## Deliverable Report

<table>
<thead>
<tr>
<th>Deliverable title</th>
<th>Extended BEBA abstraction API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>1.0</td>
</tr>
<tr>
<td>Due date of deliverable</td>
<td>June 2016</td>
</tr>
<tr>
<td>(month)</td>
<td></td>
</tr>
<tr>
<td>Actual submission date</td>
<td>01/07/2016</td>
</tr>
<tr>
<td>of the deliverable</td>
<td></td>
</tr>
<tr>
<td>(dd/mm/yyyy)</td>
<td></td>
</tr>
<tr>
<td>Start date of project</td>
<td>01/01/2015</td>
</tr>
<tr>
<td>(dd/mm/yyyy)</td>
<td></td>
</tr>
<tr>
<td>Duration of the project</td>
<td>27 months</td>
</tr>
<tr>
<td>Work Package</td>
<td>WP2</td>
</tr>
<tr>
<td>Task</td>
<td>T2.2</td>
</tr>
<tr>
<td>Leader for this deliverable</td>
<td>CNIT</td>
</tr>
<tr>
<td>Other contributing partners</td>
<td>NEC, KTH, CESNET</td>
</tr>
<tr>
<td>Authors</td>
<td>Giuseppe Bianchi, Marco Bonola, Davide Sanvito, Carmelo Cascone (CNIT), Georgios Katsikas, Dejan Kostic (KTH), Libor Polcak (CESNET), Roberto Bifulco (NEC)</td>
</tr>
<tr>
<td>Deliverable reviewer(s)</td>
<td>Marco Bonola (CNIT), Viktor Puš (CESNET)</td>
</tr>
</tbody>
</table>
Project co-funded by the European Commission within the Horizon 2020 (H2020) Programme

DISSEMINATION LEVEL

| PU | Public | X |
| PP | Restricted to other programme participants (including the Commission Services) |
| RE | Restricted to a group specified by the consortium (including the Commission Services) |
| CO | Confidential, only for members of the consortium (including the Commission Services) |

REVISION HISTORY

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>Author</th>
<th>Organisation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>26/04/2016</td>
<td>Bianchi</td>
<td>CNIT</td>
<td>TOC, and section 1,2 first draft</td>
</tr>
<tr>
<td>0.2</td>
<td>03/05/2016</td>
<td>Sanvito</td>
<td>CNIT</td>
<td>Section 3 added</td>
</tr>
<tr>
<td>0.3</td>
<td>04/05/2016</td>
<td>Bifulco</td>
<td>NEC</td>
<td>Section 8</td>
</tr>
<tr>
<td>0.4</td>
<td>30/05/2016</td>
<td>Bonola</td>
<td>CNIT</td>
<td>Section 4</td>
</tr>
<tr>
<td>0.5</td>
<td>14/06/2016</td>
<td>Polcak</td>
<td>CESNET</td>
<td>Section 4.1.1</td>
</tr>
<tr>
<td>0.6</td>
<td>16/06/2016</td>
<td>Katsikas</td>
<td>KTH</td>
<td>Section 5</td>
</tr>
<tr>
<td>0.7</td>
<td>22/06/2016</td>
<td>Cascone</td>
<td>CNIT</td>
<td>Section 7</td>
</tr>
<tr>
<td>0.8</td>
<td>30/06/2016</td>
<td>Puš</td>
<td>CESNET</td>
<td>Sections 4 thru 7 review</td>
</tr>
<tr>
<td>1.0</td>
<td>30/06/2016</td>
<td>Bonola</td>
<td>CNIT</td>
<td>Final version</td>
</tr>
</tbody>
</table>

PROPRIETARY RIGHTS STATEMENT

This document contains information, which is proprietary to the BEBA consortium. Neither this document nor the information contained herein shall be used, duplicated or communicated by any means to any third party, in whole or in parts, except with the prior written consent of the BEBA consortium. This restriction legend shall not be altered or obliterated on or from this document.

STATEMENT OF ORIGINALITY

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both
TABLE OF CONTENT

EXECUTIVE SUMMARY ........................................................................................................... 5

1 INTRODUCTION .................................................................................................................. 6
  1.1 SUMMARY OF THE BEBA TASK 2.3 GOALS ............................................................. 6

2 BEBA FULL XFSM ARCHITECTURE DESCRIPTION .......................................................... 8
  2.1 INTRODUCTION ........................................................................................................... 8
     2.1.1 Summary of the contribution ................................................................................. 9
  2.2 CONCEPT ....................................................................................................................... 11
     2.2.1 XFSM abstraction ................................................................................................. 12
     2.2.2 Programming the BEBA full XFSM switch .......................................................... 16

3 BEBA PROTOCOL EXTENSIONS FOR FULL XFSM SUPPORT ........................................... 18
  3.1 EXPERIMENTER MESSAGES ......................................................................................... 18
     3.1.1 State modification messages .............................................................................. 18
  3.2 EXPERIMENTER ACTIONS ........................................................................................... 21
     3.2.1 Set-data-variable action .................................................................................. 21
     3.2.2 Match fields ...................................................................................................... 23
  3.3 EXPERIMENTER STATISTICS MESSAGE .................................................................. 25
     3.3.1 State statistics message .................................................................................... 25
  3.4 SIMPLE TOY EXAMPLE .............................................................................................. 26

4 NEW PERSPECTIVES FOR USE CASE IMPLEMENTATION ..................................................... 30
  4.1.1 Monitoring in BEBA networks .............................................................................. 30
  4.1.2 A simple port scan detection and mitigation ....................................................... 32
  4.2 TRAFFIC POLICING WITH TOKEN BUCKETS ............................................................. 34
  4.3 DECISION TREE BASED TRAFFIC CLASSIFICATION ............................................... 35

5 PRELIMINARY RESULTS FOR A BEBA HIGH LEVEL LANGUAGE ..................................... 38
5.1 **Building High-level Programming Abstractions** ........................................ 38
5.2 **Programming Abstractions in the Basic BEBA API** ........................................ 38
5.3 **Example BEBA Applications** ........................................................................ 40
5.4 **MAC Learning Use Case (UC01)** ...................................................................... 40
5.5 **Preliminary Analysis for a Programming Abstractions in the Extended BEBA API** ...... 44

6 **Discussions** ........................................................................................................... 45
6.1 **Proposed Abstraction: At Which “Level”?** ......................................................... 45
6.2 **Use of P4 for the Description of BEBA Advanced Stages.** ................................... 45
6.2.1 **BEBA_advanced.p4 library** ............................................................................. 46
6.3 **Structural Limitations and Possible Extensions** ..................................................... 49

7 **Future Works** ......................................................................................................... 51
7.1 **Programmable ASICs** .......................................................................................... 51
7.2 **Hybrid Software-Hardware Switches** .................................................................... 51
7.3 **End-Host Stack Abstractions** ............................................................................... 52
7.4 **High-Level Programming Abstractions** ................................................................. 52
7.5 **Composition and Optimization** ............................................................................ 53

APPENDIX FORWARDING CONSISTENCY USE CASE (UC06)................................. 54
Executive summary
This deliverable documents the project’s results related to activity T2.2.
1 Introduction

1.1 Summary of the BEBA Task 2.3 goals

Task T2.3 "Extended XFSM-based API and flow computing engine" was originally meant to improve the BEBA results in defining a novel stateful abstraction for SDN programmable switches well beyond the simple support of state transitions at data plane. We briefly recall that the first phase of WP2 committed to specify and experimentally validate a basic abstraction targeting the stateful adaptation of flow forwarding rules, so as to provide very early in the important input for the ONF standardization activities. At the (possible) expense of generality and ambitiousness, the work in this phase has been very focused, with in mind compliance with OpenFlow, so as to provide a specification targeted to concretely impact the Open Networking Foundation.

Task T2.3 was motivated by a much more ambitious goal: transform a switch in a sort of network/flow processor programmed through a platform-independent abstraction. Our belief at the time of the BEBA proposal writing was that such goal could be accomplished by further introducing the ability to store temporary data into "memory registries" associated to flow entries, and provide the ability to enforce state transitions only if conditions on such registries are satisfied, as well as support registry updates upon the occurrence of events and/or state transitions. This forwarding behaviour can be conveniently formalized by means of abstraction in the form of an Extended Finite State Machine (XFSM). To support the execution of XFSM directly at data plane without the intervention of an external controller we addressed the server technical challenges, which include: (i) the ability to dynamically deploy memory registers per flow, and apply conditions (in the form of Boolean and simple arithmetic operations) before triggering state transitions, (ii) the identification of global switch-level registries (e.g. queue states, link level measurements, predefined flow statistics and counters, etc) and their integration in the state machine execution engine, (iii) the definition of an event matching specification language; (iv) support soft states by including means to configure relevant timers, and extend the set of programmable events to time-based ones (e.g. timer triggers). Many issues addressed in task 2.1 have been revisited in sight of the proposed extensions. The final challenge is to perform all the above operations at wire speed, on a packet by packet basis, without impairing performance and scalability.

