Patterns are specified as regular expressions over packets, with the added ability to record packet parameters for later use. FLM programs can trigger local switch actions immediately upon matching a pattern, and they can monitor packets at any desired granularity (e.g., flow, host).

We convert FLM programs into imperative code using a novel core data structure, which represents a state machine maintaining a variable environment. We transform operations on this data structure into operations on PISA switch registers by carefully dividing them into variable update and transition code for a deterministic finite automaton (DFA), reflecting the pattern's match progress. Our implementation prioritizes line rate execution on existing network hardware like the Intel Tofino [13], for any accepted packet sequence. We have formally proven that our compilation process from patterns to pipeline stages preserves the original pattern semantics.

We evaluated FLM by encoding 15 monitoring tasks drawn from prior work [10, 15, 18, 21, 23, 24, 27, 28, 30, 31]. We find that we can express all of these tasks in 10-41 lines of FLM code, which demonstrates the flexibility of FLM. We also find that we can compile all of these tasks to the Intel Tofino switch, which demonstrates FLM's ability to provide line-rate monitoring. All of these tasks fit within the number of stages on the switch and consume less than 15% of additional metadata memory or instruction words when run alongside switch programs.

In summary, this work makes three main contributions:

- FLM, a language to express sequence monitors that can run at line-rate on a switch with a novel definition of a pattern syntax and semantics.
- Provably correct compilation from an FLM program to a state machine representation that runs on PISA hardware using a minimal number of stages.
- Evaluation that shows that FLM can express a wide variety of sequence monitoring tasks and compile them to existing network hardware.

2 Background

Our programming abstractions are broadly applicable to contexts where real-time recognition of event sequences is useful, (programmable ASICs, FPGAs, NICs, or software switches). Our implementation is designed for the PISA model. PISA architectures rely on a series of stages. To achieve a high processing rate and avoid memory access hazards such as contention, each stage has its own nearby memory region. Due to this memory layout, data can only be accessed by a single stage, and can only flow *forward* in the pipeline by changing the packet being processed. Within each stage, the switch is able to perform a Read-Modify-Write instruction on a value stored in its memory, where the modify step is specified by a micro-program called a Register Action. A core problem we tackled is writing a DFA transition function in such a way that it fits into one Register Action in order to be applied to packets at line rate. In this work, we consider a monitor recognizing patterns in packets passing through a single switch *pipeline*. A switch containing multiple pipelines operating in parallel could contain multiple monitors.

To simplify our implementation, we build on Lucid [24], an event-driven programming language for PISA switches. Lucid programs declare *events* (i.e., notifications of data plane packet arrival or network control signals) and corresponding *handlers* (i.e., code to react to events). Lucid programs are compiled to P4 code that runs on PISA switches. Both P4 and Lucid are useful intermediate languages simplify our implementation, but are not critical for FLM's key abstractions.

3 Example walk through

In this section, we walk through an example monitoring task for the DHCP protocol step-by-step to show how FLM enables network programmers to more easily build flexible, line-rate packet sequence monitors.

3.1 DHCP Anomaly Detection

Suppose a network operator wants to verify that DHCP, which enables clients to lease IP addresses from a server, is not being misused. DHCP begins with the client broadcasting a "Discover" message, to which the server responds with an "Offer" message with available IP addresses. The client sends a "Request" message for an address, which is confirmed with an "Acknowledge" from the server. The client is then expected to use the acknowledged IP until the end of its lease.

The operator wants to ensure that clients only use their assigned IP. This monitoring task can be performed on the access switch that processes all client communication, including that with the DHCP server. Misuse would appear as a packet sequence belonging to a client, identified by its MAC (link-layer address), of some number of packets (the DHCP protocol), a DHCP "Acknowledge," and then a packet whose source IP does not match the acknowledged one. Below, we show how FLM's core abstractions help achieve this goal.

Events. FLM programs are written in terms of events. For our task, we can define these two events:

```
event DHCP_Ack(int cip, int cmac);
event IP_Pkt(int sip, int smac);
```

Events are detected by parsing packets that arrive at the switch. From the `DHCP_Ack` packet, the system parses the DHCP message payload and extracts the client's MAC and new IP. Other packets are parsed as generic `IP_Pkts`, from which the source IP (`sip`) and MAC (`smac`) are extracted.

Patterns. FLM patterns are regular expressions over events, including concatenation (`.`) and closure (`*`). To begin to tackle the problem DHCP misuse, a programmer might create the following pattern, which identifies the presence of a `DHCP_Ack` amongst any number of other IP packets:

$$\texttt{IP\_Pkt}^* \ . \ \texttt{DHCP\_Ack} \ . \ \texttt{IP\_Pkt}^*$$

Recording parameters. The pattern above would match *any* use of the DHCP server. It recognizes an event sequence, but not the event parameters. Including one character for every possible IP address in the regular expression alphabet would make it too large. Instead, FLM patterns allow for the *binding* of parameters to recognize patterns over very large alphabets (e.g., all IP addresses). The operator can record the value of the client's assigned IP in the `DHCP_Ack` messsage by writing:

FLM Program

```
entry event DHCP_Ack(int cip, int cmac);
entry event IP_Pkt(int sip, int smac);
spec<2048> dhcp_misuse =
IDX = {
    | DHCP_Ack -> {hash(cmac)}
    | IP_Pkt -> {hash(smac)}
}
DETECT {
     IP_Pkt*
    .DHCP_Ack(@int assigned = cip)
    .(IP_Pkt(sip == assigned))*
    .IP_Pkt(sip != assigned)
} => {
    (...Some user response here...);
};
```

FLM Intermediate Representation

```
re<2048> dhcp_misuse =
     IP_Pkt*
    .DHCP_Ack(@int assigned = cip)
    .(IP_Pkt(sip == assigned))*
    .IP_Pkt(sip != assigned);
entry event DHCP_Ack(int cip, int cmac) {
    idx = hash(cmac);
    if (transition(dhcp_misuse, idx, this)) {
        (...user-defined response...);
    }}
entry event IP_Pkt(int sip, int smac) {
    idx = hash(smac);
    if (transition(dhcp_misuse, idx, this)) {
        (...user-defined response...);
    }}
```

Figure 1: An FLM program that monitors for DHCP misuse, and its translation with explicit state machine transitions.

```
DHCP_Ack(@int assigned = cip)
```

This pattern will match any `DHCP_Ack` packet, and record the value of its `cip` parameter in the new variable `assigned`. To check whether future packets use this IP, the operator can add a *predicate* over the parameters of an `IP_Pkt` by writing:

```
IP_Pkt(sip == assigned)
```

This pattern will match any `IP_Pkt` event whose `sip` parameter equals `assigned`. In our application, we also need the negation (`IP_Pkt(sip != assigned)`). Hence, using binding and predicates, the operator can now construct the following FLM pattern to detect DHCP misuse from a client:

```
IP_Pkt*
.DHCP_Ack(@int assigned = cip)
.IP_Pkt(sip == assigned)*
.IP_Pkt(sip != assigned)
```

Arrays of patterns. The above pattern characterizes an anomaly in the communications with a single client. In reality, an operator wants to monitor many clients. To track multiple clients, one specifies an *array* of patterns (in this case of size 2048), which will all be active simultaneously:

```
spec<2048> dhcp_misuse = ...
```

Next, one must specify a mapping of events to patterns, so all clients with the same MAC are identified and applied to the same pattern. The operator provides an indexing function, which computes an array index from the values carried by each event. For `DHCP_Ack` events, the index is the hash of the client MAC. For `IP_Pkt` events (to catch the *outgoing* packets), the index is the hash of the source MAC.

```
IDX = {
    | DHCP_Ack -> {hash(cmac)}
    | IP_Pkt -> {hash(smac)}
}
```

The expressions for each index calculation are user-defined. If the operator wishes to prevent hash collisions, they can implement algorithms such as probabilistic data structures or detect collisions by storing keys and siphoning overflows to a software controller, as in Sonata and Aragog [11, 27].

Responses. Finally, the operator will want to react to detected misuse somehow—perhaps by blocking the client or logging the anomaly. FLM allows users to react however they choose by invoking an arbitrary Lucid subroutine.

Putting it all together. The left side of Figure 1 shows the combination of all the features above in an FLM program implementing the DHCP specification. The `spec` declaration generates an array of size 2048, where each index represents one copy of the FLM pattern. The `IDX` block identifies the flows, the `DETECT` block contains the pattern, and the block after "=>" contains the response.

3.2 Compilation Overview

We compile FLM programs to Lucid and use the Lucid compiler to generate P4. While Lucid frees us from defining event parsers in P4, we must still map our pattern-based programs to PISA stages. This presents two key challenges:

1. To match a pattern, the program must both update the register holding the values of the variables stored in the pattern (for example, storing a particular client IP in the `assigned` variable in the DHCP example) and update the register holding the position in the pattern. However, this requires too much memory and computation to fit into a single stage and register, as it must in order to keep the state of the pattern up-to-date.

2. It is not even clear how to implement an arbitrary pattern that does not contain variable bindings. For example, consider a state machine representing a pattern without any variable bindings. A simple implementation of its transitions would take the form of:

```
//Compute the character using the old variables
if (event.type == DHCP_Ack) {
    c = ACK;
    //On ack, update the store value
    assigned := event.cip;}
if (event.type == IP_Pkt and sip != assigned) {
    c = IP0;}
if (event.type == IP_Pkt and sip == assigned) {
    c = IP1;}
//Synthesized mapping f from character to value
f(c){   if (c == ACK) {return 12;}
        if (c == IP0) {return 0;}
        if (c == IP1) {return 12;}}
//Synthesized mapping g from character to value
g(c){   if (c == ACK) {return 2;}
        if (c == IP0) {return 8;}
        if (c == IP1) {return 9;}}
//Synthesized update function to do transition
update_state(curr, x, y) {
    if(curr + y < 3){
        return x ⊕ 5;}
    else{
        return curr & 3;}}
//Update the stored state using the f and g maps
state := update_state(state, f(c), g(c));
```

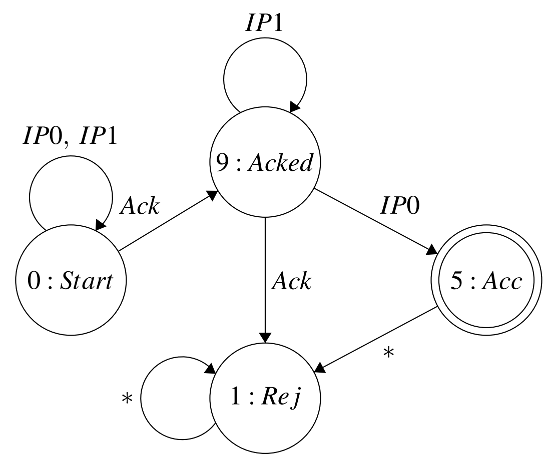


Figure 2: A DFA representation of the translated DHCP FLM pattern the memop for its transition function, and preamble code to compute the input character and variable updates. All integers are 4 bits, and the update function uses addition overflow to model the transitions correctly.

```
transition (state, input):
    if(state == s1 and input == i1):
        return s2;
    else if (state == s1 and input == i2):
        return s3;
...
```

This will contain more branches than are allowed in a single stage for most machines. However, the transition must fit into a single stage in order to read and write at line rate. The reason for this is that the transition depends on the current state of the machine, which is only available after reading a register. In PISA, registers can only be accessed once per pipeline pass, so the transition must occur at the same time as the read.

Solution overview. We solve these challenges by carefully compiling an FLM program in a series of steps. In a preprocessing step, we transform the FLM program to an intermediate representation. This step inserts explicit "transition" statements into each event handler that will be compiled away later, and keeps the FLM pattern definition as a global definition. In the first step, we break a pattern into a series of variable updates and predicate computations that computes an input character from a finite set determined by the event type and predicate evaluations. This solves the first challenge above, as we can move all of the variable storage and predicate computation into earlier stages. In the second step, we transform a pattern without bindings into a classical regular expression over the input characters from the first step, and synthesize an implementation of the corresponding DFA in a single stage. Condensing the transition function into a single stage solves problem 2, but is difficult as it requires searching through all possible state numberings and bit-wise ALU operations for one that satisfies all transitions. We offload this hard work to an automated SMT solver.

Preprocessing on DHCP. We translate the DHCP FLM program into an intermediate representation that allows for more control of the inputs to an underlying state machine. The core data structure is `re<size>`, which defines an array of state machines with `size` indices that match an FLM pattern. The pattern is copied from the high-level program. To interact with it, the expression `transition(name, idx, ev)` applies the event `ev` to the state machine at index `idx`, and evaluates to a boolean indicating whether or not the sequence of events applied to it so far matches the FLM pattern `name`. For the DHCP example, in the handler of each event, we compute the index using the expressions provided in the high-level program (hashing the MAC address). Then, we add `transition(dhcp_misuse, idx, this)`, where `this` represents the event for the current handler. If that returns true, we run the user-defined response code. The right side of Figure 1 shows the intermediate representation with explicit transition statements in event handlers.

Step 1 for DHCP. The next two steps compile the remaining FLM pattern and `transition` statements from the intermediate representation into simple assignments and register operations. We will refer to the pseudocode in Figure 2, which represents the implementation of one `transition` statement. First, we separate out the variable bindings. For the DHCP example, it is enough to store the parameter of the first `DHCP_Ack` event in the variable `assigned`. Furthermore, we wish to compute an input character `c` from a finite alphabet by evaluating the predicates in the FLM pattern. This alphabet is composed of all of the event *types* of the original, followed by bit strings representing the values of the predicates. Because `IP_Pkt` appears with a predicate, it is expanded to the letters `IP_Pkt1` and `IP_Pkt0`. `DHCP_Ack` does not appear with one, so it stays as is. These translations are shown in Lines 1–9 of Figure 2.