This deliverable reports the work carried out in the frame of task T2.3 and describes the following results achieved by the project consortium:

1) The definition of the **BEBA advanced abstraction** support the stateful XFSM forwarding model

2) The extension of the BEBA protocol (between the switch and the external controller) to support the full XFSM switch primitives

3) The identification and support of a set of novel (and way more challenging) use cases that could not be support by the BEBA basic forwarding abstraction and API
4) Preliminary results in the definition of an high level BEBA programming language

Moreover, this deliverable reports two important discussions carried out among the partners involved in task 2.3:

1. A comparison between P4 and the BEBA full XFSM switch;
2. The future works identified to provide fully functional and deployable stateful programmable switch motivated by the BEBA ideas.

The reminder of this document maps the item listed above and is structured as follows:

• Section 2 describe the details of the BEBA full XFSM switch architecture
• Section 3 describe the BEBA full XFSM protocol
• Section 4 describes three new uses cases supported by the BEBA full XFSM abstraction
• Section 5 provide the comparison with P4
• Section 6 concludes this deliverable by discussing possible future research directions
2 BEBA full XFSM Architecture Description

2.1 Introduction

To face the emerging needs for service flexibility, network efficiency, traffic diversification, and network security and reliability, today's network nodes are called to a more flexible and richer packet processing. The original Internet nodes, historically limited to switches and routers providing "just" the plain forwarding services, have been massively complemented with a variety of heterogeneous middlebox-type functions [1,2,3,4] such as network address translation, tunneling, load balancing, monitoring, intrusion detection, and so on. The diversification of network equipment and technologies has definitely provided an increased availability of network functionalities, but at the cost of a significant extra complexity in the control and management of large scale multi-vendor networks.

Software-defined Networking (SDN) emerged as an attempt to address such problem. Coined in 2009 [5] as a direct follow-up of the OpenFlow proposal [6], SDN has broadly evolved since then [7] and does not in principle restricts to OpenFlow (a “minor piece in the SDN architecture”, according to the OpenFlow inventors themselves [8]) as device-level abstraction. Nevertheless, most of the high level network programming abstractions proposed in the last half a dozen years [9][10][11][12][13][14][15] still rely on OpenFlow as southbound (using RFC 7426’s terminology) programming interface. Indeed, OpenFlow was designed with the desire for rapid adoption, opposed to first principles [7]; i.e., as a pragmatic attempt to address the dichotomy between i) flexibility and ability to support a broad range of innovation, and ii) compatibility with commodity hardware and vendors' need for closed platforms [8].

The aftermath is that most of the above mentioned network programming frameworks circumvent OpenFlow's limitations by promoting a "two-tiered" [16] programming model: any stateful processing intelligence of the network applications is delegated to the network controller, whereas OpenFlow switches limit to install and enforce stateless packet forwarding rules delivered by the remote controller. Centralization of the network applications' intelligence may not be a problem (and actually turns out to be an advantage) whenever changes in the forwarding states do not have strict real time requirements, and depend upon global network states. But for applications which rely only on local flow/port states, the latency toll imposed by the reliance on an external controller rules away the possibility to enforce software-implemented control plane tasks at wire speed, i.e. while remaining on the fast path\(^1\)

\(^1\) A 64 bytes packet takes about 5 ns on a 100 gbps speed, roughly the time needed for a signal to reach a control entity placed one meter away. And the execution of an albeit simple software-implemented control task may take way more time than this. Thus, even the physical, capillary, distribution of control agents (as proxies of the remote SDN controller for low latency tasks) on each network device would hardly meet fast path requirements
One might argue that we do not even need such ultra-fast processing and packet-by-packet manipulation and control capabilities. However, not only the large real-world deployment of proprietary hardware network appliances (e.g. for traffic classification, control/balancing, monitoring, etc), but also the evolution of the OpenFlow specification itself shows that this may not be the case. As a matter of fact, since the creation of the Open Networking Foundation (ONF) in 2011, and up to the latest (version 1.5) specification, we have witnessed an hectic evolution of the OpenFlow standard, with several OpenFlow extensions devised to fix punctual shortcomings and accommodate specific needs, by incorporating extremely specific stateful primitives (such as meters for rate control, group ports for fast failover support or dynamic selection of one among many action buckets at each time - e.g. for load balancing -, synchronized tables for supporting learning-type functionalities, etc).

Indeed, in the last couple of years, a new research trend has started to challenge improved programmability of the data plane, beyond the elementary ```match/action``` abstraction provided by OpenFlow, and (even more recently) initial work on higher level network programming frameworks devised to exploit such newer and more capable lower-level primitives are starting to emerge [16][17]. Proposals such as POF [18][19], although not yet targeting stateful flow processing, do significantly improve header matching flexibility and programmability, freeing it from any specific structure of the packet header. Programmability of the packet scheduler inside the switch has been recently addressed in [20]. Works such as OpenState [21][22] and FAST [23] explicitly add support for per-flow state handling inside OpenFlow switches, although the abstractions therein defined are still simplistic and severely limit the type of applications that can be deployed (for instance, OpenState supports only a special type of Finite State Machines, namely Mealy Machines, which do not provide the programmer with the possibility to declare and use own memory or registries). The P4 programming language [24][25] leverages more advanced hardware technology, namely dedicated processing architectures [26] or Reconfigurable Match Tables [27] as an extension of TCAMs (Ternary Content Addressable Memories) to permit a significantly improved programmability in the packet processing pipeline. In its latest 1.0.2 language specification [28] P4 has made a further crucial step in improving stateful processing, by introducing registers defined as “stateful memories” [which] “can be used in a more general way to keep state”. However, the P4 language does not specify how registries should be scalably supported and managed by the underlying HW.

2.1.1 Summary of the contribution

This work is an attempt to revisit fast-path programmability, by (i) proposing a programming abstraction which retains the platform independent features of the original ```match/action``` OpenFlow abstraction, and by (ii) showing how our abstraction can be directly “executed” over an HW architecture (whose feasibility is concretely shown via an HW FPGA prototype). In analogy with the OpenFlow’s ```match/action``` abstraction, which exposes a network node’s TCAM to third party programmability, also our abstraction directly refers to the HW interface, and as such it can be directly exposed to the programmer as a machine-level “configuration”
interface, hence without any intermediary compilation or adaptation to the target (i.e., unlike the case of P4).

In conceiving our abstraction, we have been largely inspired by [21], where eXtended Finite State Machines [29] (therein referred to as “full” XFSM) were conjectured as a possible forward-looking abstraction. Our key difference with respect to [21] is that we not limit to postulate that such “full” XFSMs may ultimately be a suitable abstraction, but we concretely show their viability and their "executability" over an HW architecture leveraging commodity HW (standard TCAMs, hash tables, ALUs, and somewhat trivial additional circuitry), and with a strictly bounded number of clock ticks.

**Figure 1** A typical OpenFlow pipeline architecture. b) the BEBA full XFSM enabled pipeline. BEBA "stages" can be pipelined with other stages or ordinary OpenFlow Match/Action stages

A limitation in this work is our focus on a "single" packet processing stage, opposed to a more general packet processing pipeline comprising multiple match/action tables. In essence, our work shows the viability of an XFSM-based abstraction as a (significant) generalization of the original single-table OpenFlow’s match/action. While multiple pipelined instances of our atomic Open Packet Processor stages are clearly possible, exactly as multiple match/action tables can be pipelined since OpenFlow version 1.1 (see Figure 1), our present work does not yet take advantage of HW pipeline optimizations such as Reconfigurable Match Tables [27].

Finally, and similarly to the OpenFlow's original design philosophy, even if our proposed architecture is pragmatically limited by the specific set of primitives implemented by the HW (supported packet processing and forwarding actions, matching facilities, arithmetic and logic operations on registry values, etc), it nevertheless remains extensible (by adding new actions or instructions) and largely expressive in terms of how the programmer shall use and combine
such primitives within a desired stateful operation. As it will be hopefully apparent later on, a “full” XFSM permits to formally describe a wide variety of programmable packet processing and control tasks, which our architecture permits to directly convey and deploy inside the switch. And even if probably not of any practical interest, the fact that not only the OpenFlow legacy statistics, but also further tailored stateful extensions today integrated in the OpenFlow standard (hence hardcoded in the switch) might be externally programmed using an apparently viable platform agnostic abstraction merits further considerations.

### 2.2 Concept

As anticipated in the previous section, our work focuses on the design of a single BEBA full XFSM stage, as a significant generalization of the traditional OpenFlow's Match/Action abstraction. More specifically, our goal is to provide a packet processing stage which holds the following properties:

- **Ability to process packets directly on the fast path**, i.e., while the packet is traveling in the pipeline (nanoseconds time scale). The requirement of performing packet processing tasks in a deterministic and small (bounded) number of HW clock cycles hardly copes with the possibility to employ a standard CPU (and the relevant programming language), and requires us to implement a domain-specific (traffic/network) computing architecture from scratch.

- **Efficient storage and management of per-flow stateful information**. Other than parsing packet header information and exposing such fields to a match/action stage (or a pipeline of match action stages [26][27]), we also aim at permitting the programmer to further use the past flow history for defining a desired per-packet processing behaviour. This can easily accomplished by pre-pending a dedicated storage table (concretely, an hash table) that permits to retrieve, in O(1) time, stateful flow information. We name this structure as Flow Context Table, as, in somewhat analogy with context switching in ordinary operating systems, it permits to retrieve stateful information associated to the flow to which an arriving packet belongs, and store back an updated context the end of the packet processing pipeline. Such (flow) context switching will operate at wire speed, on a packet-by-packet basis.

- **Ability to specify and compute a wide (and programmable) class of stateful information**, thus including counters, running averages, and in most generality stateful features useful in traffic control applications. It readily follows that the packet processing pipeline, which in standard OpenFlow is limited to match/action primitives, must be enriched with means to describe and (on the fly) enforce conditions on stateful quantities (e.g. the flow rate is above a threshold, or the time elapsed since the last seen packet is greater than the average inter-arrival time), as well as provide arithmetic/logic operations so as to update such stateful features in a bounded number of clock cycles (ideally one).
• **Platform independence.** A key pragmatic insight in the original OpenFlow abstraction was the decision of restricting the OpenFlow switch programmer's ability to just select actions among a finite set of supported ones (opposed to permitting the programmer to develop own custom actions), and associate a desired action set (bundle) to a specific packet header match. We conceptually follow a similar approach, but we cast it into a more elaborate eXtended Finite State Machine (XFSM) model. An XFSM abstraction permits us to formalize complex behavioral models, involving custom per-flow states, custom per-flow registers, conditions, state transitions, and arithmetic and logic operations. Still, an XFSM model does not require us to know how such primitives are concretely implemented in the hardware platform, but "just" permits us to combine them together so as to formalize a desired behaviour. Hence, it can be ported across platforms which support a same set of primitives.

### 2.2.1 XFSM abstraction

The OpenFlow’s "Match-action" abstraction has been widely extended throughout the various standardization steps, with the extension of the match fields (including the possibility to perform matches on meta-data), with new actions (and instructions), and with the ability to associate a set of actions to a given match. Nevertheless, the basic abstraction conceptually remains the same. It is instructive to formally re-interpret the (basic) OpenFlow match/action abstraction as a map

\[ T : I \rightarrow O, \text{ where } I=i_1, \ldots, i_M \]

is a finite set of Input Symbols, namely all the possible matches which are technically supported by an OpenFlow specification (being irrelevant, at least for this discussion, to know how such Input Symbols' set \( I \) is established, and that each input symbol is a Cartesian combination of all possible header field matches), and \( O=o_1, \ldots, o_K \) is a finite set of Output Symbols, i.e. all the possible actions supported by an OpenFlow switch. The obvious limit of this abstraction is that the match/action mapping is statically configured, and can change only upon controller's intervention (e.g. via flow-mod OpenFlow commands). Finally, note that the “engine” which performs the actual mapping \( T : I \rightarrow O \) is a standard TCAM.

As observed in [21], an OpenFlow switch can be trivially extended to support a more general abstraction which takes the form of a Mealy Machine, i.e. a Finite State Machine with output, and which permits to formally model dynamic forwarding behaviors, i.e. permit to change in time the specific action(s) associated to a same match. It suffices to add a further finite set \( \{s_1, s_2, \ldots, s_N\} \) of *programmer-specific states*, and use the TCAM to perform the mapping \( T: S \times I \rightarrow S \times O \). While remaining feasible on ordinary OpenFlow hardware, such Mealy Machine abstraction brings about two key differences with respect to the original OpenFlow abstraction. First, the (output) action associated to a very same (input) match may now differ depending on an (input) state \( s_i \in S \), i.e., the state in which the flow is found when a packet is being processed. Second, the Mealy Machine permits to specify in which, possibly different, (output) state \( s_o \in S \) the flow shall enter once the packet will be processed. While quite
interesting, this generalization appears still insufficient to permit the programmer to implement meaningful applications, as it lacks the ability to run-time compute and exploit in the forwarding decisions per-flow features commonly used in traffic control algorithms.

Table 1 eXtended Finite State Machine model

<table>
<thead>
<tr>
<th>XFSM formal notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>input symbols</td>
</tr>
<tr>
<td>O</td>
<td>output symbols</td>
</tr>
<tr>
<td>S</td>
<td>custom states</td>
</tr>
<tr>
<td>D</td>
<td>n-dimensional linear space $D_1 \times \cdots \times D_n$</td>
</tr>
<tr>
<td>F</td>
<td>set of enabling functions $f_i : D \rightarrow {0,1}$</td>
</tr>
<tr>
<td>U</td>
<td>set of update functions $u_j : D \rightarrow D$</td>
</tr>
<tr>
<td>T</td>
<td>transition relation $T : S \times F \times I \rightarrow S \times U \times O$</td>
</tr>
</tbody>
</table>

The goal of the work described in section 2 is to show that a switch architecture can be further easily extended to support an even more general Finite State Machine model, namely the eXtended Finite State Machine (XFSM) model introduced in [29]. As summarized in table 1, this model is formally specified by means of a 7-tuple $M=(I,O,S,D,F,U,T)$. Input symbols $I$ (OpenFlow-type matches) and Output Symbols $O$ (actions) are the same as in OpenFlow. Per-application states $S$ are inherited from the Mealy Machine abstraction [21], and permit the programmer to freely specify the possible states in which a flow can be, in relation to her desired custom application (technically, a state label is handled as a bit string). For instance, in an heavy hitter detection application, a programmer can specify states such as NORMAL, MILD, or HEAVY, whereas in a load balancing application, the state can be the actual switch output port number (or the destination IP address) an already seen flow has been pinned to, or DEFAULT for newly arriving flows or flows that can be rerouted. With respect to a Mealy Machine, the key advantage of the XFSM model resides in the additional programming flexibility in three fundamental aspects.

1. **Custom (per-flow) registers and global (switch-level) parameters.** The XFSM model permits the programmer to explicitly define her own registers, by providing an array of per-flow variables whose content (time stamps, counters, average values, last TCP/ACK sequence number seen, etc) shall be decided by the programmer herself.
Additionally, it is useful to expose to the programmer (as further registers) also switch-level states (such as the switch queues' status) or global shared variables which all flows can access. Albeit practically very important, a detailed distinction into different register types is not foundational in terms of abstraction, and therefore all registers that the programmer can access (and eventually update) are summarized in the XFSM model presented in Table 1 via the array $D$ of memory registers.

2. **Custom conditions on registers and switch parameters.** The sheer majority of traffic control applications rely on comparisons, which permit to determine whether a counter exceeded some threshold, or whether some amount of time has elapsed since the last seen packet of a flow (or the first packet of the flow, i.e., the flow duration). The enabling functions $f_i : D \rightarrow \{0, 1\}$ serve exactly for this purpose, by implementing a set of (programmable) boolean comparators, namely conditions whose input can be decided by the programmer, and whose output is 1 or 0, depending on whether the condition is true or false. In turns, the outcome of such comparisons can be exploited in the transition relation, i.e. a state transition can be triggered only if a programmer-specific condition is satisfied.

3. **Register updates.** Along with the state transition, the XFSM models also permits the programmer to update the content of the deployed registers. As we will show later on, registers' updates require the HW to implement a set of update functions $u_i : D \rightarrow D$, namely arithmetic and logic primitives which must be provided in the HW pipeline, and whose input and output data shall be configured by the programmer.

Finally, we stress that the actual computational step in an XFSM is the transition relation $T : S \times F \times I \rightarrow S \times U \times O$, which is nothing else than a "map" (albeit with more complex inputs and outputs than the basic OpenFlow map), and hence is naturally implemented by the switch TCAM, as shown in the next section.
To our view, what makes the previously described XFSM abstraction compelling is the fact that it can be directly executed on the switch’s fast path using off the shelf HW, as we will prove in the future D2.4 with a concrete HW prototype. As discussed in the next section, practical restrictions of course emerge in terms of memory deployed for the registers, as well as capability of the ALUs used for register updates, but such restrictions are mostly related to an actual implementation, rather than to the design which remains at least in principle very general and flexible. A sketch of the proposed Open Packet Processor architecture is illustrated in Figure 2. The packet processing workflow is best explained by means of the following stages.

**Stage 1: flow context lookup.** Once a packet enters a processing block, the first task is to extract, from the packet, a Flow Identification Key (FK), which identifies the entity to which a state may be assigned. The flow is identified by an unique key composed of a subset of the information stored in the packet header. The desired FK is configured by the programmer (an IP address, a source/destination MAC pair, a 5-tuple flow identifier, etc) and depends on the specific application. The FK is used as index to lookup a Flow Context Table, which stores the flow context, expressed in terms of (i) the state label \( s_i \) currently associated to the flow, and (ii) an array \( R = \{R_0, R_1, ..., R_k\} \) of (up to) \( k+1 \) registers defined by the programmer. The retrieved flow context is then appended as metadata and the packet is forwarded to the next stage.

**Stage 2: conditions’ evaluation.** Goal of the Condition Block illustrated in Figure 2 (and implemented using ordinary boolean circuitry) is to compute programmer-specific conditions, which can take as input either the per flow register values (the array \( R \)), as well as global registers delivered to this block as an array \( G = \{G_0, G_1, ..., G_h\} \) of (up to) \( h+1 \) global variables and/or global switch states. Formally, this block is therefore in charge to implement the enabling functions specified by the XFSM abstraction. In practice, it is trivial to extend the
assessment of conditions also to packet header fields (for instance, port number greater than a given global variable or custom per-flow register). The output of this block is a boolean vector $C = c_0, c_1, ..., c_m$ which summarizes whether the $i$-th condition is true ($c_i=1$) or false ($c_i=0$).