Step 2 for DHCP. In this step, we will translate the FLM pattern into a classical regex over the alphabet described above, and then implement its transition in a single stage. To translate the pattern, events that appear with a predicate become *unions* of any event with the same type where that predicate is true; similarly, events without a predicate are unions of any event of their respective type. Other constructs such as concatenation and closure remain as they are. For this example, the translated alphabet is the set {`DHCP_Ack`, `IP_Pkt0`, `IP_Pkt1`} and the classical regex is:

```
(IP_Pkt0 + IP_Pkt1)*
.DHCP_Ack
.IP_Pkt1*
.(IP_Pkt0)
```

Next, we translate this classical regex into a DFA and synthesize its implementation in a single pipeline stage. This is required as a naive implementation of the transitions would not fit in the limited computation available in one stage. Instead, we search through all of the state numberings and bit-wise operations to find ALU operations that complete all transitions correctly. On the right side of Figure 2, we show a picture of the DFA representing the above regex. To implement it, we take advantage of the fact that a single register read-modify-write action can take up to two arguments computed in prior stages. Given a DFA, we search for the following:

- A mapping from states to integers, as shown by numbers preceding each state in Figure 2 (e.g. *Start* is numbered 0).
- Two mappings (`f` and `g`) from the alphabet to integers that will be used as inputs to the read-modify-write instruction. These are shown on lines 11 and 15 of Figure 2. Because they can be computed in earlier stages, we can use lookup tables to implement them, which are not available when updating the DFA state.
- A read-modify-write instruction that implements the transition function of the DFA using operations available on the switch and results from the `f` and `g` mappings. This is shown on line 19 of Figure 2.

The code in Figure 2 is laid out on the switch to compute `c, f,` and `g`. In Figure 1, the `re` definition is replaced with a register definition, and each `transition` statement is replaced with the code in Figure 2: it first reads and updates the variables at the current index, then computes the input character and its corresponding mapping values, and finally applies the transition function to the state register with those values. It outputs whether the result represents an accepting state in the DFA (in this example, result was 5 for "Acc").

4 FLM Language Definitions

In this section, we describe the FLM language, provide its regular-expression-like syntax, and define the language's semantics over packet traces. In section 6, we prove that our compiler translations are correct: the low-level switch program correctly implements the high-level pattern semantics.

4.1 FLM Language

The FLM language is a wrapper to provide access to the expressive FLM patterns. An FLM program consists of:

1. A name and size, written `spec<i> myname = ...` where `i` is the number of replicated state machines.
2. An `IDX = {...}` block which determines which index to use for each event.
3. Optionally, a `DATA {...}` block that declares one or more registers to be used for a stateful response to a sequence match (for example, counting matches).
4. A `DETECT{pat} => {response}` block, where `pat` is an FLM pattern and `response` is Lucid code indicating what to do when the pattern is recognized.

Finally, to recognize properties such as liveness or timeouts, such as detecting half-open TCP queries [21], we provide a special event called `maintenance`. This event is guaranteed to eventually visit every state machine in a spec, so it acts as a final event to match a pattern that might never observe any more packets arriving. More about maintenance events is included in the Appendix.

4.2 Syntax and Semantics of Patterns

In each FLM program, there is a finite set A of event *types*, such as `DHCP_Ack` and `IP_Pkt`. An *event* is a pair of an event type $a \in A$ and an integer z, written $a\langle z\rangle$. While this only includes events with a single parameter, it generalizes easily to any number of parameters. The top of Figure 3 shows the syntax of an FLM pattern. We allow *predicates* over the parameters of events ($a\langle p\rangle$) to denote events of type a whose parameter satisfies p. We also allow *binding* parameters in events ($a\langle @y; p\rangle$) for use in predicates.

Patterns (and their contained predicates) are evaluated under an *environment*. An *environment* E is a mapping from variables to integers, with its domain denoted by $Dom(E)$. The empty environment is denoted by ".". A *predicate* p is a

FLM pattern $\boxed{R}$

$$
\begin{array}{llll}
p & & & \textit{predicate} \\
r ::= & & \emptyset & \textit{empty set} \\
 & | & \varepsilon & \textit{empty string} \\
 & | & a\langle p\rangle & \textit{event} \\
 & | & a\langle @y;p\rangle.r_1 & \textit{binding event} \\
 & | & r_1.r_2 & \textit{concatenation} \\
 & | & (r_1)^* & \textit{closure} \\
 & | & r_1 + r_2 & \textit{union} \\
 & | & r_1 \,\&\, r_2 & \textit{intersection}
\end{array}
$$

Denotational Semantics $\boxed{[\![r]\!]_E : \text{Set of strings}}$

$$
\begin{array}{ll}
[\![\emptyset]\!]_E & = \emptyset \\
[\![\varepsilon]\!]_E & = \{\varepsilon\} \\
[\![a\langle p\rangle]\!]_E & = \bigcup_{z\in\mathcal{Z}} \{a\langle z\rangle | [\![p(z)]\!]_E\} \\
[\![a\langle @y;p\rangle.r]\!]_E & = \bigcup_{z\in\mathcal{Z}} (\{a\langle z\rangle | [\![p(z)]\!]_E\} \circ [\![r]\!]_{E,y\leftarrow z}) \\
[\![r+s]\!]_E & = [\![r]\!]_E \bigcup [\![s]\!]_E \\
[\![r\&s]\!]_E & = [\![r]\!]_E \bigcap [\![s]\!]_E \\
[\![r.s]\!]_E & = [\![r]\!]_E \circ [\![s]\!]_E \\
[\![r^*]\!]_E & = \bigcup_{n\in\mathcal{N}} [\![r^n]\!]_E
\end{array}
$$

Auxillary definitions

$$
\begin{array}{ll}
R\circ S & = \{r.s \mid r\in R \wedge s\in S\} \\
r^0 & = \varepsilon \\
r^{i+1} & = r.r^i
\end{array}
$$

Figure 3: Technical cheat sheet. Definitions for FLM patterns and their derivatives.

function from integer to boolean that may contain one or more free variables, and is *closed under an environment E* if the free variables of p are contained in $Dom(E)$. The evaluation of a predicate p applied to an integer z under an environment E is denoted by $[\![p(z)]\!]_E$, and exists if p is closed under E.

In this section (except for the DHCP pattern, for continuity), we use lambda notation to define predicates. For example, $\lambda x.(x \geq 10)$ is a predicate that returns true if the given integer is at least 10. We use the standard semantics of lambda functions. Finally, an FLM pattern r is *closed under an environment E* if all of its free variables appear in E.

On the bottom of Figure 3, we show the semantics of an FLM pattern. Each FLM pattern defines a set of strings of events that belong to its language. A binding has a scope for its variable. Predicates within the scope can use the variable. For example, $[\![a\langle @y;\lambda x.true\rangle.(b\langle\lambda x.(x == y)\rangle)]\!]_.$ is the set of any event of type a (the predicate is always true) followed by one of type b with the same parameter (e.g. $a\langle 12\rangle.b\langle 12\rangle$). Constructors of FLM patterns are defined similarly to those of classical regular expressions; $[\![r+s]\!]_E$ and $[\![r\,\&\,s]\!]_E$ represent union and intersection of the sets $[\![r]\!]_E$ and $[\![s]\!]_E$, respectively, and $[\![r^*]\!]_E$ represents zero or more copies of $[\![r]\!]_E$.

4.3 FLM Intermediate Representation

The FLM intermediate representation simplifies the higher-level language features to leave just the patterns. It includes two new features not present in Lucid:

1. `re<i> myname = pat` is a statement that defines an array of `i` finite state machines named `myname`, which each recognize the FLM pattern `pat`.
2. `transition(myname, idx, ev)` is an expression that applies a transition with the event `ev` to the state machine at index `idx` of `myname`. It evaluates to `true` if the state machine is in an accepting state (the pattern has been recognized), and `false` otherwise.

An FLM program is transformed into the definition of a state machine with the same pattern, size and name. Then, at the beginning of each event handler, the compiler adds the following code:

```
if (transition(myname, idx, this)) {
    response;
}
```

`myname` is the name of the state machine, `this` is the event to transitioned with (the current event for the handler), `idx` is computed using the `IDX` block, and `response` is the user-defined response.

5 From FLM patterns to Regular Expressions

We showed in section 3 how to build a DFA and some preamble code for the DHCP example. Here, we describe more generally how to translate an FLM pattern into a regular expression, which we translate to a DFA in section 7. Due to hardware restrictions, we cannot complete all of the actions necessary to store variables, evaluate predicates, and transition pattern state machines in one stage, so our plan is to carefully separate those operations into a series of stages. First, we lift variable bindings out of patterns, then remove predicates to reduce the problem to implementing a finite state machine over a finite alphabet where events are paired with bits representing the predicates in a pattern.

5.1 Lifting out variable bindings

A binding FLM pattern has the form $b\langle @y;p\rangle.r$. These may occur deep within a pattern, posing a problem for implementing the variable bindings in a pipeline stage before the pattern state. In order to place bindings in an earlier stage, binding occurrences must depend only on the incoming event and environment, not the state of the pattern. We move the bindings to the top-level of a pattern while preserving its semantics by introducing a new form of patterns:

$$b\langle @y\rangle \triangleright r$$

Intuitively, this construct binds the first occurrence of an event with type b's value to the variable y, and then proceeds with matching r. The key aspect of the $\triangleright$ syntax is that it separates the binding out from the rest of the pattern. Our goal is to move these bindings all the way to the top-level using *rewrite rules*, to get a pattern that is written as a series of bindings, followed by a pattern without any variable changes at all. We show how this works on the DHCP example:

```
IP_Pkt*
.DHCP_Ack(@int assigned = cip)
.(IP_Pkt(sip == assigned))*
.IP_Pkt(sip != assigned)
```

The first rule converts an "@" binding into a "$\triangleright$" one in place.

```
IP_Pkt*
.DHCP_Ack(@int assigned = cip)▷
    (DHCP_Ack
    .IP_Pkt(sip == assigned)*
    .IP_Pkt(sip != assigned))
```

The second rule moves the binding up by one level.

```
DHCP_Ack(@int assigned = cip)▷
    (IP_Pkt*
    .DHCP_Ack
    .IP_Pkt(sip == assigned)*
    .IP_Pkt(sip != assigned))
```

This pattern has the same semantics as the original, but is in *prefix form*: a binding followed by a *binding-free* pattern.

Definition 1. *An FLM pattern s is binding-free if it contains neither $\triangleright$ nor @.*

In the last DHCP example above, the binding-free pattern is the portion after the $\triangleright$.

Definition 2. *An FLM pattern r is in prefix form if it is written as $B \triangleright s$, where B is a series of bindings ($b_1\langle @y_1\rangle \triangleright b_2\langle @y_2\rangle \ldots$), and s is binding-free.*

The last version of the DHCP pattern above is in prefix form. FLM patterns in prefix form cannot contain @ bindings, as they are all converted to the $\triangleright$ syntax. The semantics of the $\triangleright$ operator is the union of two sets. The first covers cases when the binding is required. In this case, the binding $b\langle @y\rangle \triangleright r$ should bind the value of the first occurrence of event b to the variable y. The second covers cases when the variable y is *not* used to match the pattern. For example, the pattern:

$$b\langle @y\rangle \triangleright (a\langle \lambda x.true\rangle + (b\langle \lambda x.true\rangle .a\langle \lambda x.x == y\rangle))$$

is meant to define either any string with a single event of type a, or a sequence of an event of type b followed by a where their parameters match. However, in the case of a single a event, the binding is not needed. For this case, we quantify over *all* possible values for y when defining it, which ensures the value of y does not matter for matching.

Definition 3.

$$\begin{aligned}\llbracket b\langle @y\rangle \triangleright r\rrbracket_E &= \{w_1.b\langle z\rangle .w_2 \in \llbracket r\rrbracket_{E,y\leftarrow z} \mid b \notin w_1\}\\ &\cup \{w \mid b \notin w \textit{ and } \forall z.w \in \llbracket r\rrbracket_{E,y\leftarrow z}\}\end{aligned}$$

In section 6, we show that if the rewrite rules can transform a regular expression into a new one that is in prefix form, it can always be implemented in a pipeline. A full list of rewrite rules is contained in the Appendix. For all of these rules, we show that if r is rewritten to r', then for all environments E under which r is closed, $\llbracket r\rrbracket_E = \llbracket r'\rrbracket_E$.

Unimplementable patterns The rewrite rules are not complete. Some patterns cannot be easily rewritten into prefix form to be recognized in a pipeline. For example, the following pattern fails to reach prefix form:

$$a\langle \lambda x.true\rangle .a\langle @y\rangle .a\langle \lambda x.x == y\rangle$$

This is meant to define a sequence of three events of type a, where the parameters of the second and third events match. This is semantically well defined, but it cannot be implemented easily because the variable updates happen *before* accessing the pattern state. When an a event arrives, the decision to record y must be made without knowledge of previous events. Our rewrite rules would reject this pattern.

5.2 Translating events

Now, we will focus on recognizing a *binding-free* pattern by translating it into a DFA over a new alphabet.

Alphabet. The alphabet is formed by combining all of the event *types* of the pattern with all possible combinations of values of the predicates. As shown in subsection 3.2, the alphabet for the DHCP example is {`DHCP_Ack`, `IP_Pkt0`, `IP_Pkt1`}. `IP_Pkt` appears with a bit representing the value of the associated predicate, and `DHCP_Ack` remains as it is because it does not appear with a predicate.

In general, the alphabet for the translation of an FLM pattern r is defined as follows, where $\text{bin}(n)$ is the binary representation of a natural n, $P(a)$ is the list of all of the predicates in r for event type a, and A is all of the event types:

$$\{a.\text{bin}(n) \mid a \in A \wedge n < 2^{len(P(a))}\}$$

Events. We need to take a single *concrete* event $a\langle z\rangle$ and output a character in the new alphabet. To do this, we define the *letter* translation T_l, which keeps the event type and appends each predicate's evaluation under the given environment. In our example, consider an `IP_Pkt` event where `sip = 10.0.0.1`. If the environment contains the mapping from `assigned` to `10.0.0.1`, then the translated letter is `IP_Pkt1`. Otherwise, it is `IP_Pkt0`.

Definition 4. *The event translation*

$$T_l(a\langle z\rangle, E, P) = a\llbracket p_1(z)\rrbracket_E \ldots\llbracket p_n(z)\rrbracket_E$$

5.3 Eliminating predicates from patterns

Now, we translate a binding-free pattern into a classic regular expression over a finite alphabet by eliminating all remaining predicates. The translation, called T_{re}, maps events with predicates to *unions* of characters with the same event and the corresponding predicate being true. The formal definition is in the Appendix. For example, `IP_Pkt(sip == assigned)` is translated to the pattern `IP_Pkt1`. Because `IP_Pkt` alone specifies no predicates, it is translated to the pattern `(IP_Pkt0 + IP_Pkt1)`. Union, intersection, concatenation, and closure all just apply the translation recursively. The full translation of the DHCP pattern is:

```
(IP_Pkt0 + IP_Pkt1)*
.DHCP_Ack
.IP_Pkt1*
.IP_Pkt0
```

6 Translation Correctness

In this section, we present the theorem of correctness for our translations. Intuitively, this means that a *translated* string of events is accepted by a *translated* regular expression exactly when the original string of events is in the language of the original pattern, assuming that the variable updates are performed correctly. We first introduce the concept of *derivatives*, which formalize what should happen when a single event is processed. Then, we state our main theorem, which relates a series of derivatives to processing using our translations.