**Stage 3: XFSM execution step.** Since boolean conditions have been transformed into 0/1 bits, they can be provided as input to the TCAM, along with the state label and the necessary packet header fields, to perform a wildcard matching (different conditions may apply in different states, i.e. a bit representing a condition can be set to *don't care* for some specific states). Each TCAM row models one transition in the XFSM, and returns a 3-tuple: (i) the next state in which the flow shall be set (which could coincide with the input state in the case of no state transition, i.e., a self-transition in the XFSM), (ii) the actions associated the transition (usual OpenFlow-type forwarding actions, such as drop, push_label, set_tos etc...), and (iii) the information needed to update the registers as described below.

**Stage 4: register updates.** Most applications require arithmetic processing when updating a stateful variable. Operations can be as simple as integer sums (to update counters or byte statistics) or can require tailored floating point processing (averages, exponential decays, etc). The role of the *Update logic block* component highlighted in Figure 2 is to implement an array of Arithmetic and Logic Units (ALUs) which support a selected set of computation primitives which permit the programmer to update (re-compute) the value of the registers, using as input the information available at this stage (previous values of the registers, information extracted from the packet, etc). We implement a set of operations which appear to be either useful to the specific network programmer's needs, as well as computationally effective in terms of implementation (ideally, executable in a single clock tick). It is worth to mention that the problem of extending the set of supported ALU instructions is merely a technical one, and does not affect the proposed architecture.

**Extension: Cross-flow context handling.** As noted in [21], there are many useful stateful control tasks, in which states for a given flow are updated by events occurring on different flows. A simple but prominent example is MAC learning: packets are forwarded using the destination MAC address, but the forwarding database is updated using the source MAC address. Thus, it may be useful to further generalize the XFSM abstraction as suggested in [21], i.e. by permitting the programmer to use a Flow Key during lookup (e.g. read information associated to a MAC destination address) and employ a possibly different Flow Key (e.g. associated to the MAC source) for updating a state or a register.

### 2.2.2 Programming the BEBA full XFSM switch

It is useful to conclude this section with at least a sketch of which types of applications (and programs) may be deployed.

A first trivial example of dynamic forwarding actions is that of a simple mechanism which distinguishes *short-lived flows from long-lived flows* by considering “long” any flow that has transmitted at least N packets, and applies different DSCP tags. The BEBA programmer would
simply need to define two states (DEFAULT also associated to every new flow, and LONG), one
per-flow register $R_0$ (a packet counter), one global register $G_0$ (storing the constant
threshold $N$), a condition $R_0 > G_0$ applicable when in state DEFAULT, and an update function
$ADD(R_0, 1) \rightarrow R_0$. Note that we did not assume any pre-implemented counters or meters in
the switch, but the counter and the relevant threshold check has been programmed using the
proposed abstraction.

The usage of packet inter-arrival times and timers is exemplified by a **dynamic intra-flow
load balancing** application, which can reroute a flow while it is in progress. As suggested in
[30], rerouting should *not* occur during packet bursts, to avoid out of ordering and relevant
performance impairments. Support in BEBA advanced just requires, for each packet being
transmitted, to update a per-flow register $R$ with the quantity $t + D$, being $t$ the actual packet
timestamp and $D$ a suitable threshold. When the next packet arrives (time $t_1$), we check the
condition $t_1 > R$. If this is false, we route the packet to the assigned path (indicated by the
state label); conversely we route it to an alternative path, and we change the state
accordingly. Again, note that we have not assumed any pre-implemented support from the
switch (e.g. timeouts or soft states), besides the ability to provide time information (e.g., via a
global register, or timestamps as packet metadata).

The integration in the ALU design of monitoring-specific update instructions (averages,
variances, smoothing filters) several features frequently used in traffic control and
classification applications can be computed on the fly during the pipeline. We defer relevant
examples to section 4.

Finally, traffic control with non-standard statistics might be needed in applications such as DNS
**reflection attack mitigation**. DNS reflection is a well-known DDoS attack that aims at
flooding a target server with unsolicited DNS responses as consequence of a huge number of
spoofed DNS requests. A basic mitigation strategy consists in dropping DNS responses when
anomalous conditions occur. In our proposed switch such system could be programmed using
two states: DEFAULT and ATTACK, with an associated drop action. To define anomalous
conditions, a possibility is to track, per each possible IP target, the number of DNS requests
and responses (hence two per-flow registers), compute (during register update operations)
**derived features** such as difference and/or ratio among requests and responses (not natively
provided by OpenFlow), specify the conditions which identify an attack conditions, and change
the flow state to ATTACK when such conditions are verified. Besides details (omitted, also
because the classifier example in section 4 will specifically discuss feature computation in a
more realistic setting), it’s worth to remark that different conditions can be enforced in
different states (e.g., hysteresis thresholds).
3 BEBA protocol extensions for Full XFSM support

This section describes an extension to the OpenFlow protocol specification v1.3 to enable support to stateful packet forwarding inside OpenFlow-enabled switches by considering as a starting point the BEBA basic stateful protocol discussed in section 4 of Deliverable D2.1.

With respect to the BEBA basic API we need these additional components:

1. an extended per-flow memory, the so called flow context, made of state+flow data variables
2. an extended set of global memories, the so called global data variables, which is a generalization of global states
3. the ability to configure conditions and to match on the result of conditions evaluation
4. new actions to update data variables

3.1 Experimenter messages

3.1.1 State modification messages

The configuration of a stateful stage are still performed with a State Modification message.

The main structure of the message has been preserved (it’s an experimenter message with exp_type field set to OFPT_EXP_STATE_MOD) but the list of possible values for 'command' field in ofp_exp_msg_state_mod has been extended with 4 additional commands.

```c
enum ofp_exp_msg_state_mod_commands {
    OFPSC_STATEFUL_TABLE_CONFIG = 0,
    OFPSC_EXP_SET_L_EXTRACTOR,
    OFPSC_EXP_SET_U_EXTRACTOR,
    OFPSC_EXP_SET_FLOW_STATE,
    OFPSC_EXP_DEL_FLOW_STATE,
    OFPSC_EXP_SET_GLOBAL_STATE,
    OFPSC_EXP_RESET_GLOBAL_STATE,
    OFPSC_EXP_SET_HEADER_FIELD_EXTRACTOR,
    OFPSC_EXP_SET_CONDITION,
    OFPSC_EXP_SET_GLOBAL_DATA_VAR,
    OFPSC_EXP_SET_FLOW_DATA_VAR
};
```

Set header field extractor message
An OFPT_STATE_MOD message with command field set to OFPSC_EXP_SET_HEADER_FIELD_EXTRACTOR has a payload structure as defined by ofp_exp_set_flow_state.

```c
struct ofl_exp_set_header_field_extractor {
    uint8_t table_id;
    uint8_t extractor_id;
    uint32_t field;
};
```

The `table_id` field specifies the table to be configured.

The `extractor_id` field specifies the ID of the header field extractor to be configured.

The `field` field contains the Type Length Value (TLV) of the field to be extracted.

**Set condition message**

An OFPT_STATE_MOD message with command field set to OFPSC_EXP_SET_CONDITION has a payload structure as defined by ofl_exp_set_condition.

```c
struct ofl_exp_set_condition {
    uint8_t table_id;
    uint8_t condition_id;
    uint8_t condition;
    uint8_t operand_types;
    uint8_t operand_1;
    uint8_t operand_2;
};
```

The `table_id` field specifies the table to be configured.

The `condition_id` field specifies the ID of the condition to be configured.

The `condition` field specifies the condition to be evaluated between the two operands and can assume one of the following constants:

```c
enum ofp_exp_conditions {
    CONDITION_GT = 0,
    CONDITION_LT,
};
```
CONDITION_GTE,
CONDITION_LTE,
CONDITION_EQ,
CONDITION_NEQ
};

The operand_types field is made of two 2-bits subfields to specify the type of each operand according to the following scheme: operand_types=xxyy0000 where xx=operand_1_type and yy=operand_2_type.

The two subfields can assume one of the following constants:

enum ofp_exp_operand_types {
    OPERAND_TYPE_FLOW_DATA_VAR = 0,
    OPERAND_TYPE_GLOBAL_DATA_VAR,
    OPERAND_TYPE_HEADER_FIELD,
    OPERAND_TYPE_CONSTANT
};

The last two fields operand_1 and operand_2 contains the ID of the operand (if the type is 0,1,2) or the value of the constant (if the type is 4).

**Set global data variable message**

An OFPT_STATE_MOD message with command field set to OFPSC_EXP_SET_GLOBAL_DATA_VAR has a payload structure as defined by ofp_exp_set_global_data_variable.

```c
struct ofp_exp_set_global_data_variable {
    uint8_t table_id;
    uint8_t global_data_variable_id;
    uint8_t pad[2];
    uint32_t value;
    uint32_t mask;
};
```

The table_id field specifies the table to be configured.

The global_data_variable_id field specifies the ID of the global data variable to be configured.
The value field contains the new value for the global data variable.

The mask field specifies which bits of the global data variable should be modified.