6.1 FLM Pattern Derivatives

Derivatives of FLM patterns formalize what happens when one event arrives. We will define one for binding-free FLM patterns and one for lists of bindings. A derivative of a pattern, D_{re}, is taken with an event and an environment. It outputs a new pattern, which represents the *remaining* pattern to be matched. We illustrate this by example with the DHCP pattern matching a string of events on the left of Figure 4. The last pattern is ε, which means that the original DHCP pattern accepts the string of events. The full derivative rules used are shown on the right of Figure 4. This also shows the binding derivative, which takes a series of bindings, an event, and an environment, and outputs a new binding and environment with the correct variable updates.

Formally, we show that the outputted pattern of a derivative contains all the strings that would form word in the language of the input pattern when concatenated to the input event:

Theorem 1. *For all a,z,s,E where s is binding-free and closed under E:*

$$[\![D_{re}(a\langle z\rangle;s;E)]\!]_E = \{w \mid a\langle z\rangle.w \in [\![s]\!]_E\}$$

6.2 Correctness Theorem

Our translations are correct if, after we translate an FLM pattern into a DFA with T_{re}, and feed it a string of events translated with T_l, the DFA accepts only when the original string of events is in the language of the original pattern.

We show that the relation between a pattern s and its translated DFA via T_{re} is preserved when transitioning the DFA using characters translated with T_l. In particular, taking the FLM pattern derivative of s with an event and then translating it to a DFA is the same as transitioning $T_{re}(s)$ with a translated event. At the end of a string of events, we test whether the *translated DFA* is accepting, which is equivalent to string being in the language of the original pattern.

To state this formally, we define the translation of a *word* (a string of events), which repeatedly applies T_l and D_{bind} to transform the word into a string of finite-alphabet characters, given a list of bindings, an inital environment, and a list of predicates. The translated example word from Figure 4 would be `DHCP_Ack.IP_Pkt1.IP_Pkt0`.

Definition 5. *The word translation*

$$T_w(a_1\langle z_1\rangle \dots a_n\langle z_n\rangle, B, E, P) =$$
$$T_l(a_1\langle z_1\rangle, E, P).T_w(a_2\langle z_2\rangle \dots a_n\langle z_n\rangle, B', E', P)$$

Where $B', E' = D_{bind}(a\langle z\rangle; B; E)$

The word translation of ε is ε. The rewrite rules described in subsection 5.1 preserve the semantics of patterns. They also preserve a property we call *implementability*, which means that the derivative of a pattern with events that are not binding is semantically equivalent no matter the assignments to unbound variables. This property holds if input patterns bind variables using the *first* occurrence of an event type, and always use variables after they are bound. We show an unimplementable pattern in section 5. For the technical definition, see the Appendix. Finally, we have our correctness theorem, which states we can check whether a *translated* word is in the language of a *translated* regular expression to determine if a word matches a pattern. $Asgn0(B)$ simply assigns 0 to each variable of B, so that there are never undefined variables. The proof is by induction on the length of a word and is contained in the Appendix.

Theorem 2. *For any word w, binding list B, pattern s, environment E, and predicates P, if $B \triangleright s$ is in prefix form, closed under E, and implementable, then:*

$$T_w(w, B, (E, Asgn0(B)), P) \in L(T_{re}(s,P)) \iff w \in [\![B \triangleright s]\!]_E$$

7 DFA Synthesis

The previous section reduced the problem of matching FLM patterns to matching specially constructed regular expressions, but it is still not clear how to do this at line rate. In this section,

FLM pattern Derivative $\boxed{D_{re} : Event \to R \to E \to (R,E)}$

$$
\begin{aligned}
D_{re}(a\langle z\rangle;\emptyset;E) &= \emptyset \\
D_{re}(a\langle z\rangle;\varepsilon;E) &= \emptyset \\
D_{re}(a\langle z\rangle;b\langle p\rangle;E) &= \begin{cases} \varepsilon & \text{if } a == b \text{ and } [\![p(z)]\!]_E \\ \emptyset & \text{otherwise} \end{cases} \\
D_{re}(a\langle z\rangle;r.s;E) &= D_{re}(a\langle z\rangle;r;E).s + v(r).D_{re}(a\langle z\rangle;s;E) \\
D_{re}(a\langle z\rangle;r+s;E) &= D_{re}(a\langle z\rangle;r;E) + D_{re}(a\langle z\rangle;s;E) \\
D_{re}(a\langle z\rangle;r\,\&\,s;E) &= D_{re}(a\langle z\rangle;r;E)\,\&\,D_{re}(a\langle z\rangle;s;E) \\
D_{re}(a\langle z\rangle;r^*;E) &= D_{re}(a\langle z\rangle;r;E).r^*
\end{aligned}
$$

Nullability $\boxed{v : R \to R}$

$$
\begin{aligned}
v(\varepsilon) &= \varepsilon \\
v(\emptyset) &= \emptyset \\
v(a\langle p\rangle) &= \emptyset \\
v(r.s) &= v(r).v(s) \\
v(r+s) &= v(r)+v(s) \\
v(r\&s) &= v(r)\&v(s) \\
v(r^*) &= \varepsilon
\end{aligned}
$$

Binding update $\boxed{D_{bind} : Event \to B \to E \to (B,E)}$ Let $B',E' = D_{bind}(a\langle z\rangle,B,E)$ in the following definitions:

$$
\begin{aligned}
D_{bind}(a\langle z\rangle,\varepsilon,E) &= \varepsilon,E && \text{no bindings} \\
D_{bind}(a\langle z\rangle,b\langle @y\rangle \triangleright B,E) &= b\langle @y\rangle \triangleright B',E' && if\, a \neq b \\
D_{bind}(a\langle z\rangle,a\langle @y\rangle \triangleright B,E) &= B',(E',y \leftarrow z) && if\, a = b
\end{aligned}
$$

Figure 4: The progress of matching a string with the DHCP example. The left column contains the binding-free patterns, and the right tracks the environment. The arrows indicate the pattern derivative with the incoming event and current environment. Acceptance is indicated by the empty string, ε.

```
memop template (st, f, g):
  b1 = [st,0] + [f,g,0] [==, !=, <, >] c1;
  b2 = [st,0] + [f,g,0] [==, !=, <, >] c2;
  if (b1 [||, &&] b2):
    return [st,c3] [|,&,+,⊕] [f,g,c4];
  else:
    return [st,c5] [|,&,+,⊕] [f,g,c6];
```

Figure 5: A syntax template for a single register action to be synthesized. Blue-bubbled brackets represent choices between the expressions in the brackets. Red-bubbled brackets represent choices between the operators in the brackets. Each c_i represents a constant chosen by the synthesizer.

we show how to synthesize code to perform the classical regex derivative (a DFA transition) in order to use theorem 2. We use SMT-based synthesis to fit a transition function into at most four register actions, the maximum allowed on the Tofino.

7.1 Synthesis goal

To implement a DFA's transition function within the allowed register actions, the synthesizer will attempt to cleverly assign numbers to the DFA states and alphabet while generating a short function composed of a fixed number of simple instructions like bitwise operations, arithmetic operations, and conditional branches. The function will calculate the next state given the current state and input event without the need to enumerate DFA transitions.

We take as input a DFA and a bitvector length l. A DFA is a five-tuple (Q,Σ,δ,q_0,F), where: $Q = \{q_0,q_1,\cdots\}$ is a finite set of states with initial state q_0, $\Sigma = \{\sigma_0,\sigma_1\cdots\}$ is a finite set of alphabet symbols, $\delta : Q\times\Sigma \to Q$ is the transition function, and $F \subseteq Q$ is the set of accepting states. For the DHCP example from subsection 3.1, the DFA has $Q = \{Start, Acked, Acc, Rej\}$, $\Sigma = \{Ack, IP0, IP1\}$, $F = \{Acc\}$, and δ as shown in Figure 2.

We output a function to implement the state machine's transition on the Tofino or any other hardware whose memory update is at least as expressive. Specifically, we output:

- A mapping R from Q to $\{0,\ldots,2^l-1\}$ that uniquely numbers the states, and by convention we fix $R(q_0) = 0$.
- Two mappings f and g from Σ to $\{0,\ldots,2^l-1\}$. These are passed as arguments to the register actions.
- A mapping *whichop* from Σ to $\{0,1,2,3\}$ that indicates which register action each character will use.
- Up to four register actions that take the values of the maps R, f, and g on the current state q and input character σ, and output $R(\delta(q,\sigma))$ for all the letters in Σ that use them. Furthermore, the functions must follow the syntax from Figure 5 in order to fit into a single register action.

An example of correct outputs for the DHCP example is also shown in Figure 2 (all characters use the same function).

7.2 Synthesis implementation

To come up with these outputs, we use an SMT solver to do syntax-guided synthesis [1]. We make one bitvector variable for each state (the values of R), two bitvectors for each letter (f and g), and booleans to determine *whichop*. To make the templates, we make boolean indicator variables for which operations, comparisons, and boolean comparisons are used. Then, we encode the templates as constraints over the state variables. If a satisfying assignment is found, we read it to get the output. An interesting problem is to find the best template for synthesis. This is discussed more in subsection 9.3.

8 Implementation

We implement the FLM compiler atop the Lucid framework [19, 24] using approximately 1500 lines of OCaml available on GitHub[1]. We implemented DFA synthesis code in the compiler in OCaml using the z3 SMT solver [29]. It first transforms each FLM pattern into prefix form and translates it to a classical regular expression. Then, it converts the pattern into a DFA and runs syntax-guided synthesis to generate the corresponding mappings and memops, expressed as an intermediate representation Lucid program. This program is subsequently compiled using the existing Lucid framework's backend and vendor-provided P4 compiler (`bf-p4c`) to generate the final data plane program binary. We use `ocamlc` 4.14.0, `z3` 4.11.2, and `bf-p4c` 9.13.0.

9 Evaluation

We evaluate FLM by using it to implement a diverse set of sequence monitoring tasks of interest to network operators. We identified 15 such tasks from prior work, and implemented them alongside a Lucid program that used the same events. All of the monitored patterns are listed in the Appendix.

Figure 6a shows these programs and summarizes our results, including the lines of code needed to express the examples, the synthesized DFA complexity and synthesis time, and the stages used. We are able to express each of the 15 tasks in the FLM language, pointing to its flexibility. The table shows lines of code as a proxy for ease of use. We see that FLM programs are short and all tasks are expressed in a few 10s of lines. In contrast, when translated to Lucid these programs are 5-10x bigger, which is a proxy for implementation effort of expressing these tasks directly in P4.

Our compiler is able to compile each of the programs to the Intel Tofino, which demonstrate the line-rate monitoring capabilities of FLM. We discuss compilation results in more detail next.

9.1 Compilation time

Figure 6b shows the compilation time for each program on an AWS EC2 t3.medium server with 4GB memory and 2 vCPU (unlimited burst). We break the total compilation time into the frontend (Lucid compiling) and backend (P4 compiling). Note that the Lucid time *includes* the synthesis time from Figure 6a. We see that most programs need little time (seconds) to complete our compilation steps. Although the DFA synthesis step depends on the complexity of the pattern and is theoretically NP-Hard, all tasks finish in a few minutes.

Programs decorated with FLM have slower compilation time for both the Lucid compiler backend and the vendor P4 compiler. This is caused by the additional statements added to the IR and the resulting overhead for optimizing the pipeline layout, and mostly depends on the complexity of the original program. The complexity of the pattern (aside from additional variables) does not affect the backend's and vendor compiler's compiling time. All implementations take roughly the same time to compile once synthesized.

9.2 Hardware Resource Utilization

Figure 6c shows the hardware resource usage of the Tofino binaries for each program. The three most relevant metrics for FLM are the number of pipeline stages used (Figure 6a), the percentage of metadata fields (PHV) allocated, and the percentage of instruction words (VLIW) allocated.

Depending on the complexity of pre-processing involved in translating events (packets) to letters in the pattern, FLM compiles programs into 6-10 stages, all comfortably fitting within the Intel Tofino v1 (12 stages). The DFA itself always uses a single stage, regardless of the complexity of the pattern. The additional stages take care of the Lucid event handling and control flow, as well as predicate computation. Because there are data dependencies between stages arising from control flow, these stages are not always "full;" additional unrelated programs can fit alongside them without using many more stages, as the preprocessing steps are parallelized alongside existing logic.

Meanwhile, FLM programs use reasonable resources: adding FLM to an existing Lucid program only consumes 1-15% additional PHV and VLIW, much of which is for setup (parsing and pre-processing). Interestingly, the resourge usage for some tasks goes *down* with added FLM monitoring due to the additional stage usage.