**Set flow data variable message**

An OFPT_STATE_MOD message with command field set to OFPSC_EXP_SET_FLOW_DATA_VAR has a payload structure as defined by `ofp_exp_set_flow_data_variable`.

```c
struct ofp_exp_set_flow_data_variable {
    uint8_t table_id;
    uint8_t flow_data_variable_id;
    uint8_t pad[2];
    uint32_t key_len;
    uint32_t value;
    uint32_t mask;
    uint8_t key[OFPSC_MAX_KEY_LEN];
};
```

The `table_id` field specifies the table to be configured.

The `flow_data_variable_id` field specifies the ID of the flow data variable to be configured.

The `value` field contains the new value for the flow data variable.

The `mask` field specifies which bits of the flow data variable should be modified.

The `key_len` field specifies the key size (number of bytes) of the key provided in `key[]`.

The `key` field contains the key used to select the flow context in the state table.

### 3.2 Experimenter actions

#### 3.2.1 Set-data-variable action

This action is a BEBA experimenter action with `act_type` field set to OFPAT_EXP_SET_DATA_VAR. This action allows to update the flow context in a particular stage of the pipeline or the global data variables. The following structure describes the body of the set-data-variable action:
struct ofl_exp_action_set_data_variable {
    struct ofl_exp_openstate_act_header   header; /* OFPAT_EXP_SET_DATA_VAR */
    uint16_t operand_types;
    uint8_t table_id;
    uint8_t opcode;
    uint8_t output;
    uint8_t operand_1;
    uint8_t operand_2;
    uint8_t operand_3;
    uint8_t operand_4;
    int8_t coeff_1;
    int8_t coeff_2;
    int8_t coeff_3;
    int8_t coeff_4;
    uint32_t field_count;
    uint32_t fields[OFPSC_MAX_FIELD_COUNT];
};

The operand_types field is made of four 2-bits subfields to specify the type of each operand and one 1-bit subfield to specify the output type according to the following scheme:
operand_types=aabbccddee0000000 where aa=operand_1_type, bb=operand_2_type, cc=operand_3_type, dd=operand_4_type and e=output_type.

Each operand subfield can assume any constant value of enum ofp_exp_operand_types, while the last subfield can only assume 0/1 to specify that the output type is a flow/global data variable.

The opcode field specifies the operation code according to one of the following constants:

class ofp_exp_opcode {
    OPCODE_SUM = 0,
    OPCODE_SUB,
    OPCODE_MUL,
    OPCODE_DIV,
    OPCODE_AVG,
    OPCODE_VAR,
    OPCODE_EWMA,
    OPCODE_POLY_SUM
};
The output field contains the ID of the output variable which will store the operation result.

The actual number of non-ignored operands depends on the specific operation.

The 4 coeff_X fields contains the coefficients used by the POLY_SUM operation.

The field_count field specifies the number of fields provided in fields[], which contains the vector of Type Length Value (TLV) fields composing the key extractor to be used by this action.

### 3.2.2 Match fields

In addition to the BEBA Experimenter flow match fields for global state and flow state, we added 11 new experimenter match fields to allow flow table entries matching on the output of conditions evaluation, packet timestamp, a random numer and the length of the packet.

```c
enum oxm_exp_match_fields {
    OFPXMT_EXP_GLOBAL_STATE,      /* Global state */
    OFPXMT_EXP_STATE,             /* Flow State */
    OFPXMT_EXP_CONDITION0,        /* Condition 0 */
    OFPXMT_EXP_CONDITION1,        /* Condition 1 */
    OFPXMT_EXP_CONDITION2,        /* Condition 2 */
    OFPXMT_EXP_CONDITION3,        /* Condition 3 */
    OFPXMT_EXP_CONDITION4,        /* Condition 4 */
    OFPXMT_EXP_CONDITION5,        /* Condition 5 */
    OFPXMT_EXP_CONDITION6,        /* Condition 6 */
    OFPXMT_EXP_CONDITION7,        /* Condition 7 */
    OFPXMT_EXP_TIMESTAMP,         /* Timestamp */
    OFPXMT_EXP_RANDOM,            /* Random */
    OFPXMT_EXP_PKT_LEN            /* Packet length */
};
```

#### Condition match fields

The experimenter OXM_EXP_CONDITION* fields are used in the flow table in order to match on the result of the evaluation of each condition. It is a 1 bit field and it's not maskable.

```c
#define OXM_EXP_CONDITION0 OXM_HEADER     (0xFFFF, 2, 5)
#define OXM_EXP_CONDITION1 OXM_HEADER     (0xFFFF, 3, 5)
#define OXM_EXP_CONDITION2 OXM_HEADER     (0xFFFF, 4, 5)
```
#define OXM_EXPCONDITION3 OXM_HEADER (0xFFFF, 5, 5)
#define OXM_EXPCONDITION4 OXM_HEADER (0xFFFF, 6, 5)
#define OXM_EXPCONDITION5 OXM_HEADER (0xFFFF, 7, 5)
#define OXM_EXPCONDITION6 OXM_HEADER (0xFFFF, 8, 5)
#define OXM_EXPCONDITION7 OXM_HEADER (0xFFFF, 9, 5)

**Timestamap match field**

The experimenter OXM_EXP_TIMESTAMP field is used in the flow table in order to match on the timestamp of packet arrival, in ms.

It is a 32 bit field and it's not maskable.

#define OXM_EXP_TIMESTAMP OXM_HEADER (0xFFFF, 10, 8)

**Random match field**

The experimenter OXM_EXP_RANDOM field is used in the flow table in order to match on the timestamp of packet arrival, in ms.

It is a 16 bit field and it's not maskable.

#define OXM_EXP_RANDOM OXM_HEADER (0xFFFF, 11, 6)

**Packet length match field**

The experimenter OXM_EXP_PKT_LEN field is used in the flow table in order to match on the size of the packet, in bytes.

It is a 16 bit field and it's not maskable.

#define OXM_EXP_PKT_LEN OXM_HEADER (0xFFFF, 12, 6)
3.3 Experimenter statistics message

3.3.1 State statistics message

The State statistics message has been extended to report the entire flow context to the controller, rather than just the flow state.

The body for ofp_multipart_request and ofp_exp_state_stats_reply of type OFPMP_EXP_STATE_STATS described by the struct ofp_exp_state_stats_request in D2.1 have not been modified.

The reply to a OFPMP_EXP_STATE_STATS multipart request consists of an array of the following

/**Structure of a single state statistic*/

```c
struct ofp_exp_state_stats {
    uint16_t length;            /* Length of this entry. */
    uint8_t table_id;           /* ID of table flow came from. */
    uint8_t pad;
    uint32_t duration_sec;      /* Time state entry has been alive in secs. */
    uint32_t duration_nsec;     /* Time state entry has been alive in nsecs beyond duration_sec. */
    uint32_t field_count;       /* Number of extractor fields*/
    uint32_t fields[OFPSC_MAX_FIELD_COUNT]; /*Extractor fields*/
    struct ofp_exp_state_entry entry;
    uint16_t hard_rollback;
    uint16_t idle_rollback;
    uint32_t hard_timeout;      // [us]
    uint32_t idle_timeout;      // [us]
    uint8_t pad2[4];
};
```

OFP_ASSERT(sizeof(struct ofp_exp_state_stats) == 136);

/**Structure of a single state entry*/

```c
struct ofp_exp_state_entry{
    uint32_t key_len;
    uint8_t key[OFPSC_MAX_KEY_LEN];
    uint16_t state;
    uint8_t pad[2];
    uint32_t flow_data_var[OFPSC_MAX_FLOW_DATA_VAR_NUM];
};
```

OFP_ASSERT(sizeof(struct ofp_exp_state_entry) == 80);
If more than one entry matches the request, multiple ofp_exp_state_stats will be appended to the message. The length of a single ofp_exp_state_stats is stored in the length field, the table_id field specifies the source table, the field_count specifies the number of extractor field stored in the fields field and the entry field specifies the actual state entry: key, key_len, state and flow data variables.

3.4 Simple toy example

In this subsection we are going to present how to program a very simple application which allows a MAC address to send data packets for a limited time window. Despite this is a trivial application, it already contains the additional key features of the BEBA extended version:
- configuration of conditions
- match on result of conditions evaluation
- arithmetic operations between packet header fields and constant values

The initial set of messages to enable statefulness and configure the lookup and update extractors are exactly the same described in section 1.3.3 of deliverable D2.2.

In this application we are interested in keeping per-MAC states thus lookup and update scopes are configured with OXM_OF_ETH_SRC.

""" Set table 0 as stateful """
req = bebaparser.OFPExpMsgConfigureStatefulTable(
    datapath=datapath,
    table_id=0,
    stateful=1)
datapath.send_msg(req)

""" Set lookup extractor = {eth_src} """
req = bebaparser.OFPExpMsgKeyExtract(datapath=datapath,
    command=osproto.OFPSC_EXP_SET_L_EXTRACTOR,
    fields=[ofproto.OXM_OF_ETH_SRC],
    table_id=0)
datapath.send_msg(req)

""" Set update extractor = {eth_src} """
req = bebaparser.OFPExpMsgKeyExtract(datapath=datapath,
    command=osproto.OFPSC_EXP_SET_U_EXTRACTOR,
    fields=[ofproto.OXM_OF_ETH_SRC],
    table_id=0)
datapath.send_msg(req)

As soon as we receive the first data packet from a given MAC address, we want to forward traffic from that host for a limited time window (let’s say 5.5 seconds). The condition to be checked is “now() < LIMIT_TIMESTAMP ?”. In order to compute the last timestamp the host is allowed to send traffic, we need to compute LIMIT_TIMESTAMP=now()+TIME_WINDOW_SIZE.
We can use flow states to memorize if an host has already send any traffic, while flow data variables can be used to store LIMIT_TIMESTAMP which needs to be checked to forward/drop traffic from that host.

```python
match = ofp.parser.OFPMatch(state=0)
actions = [ofp.parser.OFPActionOutput(ofproto.OFPP_FLOOD),
          beba.parser.OFPExpActionSetState(state=1, table_id=0),
          beba.parser.OFPExpActionSetDataVariable(table_id=0,
          opcode=beba.proto.OPCODE_SUM, output_fd_id=0, operand_1_hf_id=1,
          operand_2_gd_id=3)]

self.add_flow(datapath=datapath,
              table_id=0,
              priority=0,
              match=match,
              actions=actions)
```

This flow entry will be matched by the first packet sent from a given host (state=0) and will execute 3 actions:
1. Flood the packet (we are broadcasting the packet just to simplify the code)
2. Update the flow state to 1 (so that the switch can learn that such host has already sent some traffic)
3. Compute the last time instant that such host is allowed to send traffic.

The information LIMIT_TIMESTAMP is computed with a `OFPExpActionSetDataVariable()` action and it is stored in the flow data variable #0 (output_fd_id=0).

This variable is computed by summing (opcode=OPCODE_SUM) header field extractor #1 (operand_1_hf_id=1) with global data variable #3 (operand_2_gd_id=3). Before installing such a flow we need to configure the header field extractor so that it will extract the packet timestamp and the global data variable so that it will contain the TIME_WINDOW_SIZE of 5.5 seconds. These 2 configurations can be achieved with two new messages:

```python
req = beba.parser.OFPExpMsgHeaderFieldExtract(
    datapath=datapath,
    table_id=0,
    extractor_id=1,
    field=beba.proto.OXM_EXP_TIMESTAMP
)
datapath.send_msg(req)

req = beba.parser.OFPExpMsgsSetGlobalDataVariable(
    datapath=datapath,
    table_id=0,
    operation=beba.proto.OPG_DM_SET,
    global_data_variable=beba.proto.GVP_TIME,
    value=5.5
)
datapath.send_msg(req)
```
Every time we receive a packet from a known host (state!=0), we need to check if the host is still allowed to send traffic, e.g. if the time window has not expired. Such a condition can be configured as “now() < LIMIT_TIMESTAMP?”, which can be represented as “HF[1] < FDV[0] ?”.

With the following message we are configuring a “<” condition (condition=CONDITION_LT) between header field extractor #1 (operand_1_hf_id=1), which will extract the current timestamp of the packet at condition evaluation time, and flow data variable #0 (operand_2_fd_id=0), which will contain the LIMIT_TIMESTAMP which has been computed when that host sent packets for the first time.

```
req = bebaparser.OFPExpMsgSetCondition(
    datapath=datapath,
    table_id=0,
    condition_id=3,
    condition=osproto.CONDITION_LT,
    operand_1_hf_id=1,
    operand_2_fd_id=0
)
```

datapath.send_msg(req)

Once the condition has been configured, we can install flow entries matching on the result of the evaluation of such condition. The condition will be evaluated for each packet arrival.

```
match = ofparser.OFPMatch(state=1,condition3=1)
actions = [ofparser.OFPActionOutput(ofproto.OFPP_FLOOD)]
self.add_flow(datapath=datapath,
    table_id=0,
    priority=0,
    match=match,
    actions=actions)
```

```
match = ofparser.OFPMatch(state=1,condition3=0)
actions = [bebaparser.OFPExpActionSetState(state=2, table_id=0)]
self.add_flow(datapath=datapath,
    table_id=0,
    priority=0,
    match=match,
    actions=actions)
```

As soon as condition is violated (results is false, so condition3=0), we can trigger a state transition to state 2 so that this host is marked as “blocked” without any further conditions evaluations.
match = ofparser.OFPMatch(state=2)
actions = []
self.add_flow(datapath=datapath,
              table_id=0,
              priority=0,
              match=match,
              actions=actions)

The resulting FSM is the following

![StateMachine](image)

and this is the actual implementation my means of state, header field extractor, flow data variables and global data variables.
4 New perspectives for use case implementation

In this section we focus on application cases supported by the BEBA full XFSM abstraction. The first use case is a simple port scan detection and mitigation meant to provide a simple analysis of the new programming facilities of the proposed abstraction. The second and third use cases provide two more complex examples in the field of traffic shaping for QoS scenarios and network traffic monitoring.

This section is not meant to provide an exhaustive and detailed presentation of possible novel use cases. Any further details will be described in future WP5 deliverables.

4.1.1 Monitoring in BEBA networks

The extended BEBA API and the revised architecture of the BEBA switch provides abstraction for certain monitoring applications, such as the SYN flood mitigation (a part of the DDOS mitigation use case). For other applications, such as the programmable network flow measurement, BEBA already provides the most important capabilities, such as customized computations for each packet, flow states, flow counters, and in switch packet generation. However, some specific features are missing, e.g. actions triggered by inactivity of a flow or when the memory capacity is not sufficient. In this section, we revise the programmable network flow measurement use case from D5.1 with the focus on what can be implemented with the current BEBA API.

The programmable network flow measurement deals with advanced passive measurement of network flows, in-switch aggregation of information into flow records, and export of flow records using standard protocols. BEBA API can be utilized to create monitoring probes, possibly along with the “normal” switching operation.

Upon receiving a packet, the monitoring node finds the matching flow record in its memory, computes the defined formulas, and updates the flow record. The monitoring process also defines the way to export flow records to a designated location, e.g. via the IPFIX protocol. For the programmable network flow measurement use case, it is necessary to define triggers that activate the transmission of the flow records to the collector.

Active timeout occurs when a flow lifespan is longer than a predefined time period. Consider a long standing remote session. Such session might last for hours or even days. The monitoring use case has to be able to signal that such flow is present in the network. When active timeout occurs, the flow record is exported and deleted. When a new packet of the flow arrives in the future, the monitoring node creates a new empty record. All records belonging to the same flow can be correlated at the flow collector.

Inactive timeout occurs when a flow is inactive (not sending any packets) for the defined time period. When the timeout fires, the flow record of the flow is exported to the collector and deleted from the monitoring node. Figure 3 shows the conceptual scheme of the monitoring chain.
The extended BEBA API allows to define specific variables for each flow. These variables can be used for storing metadata information about flows, such as the number of packets and bytes in the flow, the presence of specific TCP flags etc. The generic functions that can be configured to the switch by the controller can define custom flow update operations tailored for the specific network.

The extended BEBA API also allows to implement active timeouts. BEBA time outs can trigger a custom action. Such action can be the in-switch packet generation action. The BEBA node can propagate the flow record to the generated packet.

The passive timeout can be implemented by a forced shift to the rollback state after the last packets as specified in D3.2. For sound monitoring, it is not acceptable to discard data when the FSM shifts to the rollback state. Hence, it is desirable to allow a BEBA node to start a specific action in when reaching the rollback state. The monitoring BEBA switch should propagate the flow record content to the IPFIX collector through the in-switch packet generation action.

Flow table exhaustion can result in a collision of hash functions used to identify flow records for each monitored flow. BEBA switch currently detects such situation and replaces the information about the previously stored flows with data for the new flow. However, currently the switch does not signal such collision to a controller and it is not possible to trigger in-packet switch generation when a collision occurs. Hence, in the case of the programmable network flow...
measurement, the old flow record would be deleted and the monitoring system would miss the flow. Such behavior can allow an attacker to hide traces about their activity. This has to be prevented in production network and it should be possible to define specific action (such as in-switch packet generation) to handle flow collisions.

4.1.2 A simple port scan detection and mitigation

To understand which new primitives can be programmed in the BEBA full XFSM data plane let’s walk through a simplistic example of a (quite inefficient, but at least trivial to follow) TCP port scan detection and mitigation application. Since the target is to detect IP addresses which behave as scanners, we use as Flow Key the IP address. Figure 4 represents our desired application’s behavior, expressed in the form of an XFSM “program”, whereas Figure 5 provides a corresponding tabular configuration delivered to the switch. For every IP packet, we check in the Flow Context Table whether the IP source has an associated context; if this is not the case, a DEFAULT state is conventionally returned. The XFSM table now checks whether the packet is a TCP SYN, and only in this case we will allocate a Flow Context Table entry for the considered IP source, and we will set it in MONITOR state. In this state, we measure the rate of new TCP SYN arrivals toward hosts behind the switch port 1.

Such rate (computed with the EWMA update function) is stored and updated in the flow registers $R_0$. While in MONITOR state, the value of $R_0$ is verified for each new TCP flow. If a given threshold (say 20 SYN/s, a value stored in the global register $G_0$) is exceeded, the state associated to this flow is set to DROP and all packets from this IP address are discarded. Suppose now that the programmer wants to block the scanner for 5 seconds. Lacking explicit timers (a non-trivial HW extension), such mechanism is realised by the following procedure:
1. when the flow state transits from MONITOR to DROP the register $R_I$ is set to the packet time stamp value plus 5 sec. (a value stored in the global register $G_1$);

2. in MONITOR state the $R_I$ value is checked for every received packet;

3. If $R_I \leq pkt.ts$ the flow state is reverted to MONITOR. Moreover, the application needs to store the last packet timestamp in the flow register $R_2$. This time stamp will be used by the EWMA update function.