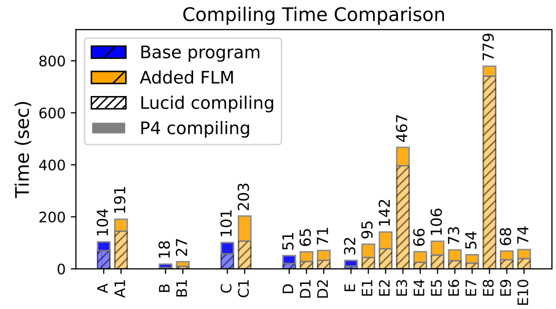
9.3 DFA Synthesis

We measured the size of each DFA in our examples, noted in the "DFA" column of Figure 6a as a pair $(|Q|, |\Sigma|)$ to show

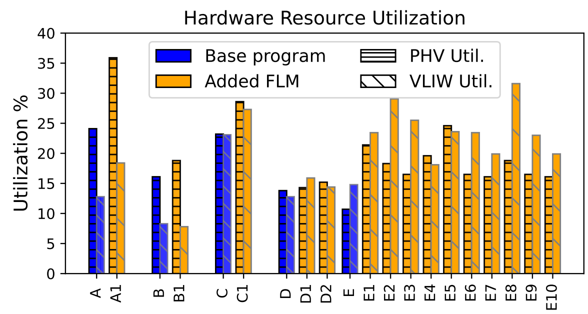
[1] https://github.com/aj3189/lucid

	Lines of code			DFA	FLM	Stages
	Lucid	IR	P4	(Q,Σ)	Time(s)	
A: Cuckoo Firewall [24]	171	388	1795	-	-	10
A1: Cuckoo Insertion	+14	+268	+616	(9,5)	7.6	+0
B: Stateful Firewall [24]	34	91	578	-	-	6
B1: Firewall Correctness	+10	+100	+308	(4,4)	.8	+4
C: SipHash [28]	185	260	2072	-	-	8
C1: SipHash Rounds	+30	+125	+981	(17,3)	19.9	+0
D: Chain Replication [30]	126	223	1319	-	-	10
D1: Sequence Numbers	+10	+75	+299	(4,2)	.5	+0
D2: Double Write	+11	+85	+336	(5,2)	1.3	+0
E: Simple RIP [24]	121	219	1158	-	-	8
E1: DHCP Anomaly [3]	+24	+160	+1184	(4,3)	.6	+0
E2: Fingerprint [23]	+41	+224	+2264	(10,8)	14.2	+2
E3: Port Knocking [10]	+19	+168	+1548	(6,16)	114.4	-1
E4: DNS TTL [21]	+15	+118	+830	(4,2)	.4	+2
E4: DNS Tunneling [21]	+18	+191	+1382	(13,5)	10.7	+1
E6: SwiSh Local [31]	+13	+147	+1123	(7,3)	1.0	+0
E7: NetChain [15]	+13	+89	+521	(4,2)	.4	+0
E8: Paxos Recovery [27]	+25	+279	+1048	(5,32)	15.7	+0
E9: ATP Sequence [18]	+23	+147	+668	(6,8)	6.0	+0
E10: ATP JobID [18]	+35	+140	+662	(19,3)	11.4	+0

(a) The results from compilation of our 15 example monitors. The gray rows show the base programs. The following white rows show one or more monitors that we added to the base programs. For the base programs, we show the lines of code of the source program (Lucid), the intermediate representation before compiling to P4 (IR), and the resulting P4 program (P4), as well as the number of stages used. For the monitoring tasks, we show the *additional* lines of code and stages added by FLM as well as the size of the DFA in terms of states (Q) and the alphabet (Σ) as well as the time required to compile FLM to Lucid.



(b) The compilation time of each example, with the base programs in blue and the monitors in orange.



(c) The PHV and VLIW utilization of each compiled program, with the base programs in vlue and the monitors in orange.

the number of states and the size of the alphabet. These varied from small automata with only a few states and symbols to ones with dozens of states and alphabet characters. The step of synthesizing an implementation, included in the "FLM Time", was relatively quick for all of them. In prior work [6], we further evaluated the difficulty of state machine synthesis.

We found that it was possible to generate implementations for state machines representing simple networking tasks such as tracking a TCP handshake. We also tested combinations of state machines via union (tracking multiple at once) or concatenation (tracking one after another) via video fingerprinting examples. While union was usually easier than concatenation, there were many examples of the same size DFA (as measured by states and alphabet) where some worked and some did not. There are other qualities of DFA complexity that show up only in some fingerprinting patterns. For example, states with many incoming or outgoing edges place many constraints on the same variables and make a solution more difficult to find.

Finally, we tried five different synthesis templates of varying complexity, ranging from simple assignment to using Tofino-specific tricks such as representing states with two integers in a paired array and updating both simultaneously. We also varied the choices of bitwise operators available in each template. We found that the template shown in Figure 5 was the best balance between expressiveness (ability to represent a wide variety of DFAs) and synthesis time.

10 Related Work

The syntax for FLM programs is inspired by Aragog [27], a system that focused on recognizing issues in distributed systems by specifying regex-like patterns. The patterns were checked by a *global* verifier that had specific events forwarded to it by all the systems. Aragog operates entirely in the control plane, whereas our work focuses on recognizing packet sequences appearing on a single data plane switch at line rate.

Many works use data plane switches to recognize regular expressions appearing in packets for the purposes of content inspection. Some early work appeared before P4 was released [20, 22]. More recent work has built on further hardware advances, and includes frameworks specialized for matching strings in a packet [14, 26]. DBVal [17] focuses on verifying the data plane execution of a single packet. DeepMatch [12] focuses on searching for regular expressions within the payload of packets, and developed some techniques to hold state between payloads for a single flow. Our work differs in that it focuses on patterns of *sequences* of packets, where all the computation happens in a single stage used once per packet, rather than a series of stages that can search for patterns *within* packet content. We also allow more expressive patterns with the binding of event values and predicates.

Our patterns draw inspiration from previous work on parametric verification [3], as well as studies of various forms of automata for wide-ranging applications. Timed automata [2] can record the timing of events and place time constraints on transitions. Symbolic automata [8, 9] permit a very wide variety of predicates on transitions without memory other than their state, while register automata [16] can record characters in registers, and check only equality. Recently, symbolic register automata [7] were proposed, which combines the two.

Our preliminary paper [6] presented an algorithm for synthesizing DFA transitions using SMT solvers. FLM extends that core with a new pattern-based language and compiler that separates bindings and predicates from classical regular expressions which are turned into DFAs.

11 Discussion

Limitations. One limitation for recognizing FLM patterns, and the reason for the rewrite rules, is the difficulty of correctly updating variables under the constraints on PISA switches. Other hardware or computation models might provide an easier path to computing the environment derivative (on a general purpose CPU with unlimited memory, it could just be computed directly), which would increase the number of FLM patterns that are recognizable. Still, we show in our evaluations that the subset of implementable patterns is expressive enough for many applications. An interesting research question for the future is to characterize which FLM patterns can be implemented under which computation models, and how best to do so while minimizing resource use.

Another limitation comes from the amount of computation available in a single stage, which dictates the size of implementable DFAs. Very long or complicated patterns could translate to DFAs that are too large to fit into one read-modify-write action. For more details, see the paragraph on synthesis hardness above. Our work is only platform-specific in that the template defined in Figure 5 is tailored to compilation on the Intel Tofino. Other hardware that has different computation available in a single stage would likely permit a more or less expressive template. A solution to this that applies for some DFAs is to carefully use more than one stage to implement it. For preliminary details about this approach, see the Appendix. As hardware improves, we hope to see both more computation available within a stage and more stages, alleviating this restriction from two angles.

Scalability and flexibility. In our examples, we used a variety of indexing functions to represent individual state machines, including per-flow, per-port, and per-MAC. However, in some networks, there is not enough memory available on current hardware to store this many values, risking index collisions. Other works [21] have dealt with this problem using complex data structures, sampling, or grouping flows to be considered together. While FLM does not employ any of these by default, it is flexible enough that a programmer could implement any of these techniques to compute an index before applying transitions to the FLM state machine array. Furthermore, if the high-level language is too restricting, a programmer can write code directly in the FLM intermediate representation, allowing them to intersperse arbitrary code for how the index is computed, which event is used to transition the state machine, and where in the control flow to transition.

Monitoring scope. We built our FLM compiler to target the Intel Tofino, but the core ideas are not reliant on any particular piece of hardware. The main requirement for an FLM program is a single pipeline with atomic updates and persistent memory that is updated per-packet. This applies to any switch implementing the PISA architecture.

We did not solve the problem of recognizing patterns in a distributed system of switches, instead focusing on how to properly compile to a single pipeline. A switch with multiple pipelines would implement multiple FLM monitors independently. For most monitoring tasks, properly configuring how ports map to pipelines (essentially slicing the index space across monitors) should preserve the monitor's reliability by sending packets intended for the same state machine through the same pipeline. We consider distributed monitoring an interesting task for future research.

12 Conclusion

We introduce FLM, a programming language that uses new abstractions to recognize and react to user-defined packet and event sequences at a switch. We develop a compilation procedure that transforms FLM programs into a series of match-action tables and register update functions, using a combination of rewrite rules and SMT-based program synthesis. Our evaluation using 15 sequence monitoring finds that FLM is flexible and supports line-rate processing on current networking hardware. This work raises no ethical concerns.

Acknowledgments

This material is based upon work supported by the National Science Foundation Grants CNS-2007073, FMiTF-2219862, and FMiTF-2219863. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8, 2013.

[2] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[3] Étienne André, Michał Knapik, Didier Lime, Wojciech Penczek, and Laure Petrucci. *Parametric Verification: An Introduction*, pages 64–100. Springer Berlin Heidelberg, Berlin, Heidelberg, 2019.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, jul 2014.

[5] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *ACM SIGCOMM 2013*, pages 99–110, 2013.

[6] Xiaoqi Chen, Andrew Johnson, Mengying Pan, and David Walker. Synthesizing state machines for data planes. In *Proceedings of the Symposium on SDN Research*, SOSR '22, page 81–88, New York, NY, USA, 2022. Association for Computing Machinery.

[7] Loris D'Antoni, Tiago Ferreira, Matteo Sammartino, and Alexandra Silva. Symbolic register automata. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification*, pages 3–21, Cham, 2019. Springer International Publishing.

[8] Loris D'Antoni, Zachary Kincaid, and Fang Wang. A symbolic decision procedure for symbolic alternating finite automata. *Electronic Notes in Theoretical Computer Science*, 336, 10 2016.

[9] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In Rupak Majumdar and Viktor Kunčak, editors, *Computer Aided Verification*, pages 47–67, Cham, 2017. Springer International Publishing.

[10] Rennie Degraaf, John Aycock, and Michael Jacobson. Improved port knocking with strong authentication. In *21st Annual Computer Security Applications Conference (ACSAC'05)*, pages 10–pp. IEEE, 2005.

[11] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '18, page 357–371, New York, NY, USA, 2018. Association for Computing Machinery.

[12] Joel Hypolite, John Sonchack, Shlomo Hershkop, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Deepmatch: Practical deep packet inspection in the data plane using network processors. In *Proceedings of the 16th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '20, page 336–350, New York, NY, USA, 2020. Association for Computing Machinery.

[13] Intel. Intel tofino series programmable ethernet switch asic. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html, 2022.

[14] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. Fast string searching on PISA. In *Proceedings of the 2019 ACM Symposium on SDN Research (SOSR 19)*, pages 21–28, 2019.

[15] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 35–49, 2018.

[16] Michael Kaminski and Nissim Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329–363, 1994.

[17] K Shiv Kumar, Ranjitha K, P S Prashanth, Mina Tahmasbi Arashloo, Venkanna U., and Praveen Tammana. Dbval: Validating p4 data plane runtime behavior. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, SOSR '21, page 122–134, New York, NY, USA, 2021. Association for Computing Machinery.

[18] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. Atp: In-network aggregation for multi-tenant learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 741–761, 2021.

[19] Devon Loehr and David Walker. Safe, modular packet pipeline programming. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[20] Chad R. Meiners, Jignesh Patel, Eric Norige, Eric Torng, and Alex X. Liu. Fast regular expression matching using small tcams for network intrusion detection and prevention systems. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, page 8, USA, 2010. USENIX Association.

[21] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 85–98, New York, NY, USA, 2017. Association for Computing Machinery.

[22] Derek Chi-Wai Pao and Xing Wang. Multi-stride string searching for high-speed content inspection. *Comput. J.*, 55:1216–1231, 2012.

[23] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Beauty and the burst: Remote identification of encrypted video streams. In *USENIX Security Symposium*, pages 1357–1374, 2017.

[24] John Sonchack, Devon Loehr, Jennifer Rexford, and David Walker. Lucid: A language for control in the data plane. In *ACM SIGCOMM 2021*, pages 731–747, 2021.

[25] The P4 Language Consortium. $P4_{16}$ language specification. https://p4.org/p4-spec/docs/P4-16-v1.1.0-spec.html, November 2018.

[26] Shicheng Wang, Menghao Zhang, Guanyu Li, Chang Liu, Zhiliang Wang, Ying Liu, and Mingwei Xu. Bolt: Scalable and cost-efficient multistring pattern matching with programmable switches. *IEEE/ACM Transactions on Networking*, pages 1–16, 2022.

[27] Nofel Yaseen, Behnaz Arzani, Ryan Beckett, Selim Ciraci, and Vincent Liu. Aragog: Scalable runtime verification of shardable networked systems. In *Operating systems and implementations (OSDI)*, October 2020.

[28] Sophia Yoo and Xiaoqi Chen. Secure keyed hashing on programmable switches. In *Proceedings of the ACM SIGCOMM 2021 Workshop on Secure Programmable network INfrastructure*, pages 16–22, 2021.

[29] z3. z3 solver, publisher = GitHub, howpublished = https://github.com/z3prover/z3/, urldate=2022-06-28, year=2022.

[30] Lior Zeno, Dan R. K. Ports, Jacob Nelson, and Mark Silberstein. Swishmem: Distributed shared state abstractions for programmable switches. In *Proceedings of the 19th ACM Workshop on Hot Topics in Networks*, HotNets '20, page 160–167, New York, NY, USA, 2020. Association for Computing Machinery.

[31] Lior Zeno, Dan RK Ports, Jacob Nelson, Daehyeok Kim, Shir Landau-Feibish, Idit Keidar, Arik Rinberg, Alon Rashelbach, Igor De-Paula, and Mark Silberstein. {SwiSh}: Distributed shared state abstractions for programmable switches. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 171–191, 2022.

13 Appendix

13.1 Translating patterns

The full definition of translating patterns uses *BigOr*, which translates an individual event and predicate.

Definition 6. $BigOr(b\langle p\rangle, P) =$
$\bigvee(\{b\} \circ \{bin(i) \mid i < 2^{len(P)}$ *and the bit representing* p *is* $1\})$

$BigOr(b\langle p_1\rangle, [p_1, p_2]) = (b10 + b11)$, as above. Similarly, $BigOr(b\langle p_2\rangle, [p_1, p_2]) = (b01 + b11)$. The other pattern translations are defined recursively. All of the constructors $(+, \&, *, .)$ stay the same, as they have similar semantics for classical regular expressions and patterns, and we translate all of the event patterns with *BigOr*.

Definition 7. *The FLM pattern translation* $T_{re}(r, P)$*:*

$$
\begin{array}{rcl}
T_{re}(\varepsilon, P) & = & \varepsilon \\
T_{re}(\emptyset, P) & = & \emptyset \\
T_{re}(a\langle p_i\rangle, P) & = & BigOr(a\langle p_i\rangle, P) \\
T_{re}(s_1.s_2) & = & T_{re}(s_1, P).T_{re}(s_2, P) \\
T_{re}(s_1 + s_2) & = & T_{re}(s_1, P) + T_{re}(s_2, P) \\
T_{re}(s_1 \,\&\, s_2) & = & T_{re}(s_1, P) \,\&\, T_{re}(s_2, P) \\
T_{re}(s^*) & = & (T_{re}(s, P))^*
\end{array}
$$

13.2 Implementability

The following is a list of the rewrite rules we use:

- $b\langle @y; p\rangle.r \xrightarrow{rw} b\langle @y\rangle \triangleright b\langle p\rangle.r$. This rule always applies, and is how we introduce the $\triangleright$ operator. It splits a binding into two parts, and removes it from the expression.
- $r.(b\langle @y\rangle \triangleright s) \xrightarrow{rw} b\langle @y\rangle \triangleright (r.s)$. This rule only applies if b and y do not appear in r. This ensures that no word in r contains an event b, mirroring the semantics of $\triangleright$.
- $(b\langle @y_1\rangle \triangleright s_1) + (b\langle @y_2\rangle \triangleright s_2) \xrightarrow{rw} b\langle y_1\rangle \triangleright (s_1 + [y_1/y_2]s_2)$. Here $[y_1/y_2]s_2$ denotes the capture-avoiding substitution of y_1 for y_2 in s_2. This rule only applies when s_2 contains no references to y_1. An equivalent rule applies for &.
- $(a\langle @y_1\rangle \triangleright s_1) + (b\langle @y_2\rangle \triangleright s_2) \xrightarrow{rw} a\langle @y_1\rangle \triangleright b\langle @y_2\rangle \triangleright (s_1 + s_2)$. This rule applies if s_1 contains no occurrences of y_2, s_2 contains no occurrences of y_1, and $a \neq b$. An equivalent rule applies for &.
- $(b\langle @y\rangle \triangleright s_1) + s_2 \xrightarrow{rw} b\langle @y\rangle \triangleright (s_1 + s_2)$. This rule applies if s_2 contains no occurrences of y. An equivalent rule applies for &.

These contain side conditions that ensure rewritten expressions have the same semantics after rewriting. For example, we check that variables are not contained in out-of-scope expressions before hoisting bindings to an outer scope. However, they also preserve a key property of FLM patterns that can be written *without* the new binding form that uses $\triangleright$. We call this property *implementability*. First, we define a shorthand for an environment defined over the variables in a list of bindings.

Definition 8. *An environment* G *is compatible with a binding list* B *if* $Dom(G) = Range(B)$*. That is,* G *contains one assignment for each variable in* B*. We will use the metavariable* G *for compatible environments to differentiate from general environments denoted by* E*.*

For example, the environment $(y_1 \leftarrow 1, y_2 \leftarrow 15)$ is compatible with $b_1\langle @y_1\rangle \triangleright b_2\langle @y_2\rangle$. We will write $Asgn0(B)$ to mean the environment that assigns every variable in B to 0. Next, we define implementability, which intuitively means that the pattern derivatives are not affected by assignments to variables in a binding list for event types that do not appear in the bindings.

Definition 9. *An FLM pattern in prefix form* $B \triangleright s$ *is implementable if and only if for any event type* $a \notin B$*, any integer* z*, any environment* E *where* $B \triangleright s$ *is closed under* E*, and any two environments* G_1 *and* G_2 *which are compatible with* B*:*

$$[\![D_{re}(a\langle z\rangle; s; E, G_1)]\!]_{E,G_2} = [\![D_{re}(a\langle z\rangle; s; E, G_2)]\!]_{E,G_2}$$

All of the patterns we introduce in the main paper are implementable. An example pattern which does not have this property is:

$$b\langle @y\rangle \triangleright a\langle \lambda x.(x == y)\rangle .b\langle \lambda x.true\rangle$$

This pattern is semantically well-defined: it is the set of events of type a followed by b with equal parameters. However, the value of y is not known when the event a appears, and so it cannot be implemented at line-rate (in general, this type of pattern would require some look-back capability). This fails the implementability property because the assignment to y will change the result of D_{re} for events of type a. We show that our rewrite rules preserve this property: if $r \xrightarrow{rw} r'$ and r is implementable, then so is r'.

Now, we define a theorem that captures the semantic meaning of *both* the binding and pattern derivatives. Intuitively, it is similar to theorem 1, but uses an environment produced by D_{bind} instead of a constant one.

Theorem 3. $\forall B, s, a, z, E$: *if $B \triangleright s$ is in prefix form, closed under E, and implementable, then:*

$$[\![D_{re}(a\langle z\rangle; s; E', Asgn0(B'))]\!]_{E', Asgn0(B')} = \{w | a\langle z\rangle .w \in [\![B \triangleright s]\!]_E\}$$

Where $B', E' = D_{bind}(a\langle z\rangle; B; E)$

As an example, consider the one from section 5, where $p_1 = \lambda x.x \geq 10$ and $p_2 = \lambda x.x == y$:

$$b\langle @y\rangle \triangleright (b\langle p_1\rangle .b\langle p_2\rangle)$$

Starting with an empty environment and an event $b\langle 12\rangle$, $D_{bind}(b\langle 12\rangle; b\langle @y\rangle; .) = (.; (y \leftarrow 12))$. Taking the pattern derivative using the new environment:

$$D_{re}(b\langle 12\rangle; b\langle \lambda x.x \geq 10\rangle .b\langle p_1\rangle; y \leftarrow 12) = b\langle p_2\rangle$$

The semantics of this remaining pattern when $y \leftarrow 12$ contains just $b\langle 12\rangle$, which is correct according to the theorem: the only string starting with $b\langle 12\rangle$ in $[\![b\langle p_1\rangle .b\langle p_2\rangle]\!]_{y \leftarrow 12}$ is $b\langle 12\rangle b\langle 12\rangle$.

13.3 Example patterns

Below is a full list of the example patterns that were used for evaluation. They range from simple checks to more complicated patterns about high-level protocols.

A1 Cuckoo Firewall [24]

```
(ip_pkt(@int saved_src=src, @int saved_dst=dst)
        .(((cuckoo_insert(fst_src==saved_src) && cuckoo_insert(fst_dst==saved_dst))*
              .(cuckoo_insert(!(fst_src==saved_src)) || cuckoo_insert(!(fst_dst==saved_dst)))
            )||((cuckoo_insert(fst_src==saved_src) && cuckoo_insert(fst_dst==saved_dst))
            .(cuckoo_insert(fst_src==saved_src) && cuckoo_insert(fst_dst==saved_dst))
            .(cuckoo_insert(fst_src==saved_src) && cuckoo_insert(fst_dst==saved_dst))
            .(cuckoo_insert(fst_src==saved_src) && cuckoo_insert(fst_dst==saved_dst)))))
```

This checks whether the cuckoo firewall insertion algorithm is working properly.

B1 Stateful FW Timeout [24]

```
ip_pkt (@int start_time = Sys.time(); ip#tos == TOS_TRUSTED)
    .(ip_pkt(ip#tos == TOS_TRUSTED)
        || ((ip_pkt(!(ip#tos == TOS_TRUSTED))
            && ip_pkt(Sys.time() - start_time < 10000))))*
    .((ip_pkt(!(ip#tos == TOS_TRUSTED))
            && ip_pkt(!(Sys.time() - start_time < 10000))))
```

This is a general specification of firewall correctness. It checks that packets from inside (`TOS_TRUSTED`) are allowed out, and that return packets are not allowed back in past the timeout threshold.

C1 SipHash [28]

```
iptcp_to_server_syn
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
```

```
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.siphash_intermediate
.iptcp_craft_out_dummy
```

This checks whether or not the siphash implementation is completing the proper number of hashing rounds.

D1 Chain replication 1 [30]

```
write(@int saved_seq=seq)
.write(seq<saved_seq)
```

This checks whether there are write events with sequence numbers out of order.

D2 Chain replication 2 [30]

```
write()
.write()
.ack()
```

This checks whether there are two write events to the same index before the first one is `ACK`ed.

E1 DHCP Anomaly (section 3)

```
IP_Pkt*
.DHCP_Ack(@int assigned = cip)
.(IP_Pkt(sip == assigned))*
.IP_Pkt(sip != assigned)
```

This is the example from the paper.

E2 Fingerprint ([23])

```
S1.S5.S1.S4.S7.S8.S1.S2
```

This represents one example video fingerprint. `S1` - `S8` denote different packet sizes. The fingerprint is 8 packets in sequence, with the denoted sizes.

E3 Port Knocking (len=4) [10]

```
ip_in(dport==1234)
.ip_in(dport==5678)
.ip_in(dport==9012)
.ip_in(dport==3456)
```

This pattern represents an example port knocking sequence. `dport` is the destination port of a packet.

E4 DNS TTL Change Count [21]

```
 DNS_packet_fwd(@int<<32>> fst_ttl = ttl)
.(DNS_packet_fwd(fst_ttl != ttl))
```

This checks whether the ttl of packets changes when using DNS by recording the first in a flow and checking the second. The intent is to count the number of TTL changes.

E5 DNS Tunneling [21]

```
DNS_resp(@int d1=dip, @int d2=dip2)
.(  (ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
    .(ip_packet(dip!=d1) && ip_packet(dip!=d2))
```

This checks that, upon receiving a DNS response, the receiver goes on to contact the requested IP. Tunneling is suspected if the receiver of a DNS response never uses the information.

E6 SwiSh Local View [31]

```
(S1||S2||S3)*
.( (S2.S1)||
   (S3.S2)||
   (S1.S3))
```

This denotes any of the switches in a chain of 3 SwiShMem switches forwarding an update to the wrong neighbor.

E7 NetChain [15]

```
NetChainUpdate(@int v=version)
.NetChainUpdate(version < v)
```

This shows an algorithm running improperly, as detected by having an earlier version after a newer one.

E8 Paxos Recovery [27]

```
Paxos(@int a=l1,@int c=l2,@int e=l3; ty==RECOVER).
(
    (Paxos(l1<a) && Paxos(ty==RECOVER)) ||
    (Paxos(l2<c) && Paxos(ty==RECOVER)) ||
    (Paxos(l3<e) && Paxos(ty==RECOVER))
).
Paxos(ty==RECOVERED)
```

This shows the sequence of exchanges of a Paxos recovery.

E9 ATP sequence [18]

```
ATP_add(@int x = cnt)
.ATP_add(cnt==x+2)
.ATP_add(cnt==x+3)
.ATP_add(cnt==x+4)
```

This shows a sequence of 4 consecutive add events with an increasing count variable.

E10 ATP JobID [18]

```
ATP_add(@int saved_jobid=jobid)
.(ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
```

```
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_add(jobid==saved_jobid)
.ATP_release(jobid==saved_jobid))
```

This shows a sequence of 16 consecutive add events with the same job ID.

13.4 Extensions

In this section, we go over various extensions to FLM to improve its expressiveness and usability.

13.4.1 Maintenance events

Some patterns, such as those that wait for a timeout, can be difficult to express as a sequence. Consider a sequence to check that received requests are replied to with decisions in a timely manner:

```
request(@t1 = Sys.time())
.request*
.decision(Sys.time() - t1 > Threshold)
```

Here, we are ignoring the indexing and response code to focus on the pattern. This pattern seems reasonable to detect *late* decisions, but what if a decision never comes? Intuitively, that should be a violation as well, but if there is never another packet for this flow, one will never be reported. The problem is that examples such as timeouts are examining a liveness property. Our solution to this is to add *maintenance events*, which do not represent incoming network events. Instead, they are guaranteed to visit every index of an array of FLM patterns eventually. To fix the above example, we can write the following, using `maintenance` for the new events.

```
maintenance*
.request(@t1 = Sys.time())
.(maintenance(Sys.time() - t1 <= Threshold) + request)*
.(decision(Sys.time() - t1 > Threshold) +
(maintenance(Sys.time() - t1 > Threshold)))
```

The placement of maintenance events will not interfere with detecting late decisions, but it will now also detect decisions that never arrive because the maintenance event will eventually come along and match the last disjunction. These can be implemented easily with a packet that repeatedly circulates through the array one index at a time, perhaps with a delay to reduce overhead.

13.4.2 Longer patterns

Some sequence monitors are too complicated to be expressed using the original syntax, causing the compiler to reject them. This could be for one of two reasons:

1. The pattern is unable to be rewritten into prefix form, usually because the programmer desires to bind variables using the same event type in two places. This would violate the rewrite rule for concatenation.

2. The pattern is able to be rewritten successfully, but an implementation of the transition function for the translated DFA cannot be found. The synthesis algorithm might either time out, or return "unsat."

The solution to both of these is to pay more stages for the ability to implement a pattern. To do so, we introduce *unambiguous concatenation*, which allows a pattern to be split into two parts that can be implemented sequentially.

Unambiguous concatenation We say that two FLM patterns in prefix form are *unambiguously concatenated* with a semantic condition that allows us to split it across stages. First, we define the *prefixes* of a pattern, which are all the prefixes of any accepted word:

Definition 10. *The prefixes of an FLM pattern r, denoted prefix(r), is the set:*

$$\{u \mid \exists E, v \text{ such that } u.v \in [\![r]\!]_E\}$$

Next, we define the *continuations* of a pattern, which are all the words which can be appended to an accepted word to get another accepted word:

Definition 11. *The Continuations of an FLM pattern r, denoted continuation(r) is the set:*

$$\{v \mid \exists E, u, w \text{ such that } u \in [\![r]\!]_E \text{ and } u.v.w \in [\![r]\!]_E\}$$

As a simple example with the pattern $a.b^*$, the prefixes and continutations can be defined with the languages of the following patterns:

$$prefix(a.b^*) = \varepsilon + a.b^*$$

$$continuation(a.b^*) = b^*$$

Finally, we say that two patterns r_1 and r_2 are unambiguously concatenated, denoted $r_1!!r_2$, if the prefixes of r_1 and the continuations of r_2 only intresect with ε, or more formally:

$$r_1!!r_2 \iff (prefix(r_1) \cap continuation(r_2)) \setminus \{\varepsilon\} = \emptyset$$

Using the above example, $a.b^*!!a.b^*$ is a valid unambiguous concatenation. ε is excluded because it is always both a prefix and continuation of any non-empty language.