To implement this application, the BEBA switch is configured by the controller using the BEBA protocol that performs the following elementary operations:

i) the DEFAULT, MONITOR and DROP states are encoded with 0, 1 and 2 respectively;

ii) the lookup and update scopes are set to ip.src;

iii) the global registers are defined as follows:

$G_0 = 20$

$G_1 = 5[s]$

iv) the condition table is configured to include two conditions:

$C_0 : R_0 \geq G_0$

$C_1 : R_1 > pkt.ts$ (packet arrival time)

v) the XFSM table is configured as in Figure 5

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>state</th>
<th>packet fields</th>
<th>next state</th>
<th>packet actions</th>
<th>update functions</th>
</tr>
</thead>
</table>
| *  | *  | 0     | syn=1         | 1          | OUT           | R0=0
|    |    |       |               |            |                | R2=pkt.ts       |
| 0  | *  | 1     | syn=1         | 1          | OUT           | R0=EWMA(R0, R2, pkt.ts);
|    |    |       |               |            |                | R2=pkt.ts       |
| 1  | *  | 1     | syn=1         | 2          | DROP          | R1=pkt.ts + G1  |
| *  | 1  | 2     | *             | 2          | DROP          | R0=0
|    |    |       |               |            |                | R2=pkt.ts       |
| *  | 0  | 2     | *             | 1          | OUT           | R0=0
|    |    |       |               |            |                | R2=pkt.ts       |

*Figure 5 XFSM table for the port scan use case*
### 4.2 Traffic policing with token buckets

In this use case we have implemented in BEBA advanced a single rate token bucket with burst size $B$ and token rate $1/Q$, where $Q$ is the token inter arrival time. Since in the BEBA advanced architecture the update functions are performed after the condition verification, we cannot update the number of tokens in the bucket based on packet arrival time before evaluating the condition (token availability) for packet forwarding. For this reason, we have implemented an alternative and equivalent algorithm based on a time window.

For each flow a time window $W$ ($T_{min}$ - $T_{max}$) of length $BQ$ is maintained to represent the availability times of the tokens in the bucket. At each packet arrival, if arrival time $T_0$ is within $W$ (Case 1), at least one token is available and the bucket is not full, so we shift $W$ by $Q$ to the right and forward packet. If the arrival time is after $T_{max}$ (Case 2), the bucket is full, so packet is forwarded and $W$ is moved to the right to reflect that $B-1$ tokens are now available ($T_{min}=T_0-(B-1)*Q$ and $T_{max}=T_0+Q$). Finally, if the packet is received before $T_{min}$ (Case 3), no token is available, therefore $W$ is left unchanged and the packet is dropped.

In the BEBA advanced implementation, upon receipt of the first flow packet, we make a state transition in which we initialize the two registers: $T_{min}=T_0-(B-1)*Q$ and $T_{max}=T_0+Q$ (initialization with full bucket).
At each subsequent packet arrival, we verify two conditions:

- \( C_0: \) \( T_{\text{now}} \geq T_{\text{min}} \)
- \( C_1: \) \( T_{\text{now}} \leq T_{\text{max}} \)

The three cases defined by the algorithm can be easily identified with these two conditions:

1. \( C_0 = \text{True} \) AND \( C_1 = \text{True} \)
2. \( C_0 = \text{True} \) AND \( C_1 = \text{False} \)
3. \( C_0 = \text{False} \)

The XFSM is shown in Figure 7

In Figure 6 we show the temporal evolution of the input traffic, output traffic and bucket level for a simple token bucket with \( B=10 \) and \( Q=1s \).

### Table

<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>state</th>
<th>packet fields</th>
<th>next state</th>
<th>packet actions</th>
<th>update functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>0</td>
<td>eth.type=IP</td>
<td>1</td>
<td>OUT</td>
<td>( R_0=\text{now} - G_0 ) ( R_1=\text{now} + G_1 )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>eth.type=IP</td>
<td>1</td>
<td>OUT</td>
<td>( R_0= R_0+G_1 ) ( R_1= R_0+G_1 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>eth.type=IP</td>
<td>1</td>
<td>OUT</td>
<td>( R_0= \text{now} - G_0 ) ( R_1= \text{now} + G_1 )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>eth.type=IP</td>
<td>1</td>
<td>DROP</td>
<td>( R_0= \text{now} - G_0 ) ( R_1= \text{now} + G_1 )</td>
</tr>
</tbody>
</table>

*Figure 7 Token bucket XFSM. The flow registers $R_0$, $R_1$ are used to store respectively $T_{\text{min}}$, $T_{\text{max}}$. The global registers $G_0$, $G_1$ are used to store $B \times Q$ and $Q$. The extractors are ip.src*

### 4.3 Decision tree based traffic classification

Machine Learning (ML) tools are widely adopted by the networking community for detecting anomalies and classifying traffic patterns [31]. We have tested the feasibility of using BEBA advanced to support this kind of traffic monitoring schemes by implementing a decision tree supervised classifier based on the C4.5 algorithm [32] that has been exploited by different work on ML based network traffic classification [33,34,35,36,37,38].
Any ML based classification mechanism has two phases: a training phase and a test phase. The training phase is off line and used to create the classification model by feeding the algorithm with labelled data that associate a measured traffic feature vector to one of \( n \) decision classes. In the case of decision tree based ML algorithms, the output of such phase is the binary classification tree. The training phase must obviously be performed outside the switch. For our use case implementation, the decision rules have been created using the Orange data mining framework [39], and a feature set proposed in [40]. We considered a simple scenario where it is necessary to discriminate between WEB and P2P (control) traffic. The selected features for each flow are: packet size average/variance and total number of received bytes. These features are mapped directly to the per flow memory registers R1, R2, R3. Moreover, the application XFSM requires two additional registers: R0 (packet counter) and R4 (measurement window expiration time). The input feature vectors are evaluated over a time window of 10 seconds.

The test phase, which is performed online, consists of two operations:

1. for each flow the feature set described above is computed;
2. after 10 seconds a decision is made according to the decision tree.

This testing mechanism is implemented in BEBA advanced according to the XFSM shown in Figure 8. The XFSM flows states are encoded as follows:

- **State 0** \( \rightarrow \) default
- **State 1** \( \rightarrow \) measurement and decision
- **State 2** \( \rightarrow \) WEB traffic (DSCP class AF11)
- **State 3** \( \rightarrow \) P2P traffic (DSCP class best effort)

The condition set is:

- **C0**: \( \text{now} > R4 \)
- **C1**: \( R2 > G2 \)
- **C2**: \( R3 > G3 \)
- **C3**: \( R1 \leq G1 \)

According to our simplified training phase, the (ceiled) thresholds values are: \( G1 = 306; G2 = 1575 \) and \( G3 = 203 \).
<table>
<thead>
<tr>
<th>C0</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>state</th>
<th>next state</th>
<th>packet actions</th>
<th>update functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>1</td>
<td>Fwd()</td>
<td>R4=now() + G0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>var(R0,R1,R2,pkt.len)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R3 = R3 + pkt.len</td>
</tr>
<tr>
<td>0</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>1</td>
<td></td>
<td>Fwd()</td>
<td>var(R0,R1,R2,pkt.len)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R3 = R3 + pkt.len</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>*</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>Fwd(),SetField(dscp=0)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>*</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>Fwd(),SetField(dscp=10)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>*</td>
<td>1</td>
<td>2</td>
<td>Fwd(),SetField(dscp=10)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>*</td>
<td>1</td>
<td>3</td>
<td>Fwd(),SetField(dscp=0)</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>2</td>
<td></td>
<td>Fwd(),SetField(dscp=10)</td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>*</td>
<td>*</td>
<td>3</td>
<td></td>
<td>Fwd(),SetField(dscp=0)</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 8 XFSM table for the traffic classification use case*
5 Preliminary results for a BEBA High level language

5.1 Building High-level Programming Abstractions

In this section we present the evolution of the basic BEBA SDN framework, as presented in D2.2, from the development point of view. As introduced in D4.2, in order to turn BEBA into a highly competitive candidate approach for network operators and SDN adopters, there is a gap we need to fill. That is to provide the means to develop BEBA applications in an intuitive and expressive way that abstracts the underlying technical details of the hosting platform from the target dataplane functionality.

In D4.2, we performed an extensive literature study in the area of SDN programming abstractions to identify relevant ideas and candidate solutions. In this study, we found Kinetic [22], the state-of-the-art SDN framework that follows a very similar approach with BEBA. Specifically, Kinetic models a dataplane task as an FSM and introduces a high-level API to describe the FSM using the Python programming language in a controller-agnostic manner. Inspired by this approach, in D4.2 we paved the way for evolving the BEBA SDN framework following a Kinetic approach.