Implementation The unambiguous concatenation condition lets us compile a single pattern into multiple stages, which will either allow a programmer to reuse events for variable bindings or compile a larger pattern to switch actions. In principle, there could be many patterns connected with !!. We assume that this is at the top level, and call each concatenated pattern a *section*. We compile a series of sections in three steps, after compiling each section individually:

1. For each DFA except the last section, add one new state, called "done." Add a self-loop for every character to "done." For each transition from any accepting state to the rejecting state, replace it with a transition to "done."

2. For each section's code except the first, add a clause to only run it if the previous section's state was "done."

3. In each section's DFA, if any later sections are not nullable, change all of its accepting states to non-accepting ones. The `transition` statement returns whether or not the last section run ended in an accepting state.

Step one allows us to track when each section has finished matching characters. This is the step where the unambiguous concatenation condition is important. The transitions from accepting states to non-reject states correspond to the *continuations* of a section, while the transitions from the initial state correspond to its *prefixes*. The condition guarantees that the prefixes of a later section do not coincide with the continuations of an earlier one, so we never miss a transition.

Step two ensures that we are running the patterns in sequence, not in parallel. Note that if there are new variables to be bound, they are only bound *after* previous sections are "done."

Step three ensures that the series of sections only accepts when the current string matches the unambiguously concatenated pattern. We leave accepting states if all later states are nullable because otherwise, we would miss some strings that only match the earlier sections.

13.5 Proofs of Theorems

Theorem 4. *"Derivatives commute":*
$\forall a, z, s, E, P$ where s is binding free, both s and each predicate in P is closed under E, and $preds(s) \subseteq P$:

$$D_{clas}(T_l(a\langle z\rangle, E, P); T_{re}(s, P)) = T_{re}(D_{re}(a\langle z\rangle; s; E), P)$$

Proof. By induction on the structure of s □

Theorem 5. *For all a, z, s, E where s is binding-free and s is closed under E:*

$$[\![D_{re}(a\langle z\rangle; s; E)]\!]_E = \{w' \mid a\langle z\rangle.w' \in [\![s]\!]_E\}$$

Proof. Proof by induction on the structure of s. Base cases:

1. $\emptyset$:

$$[\![D_{re}(a\langle z\rangle;\emptyset;E)]\!]_E$$
$$= [\![\emptyset]\!]_E = \emptyset = \{w|a\langle z\rangle.w \in \emptyset\}$$

2. ε:

$$[\![D_{re}(a\langle z\rangle;\varepsilon;E)]\!]_E$$
$$= [\![\emptyset]\!]_E = \emptyset = \{w|a\langle z\rangle.w \in \{\varepsilon\}\}$$

3. $a\langle p\rangle : a\langle z\rangle \in [\![a\langle p\rangle]\!]_E$ iff $[\![p(z)]\!]_E$. So, $\{w|a\langle z\rangle.w \in [a\langle z\rangle]_E\} = \{\varepsilon\}$ if $[\![p(z)]\!]_E$ and $\emptyset$ else, which is the derivative.

Induction cases:

1. r + s:

$$[\![D_{re}(a\langle z\rangle;r+s;E)]\!]_E$$
$$= [\![D_{re}(a\langle z\rangle;r;E)]\!]_E \,\cup\, [\![D_{re}(a\langle z\rangle;s;E)]\!]_E$$

By induction:

$$= \{w|a\langle z\rangle.w \in [\![r]\!]_E\} \,\cup\, \{w|a\langle z\rangle.w \in [\![s]\!]_E\}$$
$$= \{w|a\langle z\rangle.w \in [\![r]\!]_E \cup [\![s]\!]_E\}$$
$$= \{w|a\langle z\rangle.w \in [\![r+s]\!]_E\}$$

2. r & s: Very similar to above, substituting & and $\cap$ for $+$ and $\cup$.

3. s^*:
The following together show that $[\![D_{re}(a\langle z\rangle;s^*;E)]\!]_E = \{w|a\langle z\rangle.w \in [\![s^*]\!]_E\}$

   (a) $\{w|a\langle z\rangle.w \in [\![s^*]\!]_E\} \subseteq [\![D_{re}(a\langle z\rangle;s^*;E)]\!]_E$:
   If $a\langle z\rangle.w \in [\![s^*]\!]_E$, then $a\langle z\rangle.w \in [\![s^i]\!]_E$ for some smallest natural i. i cannot be 0. Since $a\langle z\rangle.w \in [\![s^i]\!]_E = [\![s.s^{i-1}]\!]_E$, $a\langle z\rangle.w = w_1.w_2$ s.t. $w_1 \in [\![s]\!]_E \wedge w_2 \in [\![s^{i-1}]\!]_E$. s_1 cannot be ε, otherwise i could be smaller. So, $w_1 = a\langle z\rangle.w_1'$ and $w = w_1'.w_2$.
   By induction,

   $$[\![D_{re}(a\langle z\rangle;s;E)]\!]_E = \{w|a\langle z\rangle.w \in [\![s]\!]_E\}$$

   So, $w_1' \in [\![D_{re}(a\langle z\rangle;s;E)]\!]_E$. $w_2 \in [\![s^*]\!]_E$, so $w \in [\![D_{re}(a\langle z\rangle;s;E).s^*]\!]_E = [\![D_{re}(a\langle z\rangle;s^*;E)]\!]_E$

   (b) $[\![D_{re}(a\langle z\rangle;s^*;E)]\!]_E \subseteq \{w|a\langle z\rangle.w \in [\![s^*]\!]_E\}$:
   If $w \in [\![D_{re}(a\langle z\rangle;s^*;E)]\!]_E$, then: $w = s_1.s_2$, where:

   $$s_1 \in [\![D_{re}(a\langle z\rangle;s;E)]\!]_E \text{ and } s_2 \in [\![s^*]\!]_E$$

   By induction, $a\langle z\rangle.s_1 \in [\![s]\!]_E$. So, $s_1.s_2 \in [\![s^*]\!]_E$

4. $r.s$:
The following together show that $[\![D_{re}(a\langle z\rangle;r.s;E)]\!]_E = \{w|a\langle z\rangle.w \in [\![r.s]\!]_E\}$

   (a) $\{w|a\langle z\rangle.w \in [\![r.s]\!]_E\} \subseteq [\![D_{re}(a\langle z\rangle;r.s;E)]\!]_E$:
   If a word $a\langle z\rangle.w \in [\![r.s]\!]_E$, then by definition $a\langle z\rangle.w = w_1.w_2$ s.t. $w_1 \in [\![r]\!]_E \wedge w_2 \in [\![s]\!]_E$. By definition, $[\![D_{re}(a\langle z\rangle;r.s;E)]\!]_E = [\![D_{re}(a\langle z\rangle;r;E).s]\!]_E \cup [\![v(r).D_{re}(a\langle z\rangle;s;E)]\!]_E$. There are two cases

      i. $w_1 = a\langle z\rangle.w_1'$. Then by induction, $w_1' \in [\![D_{re}(a\langle z\rangle;r;E)]\!]_E$, so $w_1'.w_2 \in [\![D_{re}(a\langle z\rangle;r;E).s]\!]_E$
      ii. $w_1 = \varepsilon$, and $w_2 = a\langle z\rangle.w_2'$. Since $w_1 = \varepsilon \in [\![r]\!]_E$, $v(r) = \varepsilon$. By induction, $w_2' \in [\![D_{re}(a\langle z\rangle;s;E)]\!]_E$, so $w \in [\![v(r).D_{re}(a\langle z\rangle;s;E)]\!]_E$

   This includes all possibilities for any word in $\{w|a\langle z\rangle.w \in [\![r.s]\!]_E\}$.

(b) $[\![D_{re}(a\langle z\rangle;r.s;E)]\!]_E \subseteq \{w|a\langle z\rangle.w \in [\![r.s]\!]_E\}$:
If $w \in [\![D_{re}(a\langle z\rangle;r.s;E)]\!]_{D_{bind}(a\langle z\rangle;r.s;E)}$, then either:

i. $w \in [\![D_{re}(a\langle z\rangle;r;E).s]\!]_E$. By induction, $w = w_1'.w_2$ such that $a\langle z\rangle.w_1' \in [\![r]\!]_E$ and $w_2 \in [\![s]\!]_E$. So, $a\langle z\rangle.w \in [\![r.s]\!]_E$.

ii. $w \in [\![v(r).D_{re}(a\langle z\rangle;s;E)]\!]_E$.Then either $v(r) = \emptyset$ and there are no such w, or $v(r) = \varepsilon$ and by induction, $a\langle z\rangle.w \in [\![s]\!]_E$. Since r is nullable, $[\![s]\!]_E \subseteq [\![r.s]\!]_E$.

This includes all possibilities for any word in $[\![D_{re}(a\langle z\rangle;r.s;E)]\!]_E$

□

Lemma 1. *For any $B \triangleright b_1\langle @y_1\rangle \triangleright b_2\langle @y_2\rangle \triangleright B' \triangleright s$ in prefix form and any E it is closed under:*

$$[\![B \triangleright b_1\langle @y_1\rangle \triangleright b_2\langle @y_2\rangle \triangleright B' \triangleright s]\!]_E = [\![B \triangleright b_2\langle @y_2\rangle \triangleright b_1\langle @y_1\rangle \triangleright B' \triangleright s]\!]_E$$

Proof. By induction on the length of B preceding the two to be exchanged. In the base case, expand the definitions twice. In the induction case, just exchange the smaller list. □

Lemma 2. *If $b\langle @y\rangle \triangleright B \triangleright s$ is implementable, then so is $B \triangleright s$.*

Proof. By expanding the semantics to get some value for y, and then using impl property on the compatible environments with B. □

Lemma 3. *If $B \triangleright s$ is implementable, then for any $a \notin B$,z, E where $B \triangleright s$ is closed, and compatible environment G, $B \triangleright D_{re}(a\langle z\rangle;s;(E,G))$ is implementable as well.*

Proof. By induction on the structure of s. □

Theorem 6. *Restatement of Theorem 3: For all a,z,B,s,E where $B \triangleright s$ is in prefix form (B is a non-redundant list of bindings and s is binding-free) and implementable:*

$$[\![B' \triangleright D_{re}(a\langle z\rangle;s;E',Asgn0(B'))]\!]_{E',Asgn0(B')} = \{w'|a\langle z\rangle.w' \in [\![B \triangleright s]\!]_E\}$$

where $B',E' = D_{bind}(a\langle z\rangle,B,E)$

Proof. By induction on the number of bindings in B. When B has no bindings, just use 1. When B has a binding with event b, proceed by cases on whether $a \in B$.

1. If $a \in B$, then by 1, we can assume without loss of generality that it is the first binding in B (i.e. $B = a\langle @y\rangle \triangleright B' \triangleright s$. Also, since B is not redundant, $D_{bind}(a\langle z\rangle;B';E,y \leftarrow z) = (B',E,y \leftarrow z)$. Then $D_{bind}(a\langle z\rangle;B;E) = (B',(E,y \leftarrow z))$. Next, we can expand the definition of $[\![B \triangleright s]\!]_E$:

$$[\![a\langle @y\rangle \triangleright B' \triangleright s]\!]_E = \underbrace{\{w_1.a\langle z'\rangle.w_2 \in [\![B' \triangleright s]\!]_{E,y\leftarrow z'} \text{ and } a \notin w_1\}}_{\textbf{S1}} \cup \underbrace{\{w|\forall z'.w \in [\![B' \triangleright s]\!]_{E,y\leftarrow z'} \text{ and } a \notin w\}}_{\textbf{S2}}$$

There are clearly no $a\langle z\rangle.w \in \textbf{S2}$, since they cannot contain the event a. Furthermore, if any word $a\langle z\rangle.w = w_1.a\langle z\rangle.w_2 \in \textbf{S1}$, w_1 must be ε. So:

$$\begin{aligned}&\{w'|a\langle z\rangle.w' \in [\![a\langle @y\rangle \triangleright B' \triangleright s]\!]_E\}\\ &= \{w'|a\langle z\rangle.w' \in \textbf{S1}\}\\ &= \{w'|a\langle z\rangle.w' \in [\![B' \triangleright s]\!]_{E,y\leftarrow z}\}\end{aligned}$$

We now want to show that this set is equal to $[\![B' \triangleright D_{re}(a\langle z\rangle;s;E,y \leftarrow z,Asgn0(B'))]\!]_{E,y\leftarrow z,Asgn0(B')}$, which is true directly by the induction hypothesis. :

$$\begin{aligned}&[\![B' \triangleright D_{re}(a\langle z\rangle;s;E,y \leftarrow z,Asgn0(B'))]\!]_{E,y\leftarrow z,Asgn0(B')}\\ &= \{w'|a\langle z\rangle.w' \in [\![B' \triangleright s]\!]_{E,y\leftarrow z}\} \quad \text{By induction}\end{aligned}$$

2. If $a \notin B$, then $B = b\langle @y\rangle \triangleright B' \triangleright s$. $D_{bind}(a\langle z\rangle; B; E) = (B, E)$, since a doesn't appear in the bindings. So, we can expand the definition of $[\![B \triangleright s]\!]_E$::

$$[\![b\langle @y\rangle \triangleright B' \triangleright s]\!]_E = \underbrace{\{w_1.b\langle z'\rangle.w_2 \in [\![B' \triangleright s]\!]_{E,y\leftarrow z'} \text{ and } b \notin w_1\}}_{\textbf{S1}} \cup \underbrace{\{w|\forall z'.w \in [\![B' \triangleright s]\!]_{E,y\leftarrow z'} \text{ and } b \notin w\}}_{\textbf{S2}}$$

Next, we can expand the definition of $[\![b\langle @y\rangle \triangleright B' \triangleright D_{re}(a\langle z\rangle; s; E, Asgn0(b\langle @y\rangle \triangleright B'))]\!]_{E,Asgn0(b\langle @y\rangle \triangleright B')}$:

$$\begin{gathered}[\![b\langle @y\rangle \triangleright B' \triangleright D_{re}(a\langle z\rangle; s; E, Asgn0(b\langle @y\rangle \triangleright B'))]\!]_{E,Asgn0(b\langle @y\rangle \triangleright B')} \\ = [\![b\langle @y\rangle \triangleright B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow 0, Asgn0(B'))]\!]_{E,y\leftarrow 0,Asgn0(B')} \\ = \underbrace{\{w_1.b\langle z'\rangle.w_2 \in [\![B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow 0, Asgn0(B'))]\!]_{E,y\leftarrow z',Asgn0(B')} \text{ and } b \notin w_1\}}_{\textbf{S3}} \\ \cup \underbrace{\{w|\forall z'.w \in [\![B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow 0, Asgn0(B'))]\!]_{E,y\leftarrow z',Asgn0(B')}\}}_{\textbf{S4}}\end{gathered}$$

Because $b\langle @y\rangle \triangleright B' \triangleright s$ is implementable, so is $B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow 0, Asgn0(B'))$ by using 2 and 3. Using this, we have the following in both **S3** and **S4**:

$$[\![B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow 0, Asgn0(B'))]\!]_{E,y\leftarrow z',Asgn0(B')}$$
$$= [\![B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow z', Asgn0(B'))]\!]_{E,y\leftarrow z',Asgn0(B')}$$

This lets us use the induction hypothesis on **S3** and **S4**, since $B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow 0, Asgn0(B'))$ is implementable by 2 and 3:

$$[\![B' \triangleright D_{re}(a\langle z\rangle; s; E, y \leftarrow z', Asgn0(B'))]\!]_{E,y\leftarrow z',Asgn0(B')} = \{w|a\langle z\rangle.w \in [\![B' \triangleright s]\!]_{E,y\leftarrow z'}\}$$

Plugging this into the definitions above gives the following:

$$\begin{gathered}[\![b\langle @y\rangle \triangleright B' \triangleright D_{re}(a\langle z\rangle; s; E, Asgn0(b\langle @y\rangle \triangleright B'))]\!]_{E,Asgn0(b\langle @y\rangle \triangleright B')} \\ = \underbrace{\{w' = w_1.b\langle z\rangle.w_2 | a\langle z\rangle.w' \in [\![B' \triangleright s]\!]_{E,y\leftarrow z'} \text{ and } b \notin w_1\}}_{\textbf{S3'}} \\ \cup \underbrace{\{w'|\forall z'.a\langle z\rangle.w' \in [\![B' \triangleright s]\!]_{E,y\leftarrow z'} \text{ and } b \notin w\}}_{\textbf{S4'}}\end{gathered}$$

This concludes the proof, since these two sets are the same as **S1** and **S2**, except for the requirement that the words start with $a\langle z\rangle$. Finally, it is clear that b is not in a word $a\langle z\rangle.w$ if b is not in w.

□

Lemma 4. *For all a, z, E, and B:*

$$D_{bind}(a\langle z\rangle; B; E, Asgn0(B)) = B', (E', Asgn0(B')$$

Where $B', E' = D_{bind}(a\langle z\rangle; B; E)$

Proof. Observe that the initial values in E do not matter for D_{bind}. If $a \in B$, then afterwards it is not in B, and each variable that was paired with a now has a value in E'. These values would have overwritten the 0 added from $Asgn0(B)$. The values of all other variables do not change. If $a \notin B$, then neither B nor E change. □

Lemma 5. *For all a, z, E, B, s where $B \triangleright s$ is closed under E:*

$$[\![B \triangleright s]\!]_E = [\![B \triangleright s]\!]_{E,Asgn0(B)}$$

Proof. By induction on the length of B.
Base case: When B is empty, $Asgn0(B)$ adds no variables, so they are the same.
Induction step: $B = b\langle @y\rangle \triangleright B'$:

$$\begin{aligned} [\![B \triangleright s]\!]_{E,Asgn0(B)} &= \{w_1.b\langle z\rangle.w_2 \in [\![B' \triangleright s]\!]_{E,Asgn0(B),y\leftarrow z} | b \notin w_1\} \cup \{w|b \notin w \text{ and } \forall z.w \in [\![B' \triangleright s]\!]_{E,Asgn0(B),y\leftarrow z}\} \\ &= \{w_1.b\langle z\rangle.w_2 \in [\![B' \triangleright s]\!]_{E,y\leftarrow z,Asgn0(B')} | b \notin w_1\} \cup \{w|b \notin w \text{ and } \forall z.w \in [\![B' \triangleright s]\!]_{E,y\leftarrow z,Asgn0(B')}\} \\ \text{By induction:} \quad &= \{w_1.b\langle z\rangle.w_2 \in [\![B' \triangleright s]\!]_{E,y\leftarrow z} | b \notin w_1\} \cup \{w|b \notin w \text{ and } \forall z.w \in [\![B' \triangleright s]\!]_{E,y\leftarrow z}\} \\ &= [\![B \triangleright s]\!]_E \end{aligned}$$

□

Theorem 7. *Restatement of Theorem 2: For any word w, bindings B, environment E, and predicates P, if an FLM pattern $B \triangleright s$ is in prefix form, $B \triangleright s$ is closed under E, and $B \triangleright s$ is implementable, then: $T_w(w, B, (E, Asgn0(B)), P) \in L(T_{re}(s, P))$ if and only if $w \in [\![B \triangleright s]\!]_E$.*

Proof. By induction on the length of w.

For the base case, T_{re} does not change the length of accepted words. So, $\varepsilon \in T_{re}(s, P) \iff \varepsilon \in [\![B \triangleright s]\!]_E$ (ε is the only word of length 0).

For the induction step: $w = a\langle z\rangle.w'$; $s' = D_{re}(a\langle z\rangle; s; E, Asgn0(B))$; and $B'; E' = D_{re}(a\langle z\rangle; B; E)$.
Starting with:

$$T_w(a\langle z\rangle.w', B, (E, Asgn0(B)), P) \in L(T_{re}(s, P))$$

By the definition of T_w and applying 4, This is equivalent to:

$$T_l(a\langle z\rangle, (E', Asgn0(B')), P).T_w(w', B', (E', Asgn0(B'), P) \in L(T_{re}(s, P))$$

Where $B', (E', Asgn0(B') = D_{bind}(a\langle z\rangle, B, (E, Asgn0(B)))$. By the definition of the classic derivative, this is equivalent to:

$$T_w(w', B', (E', Asgn0(B'), P) \in L(D_{clas}(T_l(a\langle z\rangle, (E', Asgn0(B'), P); T_{re}(s, P)))$$

By 4, this is equivalent to:

$$w' \in [\![B' \triangleright D_{re}(a\langle z\rangle, s, (E', Asgn0(B'))]\!]_{(E', Asgn0(B')}$$

By 6, this is true if and only if:

$$a\langle z\rangle.w' \in [\![B \triangleright s]\!]_{E, Asgn0(B)}$$

Finally, we can use 5 to get the result:

$$w \in [\![B \triangleright s]\!]_E$$

□

Lemma 6. *If $r \xrightarrow{rw} r'$ and r is implementable, then for all E such that r and r' are closed under E, $[\![r]\!]_E = [\![r']\!]_E$.*

Proof. By cases on which rewrite rule is used.

1. $r = b\langle @y; p\rangle.s \xrightarrow{rw} b\langle @y\rangle \triangleright b\langle p\rangle.s$

$$[\![b\langle @y\rangle \triangleright b\langle p\rangle.s]\!]_E$$

$$= \{w_1.b\langle z\rangle.w_2 \in [\![b\langle p\rangle.s]\!]_{E, y\leftarrow z} | b \notin w_1\} \cup \{w | \forall z.w \in [\![b\langle p\rangle.s]\!]_{E, y\leftarrow z} \wedge b \notin w\}$$

The right-hand set is empty, since any word in it must start with a b event.Similarly, w_1 must be ε for any word in the left-hand set.

$$= \{b\langle z\rangle.w_2 \in [\![b\langle p\rangle.s]\!]_{E, y\leftarrow z}\}$$

$$= \bigcup_{z \in Z} (\{b\langle z\rangle | [\![p(z)]\!]_E\} \circ [\![r]\!]_{E, y\leftarrow z})$$

$$= [\![b\langle @y; p\rangle.s]\!]_E$$

2. $r = B_1 \triangleright (s_1.(b\langle @y\rangle \triangleright s_2) \xrightarrow{rw} (B_1 \triangleright b\langle @y\rangle \triangleright s_1.s_2)$
In this case, we know that $b\langle @y\rangle \triangleright s_2$ and $B_1 \triangleright s_1$ are implementable, that $b \notin B_1 \triangleright s_1$, and that $y \notin B_1$. We can now show the two are equivalent by induction on the length of B_1. In the base case, we use the fact that y is out of scope in s_1:

$$[\![s_1.(b\langle @y\rangle \triangleright s_2)]\!]_E$$

$$= \{w_1.w_2 | w_1 \in [\![s_1]\!]_E \wedge w_2 \in [\![(b\langle @y\rangle \triangleright s_2)]\!]_E\}$$

$$= \{w_1.w_2 | (\forall z.w_1 \in [\![s_1]\!]_{E, y\leftarrow z}) \wedge w_2 \in [\![(b\langle @y\rangle \triangleright s_2)]\!]_E\}$$

$$= \{w_1.w_2 | (\forall z.w_1 \in [\![s_1]\!]_{E, y\leftarrow z}) \wedge (w_2 \in \{w_1'.b\langle z\rangle.w_2' \in [\![s_2]\!]_{E, y\leftarrow z} | b \notin w_1'\} \cup \{w | \forall z.w \in [\![s_2]\!]_{E, y\leftarrow z} \wedge b \notin w\})\}$$

Because we are quantifying over z for w_1, it is clear that any $w_1.w_2$ in this set must also be in $[\![b\langle @y\rangle \triangleright s_1.s_2]\!]_E$, depending on which set w_2' is in:

$$[\![b\langle @y\rangle \triangleright s_1.s_2]\!]_E$$
$$= \{w_1''.b\langle z\rangle.w_2'' \in [\![s_1.s_2]\!]_{E,y\leftarrow z} | b \notin w_1''\} \cup \{w' | \forall z.w' \in [\![s_1.s_2]\!]_{E,y\leftarrow z} \wedge b \notin w'\}$$

If w_2' is in the right-hand set, then $w_1'' = w_1.w_1'$ and $w_2'' = w_2'$. Otherwise, $w' = w_1.w$. The same expansions show membership the other way; if a word $w_1''.b\langle z\rangle.w_2'' \in [\![s_1.s_2]\!]_{E,y\leftarrow z}$, then a prefix of $w_1'' \in [\![s_1]\!]_{E,y\leftarrow z} = [\![s_1]\!]_E$, and the rest is in the right-hand set above. Otherwise, a prefix of w' is in s_1 and the rest is in the left-hand set above.

The induction case follows directly from expanding the definition:

$$[\![b_1\langle @y_1\rangle \triangleright B_1' \triangleright (s_1.(b\langle @y\rangle \triangleright s_2)]\!]_E$$

$$= \{w_1.b\langle z\rangle.w_2 \in [\![B_1' \triangleright (s_1.(b\langle @y\rangle \triangleright s_2)]\!]_{E,y_1\leftarrow z} | b \notin w_1\} \cup \{w | \forall z.w \in [\![B_1' \triangleright (s_1.(b\langle @y\rangle \triangleright s_2)]\!]_{E,y_1\leftarrow z} \wedge b \notin w\}$$

Using induction:

$$= \{w_1.b\langle z\rangle.w_2 \in [\![B_1' \triangleright b\langle @y\rangle \triangleright (s_1.s_2)]\!]_{E,y_1\leftarrow z} | b \notin w_1\} \cup \{w | \forall z.w \in [\![B_1' \triangleright b\langle @y\rangle \triangleright (s_1.s_2)]\!]_{E,y_1\leftarrow z} \wedge b \notin w\}$$

3.

$$(b\langle @y_1\rangle \triangleright s_1) + (b\langle @y_2\rangle \triangleright s_2) \xrightarrow{rw} b\langle y_1\rangle \triangleright (s_1 + [y_1/y_2]s_2)$$

Here, we assume both $b\langle @y_1\rangle \triangleright s_1$ and $b\langle @y_2\rangle \triangleright s_2$ are implementable. The proof follows directly from expanding the two $\triangleright$ expressions, and substituting the variables in s_2.

$$[\![b\langle @y_1\rangle \triangleright (s_1 + [y_1/y_2]s_2)]\!]_E$$

$$= \{w_1.b\langle z\rangle.w_2 \in [\![s_1 + [y_1/y_2]s_2]\!]_{E,y_1\leftarrow z} | b \notin w_1\} \cup \{w | \forall z.w \in [\![s_1 + [y_1/y_2]s_2]\!]_{E,y_1\leftarrow z} \wedge b \notin w\}$$
$$= \{w_1.b\langle z\rangle.w_2 \in [\![s_1]\!]_{E,y_1\leftarrow z} \cup [\![[y_1/y_2]s_2]\!]_{E,y_1\leftarrow z} | b \notin w_1\} \cup \{w | \forall z.w \in [\![s_1]\!]_{E,y_1\leftarrow z} \cup [\![[y_1/y_2]s_2]\!]_{E,y_1\leftarrow z} \wedge b \notin w\}$$
$$= \{w_1.b\langle z\rangle.w_2 \in [\![s_1]\!]_{E,y_1\leftarrow z} \cup [\![s_2]\!]_{E,y_2\leftarrow z} | b \notin w_1\} \cup \{w | \forall z.w \in [\![s_1]\!]_{E,y_1\leftarrow z} \cup [\![s_2]\!]_{E,y_2\leftarrow z} \wedge b \notin w\}$$
$$= [\![(b\langle @y_1\rangle \triangleright s_1) + (b\langle @y_2\rangle \triangleright s_2)]\!]_E$$

4.

$$(b\langle @y_1\rangle \triangleright s_1) + (c\langle @y_2\rangle \triangleright s_2) \xrightarrow{rw} b\langle @y_1\rangle \triangleright c\langle @y_2\rangle \triangleright (s_1 + s_2)$$

We assume that $b\langle @y_1\rangle \triangleright s_1$ and $c\langle @y_2\rangle \triangleright s_2$ are implementable. The proof follows directly from expanding the definitions of the two $\triangleright$ expressions.

$$[\![(b\langle @y_1\rangle \triangleright s_1) + (c\langle @y_2\rangle \triangleright s_2)]\!]_E$$
$$= [\![b\langle @y_1\rangle \triangleright s_1]\!]_E \cup [\![c\langle @y_2\rangle \triangleright s_2]\!]_E$$
$$= S1 : \{w | w = w_1.b\langle z_1\rangle.w_2 \in [\![s_1]\!]_{E,y_1\leftarrow z_1} \wedge b \notin w_1\} \cup S2 : \{w | \forall z.w \in [\![s_1]\!]_{E,y_1\leftarrow z} \wedge b \notin w\}$$
$$\cup S3 : \{w | w = w_1.c\langle z_2\rangle.w_2 \in [\![s_2]\!]_{E,y_2\leftarrow z_2} \wedge c \notin w_1\} \cup S4 : \{w | \forall z.w \in [\![s_2]\!]_{E,y_2\leftarrow z} \wedge c \notin w\}$$

Expanding the other definition:

$$[\![b\langle @y_1\rangle \triangleright c\langle @y_2\rangle \triangleright (s_1 + s_2)]\!]_E$$
$$= \{w | w = w_1.b\langle z_1\rangle.w_2 \in [\![c\langle @y_2\rangle \triangleright (s_1 + s_2)]\!]_{E,y_1\leftarrow z_1} \wedge b \notin w_1\} \cup \{w | \forall z.w \in [\![c\langle @y_2\rangle \triangleright (s_1 + s_2)]\!]_{E,y_1\leftarrow z_1} \wedge b \notin w\}$$

After expanding this to another 4 sets, it is clear that $[\![(b\langle @y_1\rangle \triangleright s_1) + (c\langle @y_2\rangle \triangleright s_2)]\!]_E \subseteq [\![b\langle @y_1\rangle \triangleright c\langle @y_2\rangle \triangleright (s_1 + s_2)]\!]_E$ by cases on which set ($S1, S2, S3, S4$) a word is in, and similarly in the reverse direction.

5.

$$(b\langle @y\rangle \triangleright s_1) + s_2 \xrightarrow{rw} b\langle @y\rangle \triangleright (s_1 + s_2)$$

This proof is very similar to the above with one fewer expansion, because the binding-free s_2 is always implementable.

The rules (equivalent to 3,4, and 5) for & have similar proof structure, but are simplified by using intersection rather than union. □

Lemma 7. *If* $r \xrightarrow{rw} r'$ *and r is implementable, then so is* r'.

Proof. By cases on which rewrite rule is used.

1. $r = b\langle @y;p\rangle.s \xrightarrow{rw} b\langle @y\rangle \triangleright b\langle p\rangle.s$
   We assume that s is binding free, so that $b\langle @y\rangle \triangleright b\langle p\rangle.s$ is in prefix form. Now, we show that it is implementable, for any $a \neq b$, the derivative of s is the same no matter the environment, since the following does not depend on z_1 at all:
$$\begin{aligned}&D_{re}(a\langle z\rangle;b\langle p\rangle.s;E,y\leftarrow z_1)\\ &= D_{re}(a\langle z\rangle;b\langle p\rangle;E,y\leftarrow z_1).s + \nu(b\langle p\rangle.D_{re}(a\langle z\rangle;s;E,y\leftarrow z_1)\\ &= \emptyset.s+\emptyset.D_{re}(a\langle z\rangle;s;E,y\leftarrow z_1)\\ &= \emptyset\end{aligned}$$

2. $r = (B_1 \triangleright s_1).(b\langle @y\rangle \triangleright s_2) \xrightarrow{rw} (B_1 \triangleright b\langle @y\rangle \triangleright s_1.s)$
   In this case, we know that $b\langle @y\rangle \triangleright s_2$ and $B_1 \triangleright s_1$ are implementable, that $b \notin B_1 \triangleright s_1$, and that $y \notin B_1$. Now, we can expand the expression, for some $a\langle z\rangle \notin B_1 \triangleright b$ and compatible environments G_1, G_2 to $B_1 \triangleright b$. We will write these as $G_1', y \leftarrow z_1$ and $G_2', y \leftarrow z_2$, separating out the compatible environments to B_1 and B_2, and expand definitions:
$$\begin{aligned}&[\![D_{re}(a\langle z\rangle;s_1.s_2;(E,G_1))]\!]_{E,G_2}\\ &= [\![D_{re}(a\langle z\rangle;s_1;(E,G_1)).s_2+\nu(s_1).D_{re}(a\langle z\rangle;s_2;(E,G_1))]\!]_{E,G_2}\\ &= [\![D_{re}(a\langle z\rangle;s_1;(E,G_1)).s_2]\!]_{E,G_2}\cup[\![\nu(s_1).D_{re}(a\langle z\rangle;s_2;(E,G_1))]\!]_{E,G_2}\\ &= [\![D_{re}(a\langle z\rangle;s_1;((E,y\leftarrow z_1),G_1')).s_2]\!]_{(E,y\leftarrow z_2),G_2'}\cup[\![\nu(s_1).D_{re}(a\langle z\rangle;s_2;((E,G_1'),y\leftarrow z_1))]\!]_{(E,G_2'),y\leftarrow z_2}\end{aligned}$$
   On the left, because y is not in scope in s_1, its value does not change the semantics of its derivative, so we can replace z_1 with z_2. Similarly, on the right we can replace G_1' with G_2' because those variables are out of scope in s_2:
$$\begin{aligned}&= [\![D_{re}(a\langle z\rangle;s_1;((E,y\leftarrow z_2),G_1')).s_2]\!]_{(E,y\leftarrow z_2),G_2'}\cup[\![\nu(s_1).D_{re}(a\langle z\rangle;s_2;((E,G_2'),y\leftarrow z_1))]\!]_{(E,G_2'),y\leftarrow z_2}\\ &= [\![D_{re}(a\langle z\rangle;s_1;((E,y\leftarrow z_2),G_1'))]\!]_{(E,y\leftarrow z_2),G_2'}\circ[\![s_2]\!]_{(E,y\leftarrow z_2),G_2'}\cup[\![\nu(s_1]\!]_{(E,G_2'),y\leftarrow z_2}\circ[\![D_{re}(a\langle z\rangle;s_2;((E,G_2'),y\leftarrow z_1))]\!]_{(E,G_2'),y\leftarrow z_2}\end{aligned}$$
   Finally, we can apply the inductive hypothesis for s_1 and s_2 and roll back up the definitions to get the desired result:
$$\begin{aligned}&= [\![D_{re}(a\langle z\rangle;s_1;((E,y\leftarrow z_2),G_2'))]\!]_{(E,y\leftarrow z_2),G_2'}\circ[\![s_2]\!]_{(E,y\leftarrow z_2),G_2'}\cup[\![\nu(s_1]\!]_{(E,G_2'),y\leftarrow z_2}\circ[\![D_{re}(a\langle z\rangle;s_2;((E,G_2'),y\leftarrow z_2))]\!]_{(E,G_2'),y\leftarrow z_2}\\ &= [\![D_{re}(a\langle z\rangle;s_1;((E,y\leftarrow z_2),G_1')).s_2]\!]_{(E,y\leftarrow z_2),G_2'}\cup[\![\nu(s_2).D_{re}(a\langle z\rangle;s_2;((E,G_2'),y\leftarrow z_2))]\!]_{(E,G_2'),y\leftarrow z_2}\\ &= [\![D_{re}(a\langle z\rangle;s_1;((E,y\leftarrow z_2),G_1')).s_2+\nu(s_2).D_{re}(a\langle z\rangle;s_2;((E,G_2'),y\leftarrow z_2))]\!]_{(E,G_2'),y\leftarrow z_2}\\ &= [\![D_{re}(a\langle z\rangle;s_1.s_2;E,G_2)]\!]_{E,G_2}\end{aligned}$$

3. $$(b\langle @y_1\rangle \triangleright s_1)+(b\langle @y_2\rangle \triangleright s_2) \xrightarrow{rw} b\langle y_1\rangle \triangleright (s_1+[y_1/y_2]s_2)$$
   Here, we assume both $b\langle @y_1\rangle \triangleright s_1$ and $b\langle @y_2\rangle \triangleright s_2$ are implementable. Then, we can expand definitions, writing $G_1 as(y_1 \leftarrow z_1)$ and G_2 as $(y_1 \leftarrow z_2)$:
$$\begin{aligned}&[\![D_{re}(a\langle z\rangle;s_1+[y_1/y_2]s_2;E,y_1\leftarrow z_1)]\!]_{E,y_1\leftarrow z_2}\\ &= [\![D_{re}(a\langle z\rangle;s_1;E,y_1\leftarrow z_1)+D_{re}(a\langle z\rangle;[y_1/y_2]s_2;E,y_1\leftarrow z_1)]\!]_{E,y_1\leftarrow z_2}\\ &= [\![D_{re}(a\langle z\rangle;s_1;E,y_1\leftarrow z_1)]\!]_{E,y_1\leftarrow z_2}\cup[\![D_{re}(a\langle z\rangle;[y_1/y_2]s_2;E,y_1\leftarrow z_1)]\!]_{E,y_1\leftarrow z_2}\end{aligned}$$
   On the left, we can apply the induction hypothesis directly. On the right, we undo the substitution and use it, then roll back up the definition.
$$\begin{aligned}&= [\![D_{re}(a\langle z\rangle;s_1;E,y_1\leftarrow z_1)]\!]_{E,y_1\leftarrow z_2}\cup[\![D_{re}(a\langle z\rangle;s_2;E,y_2\leftarrow z_1)]\!]_{E,y_2\leftarrow z_2}\\ &= [\![D_{re}(a\langle z\rangle;s_1;E,y_1\leftarrow z_2)]\!]_{E,y_1\leftarrow z_2}\cup[\![D_{re}(a\langle z\rangle;s_2;E,y_2\leftarrow z_2)]\!]_{E,y_2\leftarrow z_2}\\ &= [\![D_{re}(a\langle z\rangle;s_1;E,y_1\leftarrow z_2)]\!]_{E,y_1\leftarrow z_2}\cup[\![D_{re}(a\langle z\rangle;[y_1/y_2]s_2;E,y_1\leftarrow z_2)]\!]_{E,y_1\leftarrow z_2}\\ &= [\![D_{re}(a\langle z\rangle;s_1+s_2;E,G_2)]\!]_{G_2}\end{aligned}$$

4.

$$(b\langle @y_1\rangle \triangleright s_1) + (c\langle @y_2\rangle \triangleright s_2) \xrightarrow{rw} b\langle @y_1\rangle \triangleright c\langle @y_2\rangle \triangleright (s_1 + s_2)$$

We assume that $b\langle @y_1\rangle \triangleright s_1$ and $c\langle @y_2\rangle \triangleright s_2$ are implementable, and expand definitions, writing G_1 as $y_1 \leftarrow z_1, y_2 \leftarrow z_2$ and G_2 as $y_1 \leftarrow z_1', y_2 \leftarrow z_2'$:

$$\llbracket D_{re}(a\langle z\rangle; s_1 + s_2; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2)\rrbracket_{E, y_1 \leftarrow z_1', y_2 \leftarrow z_2'}$$

$$= \llbracket D_{re}(a\langle z\rangle; s_1; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2) + D_{re}(a\langle z\rangle; s_2; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2)\rrbracket_{E, y_1 \leftarrow z_1', y_2 \leftarrow z_2'}$$

$$= \llbracket D_{re}(a\langle z\rangle; s_1; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2)\rrbracket_{E, y_1 \leftarrow z_1', y_2 \leftarrow z_2'} \cup \llbracket D_{re}(a\langle z\rangle; s_2; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2)\rrbracket_{E, y_1 \leftarrow z_1', y_2 \leftarrow z_2'}$$

y_1 is out of scope in s_2, and y_2 is out of scope in s_1. So, we can assign any value to them without changing the derivative:

$$= \llbracket D_{re}(a\langle z\rangle; s_1; E, y_1 \leftarrow z_1, y_2 \leftarrow z_2')\rrbracket_{E, y_1 \leftarrow z_1', y_2 \leftarrow z_2'} \cup \llbracket D_{re}(a\langle z\rangle; s_2; E, y_1 \leftarrow z_1', y_2 \leftarrow z_2)\rrbracket_{E, y_1 \leftarrow z_1', y_2 \leftarrow z_2'}$$

Finally, we can apply the induction hypothesis for the other variables, using the implementability property. Then, we roll back up the definitions.

$$= \llbracket D_{re}(a\langle z\rangle; s_1; E, y_1 \leftarrow z_1', y_2 \leftarrow z_2')\rrbracket_{E, y_1 \leftarrow z_1', y_2 \leftarrow z_2'} \cup \llbracket D_{re}(a\langle z\rangle; s_2; E, y_1 \leftarrow z_1', y_2 \leftarrow z_2')\rrbracket_{E, y_1 \leftarrow z_1', y_2 \leftarrow z_2'}$$

$$= \llbracket D_{re}(a\langle z\rangle; s_1 + s_2; E, y_1 \leftarrow z_1', y_2 \leftarrow z_2')\rrbracket_{E, y_1 \leftarrow z_1', y_2 \leftarrow z_2'}$$

5.

$$(b\langle @y\rangle \triangleright s_1) + s_2 \xrightarrow{rw} b\langle @y\rangle \triangleright (s_1 + s_2)$$

This proof is very similar to the above, because the binding-free s_2 is always implementable.

$$\llbracket D_{re}(a\langle z\rangle; s_1 + s_2; E, y \leftarrow z_1)\rrbracket_{E, y \leftarrow z_2}$$

$$= \llbracket D_{re}(a\langle z\rangle; s_1; E, y \leftarrow z_1)\rrbracket_{E, y \leftarrow z_2} \cup \llbracket D_{re}(a\langle z\rangle; s_2; E, y \leftarrow z_1)\rrbracket_{E, y \leftarrow z_2}$$

y is out of scope in s_2, so we can replace its value with anything without changing the derivative.

$$= \llbracket D_{re}(a\langle z\rangle; s_1; E, y \leftarrow z_1)\rrbracket_{E, y \leftarrow z_2} \cup \llbracket D_{re}(a\langle z\rangle; s_2; E, y \leftarrow z_2)\rrbracket_{E, y \leftarrow z_2}$$

Now, we apply the induction hypothesis for s_1, and get the require result:

$$= \llbracket D_{re}(a\langle z\rangle; s_1; E, y \leftarrow z_2)\rrbracket_{E, y \leftarrow z_2} \cup \llbracket D_{re}(a\langle z\rangle; s_2; E, y \leftarrow z_2)\rrbracket_{E, y \leftarrow z_2}$$

$$= \llbracket D_{re}(a\langle z\rangle; s_1 + s_2; E, y \leftarrow z_2)\rrbracket_{E, y \leftarrow z_2}$$

The & forms of rules 3, 4, and 5 above have very similar proofs, replacing $+$ and $\cup$ with & and $\cap$. $\square$