The current form of the BEBA basic API heavily relies on the underlying controller (i.e., Ryu [23]). This means that the programmer needs to know which Ryu libraries are important to import into the BEBA application, as well as the way to call their methods. The target of this contribution is to hide this information from the developer and let her focus on describing the functionality and stateful behavior of the BEBA tasks.

To this end, in the following sections we present the new form of the basic BEBA API. In Section 5.2 we introduce the new concepts of the programming abstraction. In Section 5.3 we apply these concepts to real applications, while in Section 5.5 we discuss how the proposed abstractions can be extended to cover the new extended BEBA API.

5.2 Programming Abstractions in the Basic BEBA API

The main pillar of the basic API’s programming abstraction is a Python class that models the behavior of a BEBA switch, called AbstractBEBANode. This class takes as inputs the switch datapath identifier (DPID as seen by the controller) and the number of ports that this switch has, as follows:

```
node = AbstractBEBANode(switch_id=<DPID>, number_of_ports=N)
```

Internally, the abstract BEBA node keeps a dictionary of FSMs, each mapped to a different table identifier. This data structure makes sure that each FSM takes its own dedicated memory pool and that a switch can run multiple FSMs at the same time. Initially, the dictionary is empty and one can add an FSM by calling the method:
This method instantiates an object of the class FSM that requires three inputs parameters. The first parameter is the table identifier (i.e., table_id) where the FSM will be installed. This table identifier is the key of the FSM dictionary described above, whereas the generated "fsm" object is the value. Secondly, as described in D2.1, to create a BEBA FSM we need two keys, the lookup and update scope keys. These keys are lists of one or more header fields, used to match incoming packets and either identify the required state and action (lookup scope key), or update the state table respectively. The `add_fsm` method finally returns the newly created FSM instance. Having this instance available, a BEBA developer can encode the desired functionality by adding states and transitions to the FSM. A state can be added by calling the method:

```python
fsm.add_state(state, timeout)
```

This method adds the state “state” in the set of states of this FSM, as well as a timeout that denotes the life span of this state in the state table. Upon expiration of this timeout, the switch will flush the state from the table. Next, the developer can create a state transition by calling the method:

```python
fsm.add_transition(state, key, next_state, action, priority)
```

This method looks up for the state “state” in the set of states of this FSM. If the state exists, it is associated with the pair of values (key, next_state), otherwise the `add_state` method above is internally called to create state “state” before realizing the transition. This mapping captures the evolution of the FSM upon a condition. In particular, if the incoming flow’s header matches the key “key” and its current state is “state”, then the flow’s new state will be “next_state”.

One obscure detail when writing SDN applications is the way to build header matches. To make it clear, a developer that uses the Ryu controller can match an input UDP flow against the destination port 53 of the UDP header using the following expression:

```python
OFPMatch(eth_type=ETH_TYPE_IP, ip_proto=inet.IPPROTO_UDP, udp_dst=53)
```

However, the same match is accomplished by the POX controller using the expression:

```python
ofp_match(dl_type=pkt.ethernet.IP_TYPE,nw_proto=pkt.ipv4.UDP_PROTOCOL,tp_dst=53)
```

We propose a unified approach for matching flows in BEBA by following the Kinetic paradigm. Specifically, we implemented a match method that takes zero or more header fields associated with their values and takes care of turning these fields into controller specific commands. To realize the above match with our method, one has to write:

```python
match(eth_type=ETH_TYPE_IP, ip_proto=IPPROTO_UDP, udp_dst=53)
```
Note that, this approach offers a unified north-bound API that can be trivially translated to different controllers’ commands by simply adding a new south-bound parser for this high-level language.

Additionally, a complete SDN paradigm requires also actions for that flow; hence the parameter “action” is also associated with “state” and kept to a different dictionary. Our abstraction simplifies the developer’s life by converting high-level commands such as Flood, Drop, fwd(port_number), modify(field=value) into controller specific macros, used to take actions. For example, to specify the flood action in Ryu, one has to call the macro ofparser.OFPActionOutput(ofproto.OFPP_FLOOD). Using the BEBA abstraction this long macro is hidden and the developer can simple write Flood. Finally, the developer can optionally specify a particular priority for this rule; otherwise the FSM class assigns a default priority value.

Once all the states and transitions are added into the FSM instance, the developer can test the FSM functionality by calling:

node.apply()

This method starts a Ryu process and passes the FSM description as a set of Ryu specific commands to be executed. The FSM is started as an individual thread of the Ryu process, hence one controller instance can run multiple FSMs at the same time.

The following section utilizes the above-described programming model to code two of the BEBA use cases.

5.3 Example BEBA Applications
To highlight the simplicity of the new abstract BEBA API we implemented two “middlebox-type” use cases from D5.1. In Section 5.4, we begin from the simplest example in that deliverable, the MAC learning use case. Then, in the appendix we move to a more complex use case, namely the Forwarding Consistency use case (UC06). The goal is to compare the implementation of each use case with and without using the proposed programming abstractions and assess the practicality of the developed abstractions, as well as the relative simplification of the developer’s labor.

5.4 MAC Learning Use case (UC01)
The state machine of the MAC Learning use case is depicted in Figure 9. To realize this FSM, the original MAC learning implementation, which relies on the Ryu BEBA controller, is shown in Figure 10. To fit the code in one page we omitted the initial calls to the Ryu OpenFlow libraries, as well as the definition of the global variable N that denotes the number of ports of the target BEBA switch. This variable takes the value 4 in our case.
As you can see in Figure 10, the use case functionality is mostly implemented in the method `switch_features_handler`, where the first 4 commands setup the state table of the switch and the two keys, namely the lookup and update scope keys. Then, in the double loop, the controller sends flow mod commands that define the state transitions for each input port of the switch. The actions applied are (i) to associate an input port with a state number and (ii) forward the packet to that port (which is now learned). If the state of the incoming packet is 0, the switch simply floods the packet because it has not learned the port yet. It is obvious that even a simple use case such as the MAC learning involves a large amount of controller-specific macros that hinder the readability of the code and lead to noisy FSM representations. This was the initial motivation for us and the reason why we put the effort to make the BEBA development process easier and more appealing to the developer.

We juxtapose the implementation shown in Figure 10 with the equivalent implementation of Figure 11. The latter uses the abstract API we introduced above to achieve the very same functionality. First, we create a new BEBA node instance by calling the `AbstractBEBANode` class. In order to setup the state table, it is enough to invoke one command (`add_fsm`) as opposed to the three commands required with the original API. This command takes the two keys as input parameters and realizes the stateful table configuration internally. Then the same double loop is executed, but instead of writing Ryu-specific matches, actions, and flow modification commands we simply add states and transitions using the `add_state` and `add_transition` commands. These functions wrap aforementioned commands. Finally, we push the FSM down to the switch by calling the apply method of the `AbstractBEBANode` class. Apparently, the code of Figure 11 is much simpler and concise compared to the code depicted in Figure 10, even if the simplicity of this use case does not fully highlight the “savings” and benefits we can achieve. Essentially, the code comprises of 4 different function calls highlighted with bold: (i) the instantiation of the node’s functionality in the controller, (ii) the addition of states, (iii) transitions, and (iv) the deployment in the target switch.
class `BEBAMacLearning`(app_manager.RyuApp):
    
def __init__(self, *args, **kwargs):
        super(BEBAMacLearning, self).__init__(*args, **kwargs)

def add_flow(self, datapath, table_id, priority, match, actions):
        
        inst = [ofparser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS, actions)]
        if len(actions) > 0 else []
        mod = ofparser.OFPFlowMod(datapath=datapath, table_id=table_id,
                                  priority=priority, match=match, instructions=inst)
        datapath.send_msg(mod)

@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, event):
    
    msg = event.msg
    datapath = msg.datapath

    
    req = bebaparser.OFPExpMsgConfigureStatefulTable(  
              datapath=datapath,  
              table_id=0,  
              stateful=1)  
    datapath.send_msg(req)

    req = bebaparser.OFPExpMsgKeyExtract(datapath=datapath,  
                                          command=bebaproto.OFPSC_EXP_SET_L_EXTRACTOR,  
                                          fields=[ofproto.OXM_OF_ETH_DST],  
                                          table_id=0)
    datapath.send_msg(req)

    req = bebaparser.OFPExpMsgKeyExtract(datapath=datapath,  
                                          command=bebaproto.OFPSC_EXP_SET_U_EXTRACTOR,  
                                          fields=[ofproto.OXM_OF_ETH_SRC],  
                                          table_id=0)
    datapath.send_msg(req)

    for i in range(1, N+1):
        for s in range(N+1):
            match = ofparser.OFPMatch(in_port=i, state=s)
            out_port = ofproto.OFPP_FLOOD if s == 0 else s

            actions = [bebaparser.OFPExpActionSetState(state=i, table_id=0, hard_timeout=10),
                       ofproto.OFPActionOutput(out_port)]
            self.add_flow(datapath=datapath, table_id=0, priority=0,
                          match=match, actions=actions)
Figure 11: BEBA MAC learning (UC01) implementation using the new abstract API.
5.5 Preliminary analysis for a programming abstractions in the Extended BEBA API

This deliverable introduced the extended BEBA API in Sections 1/2/3. Although the programming abstractions we introduced afore are applied to the basic BEBA API, we believe that it would be trivial to extend this generic wrapper in order to cover the new extensions.

Since the new API is backwards compatible with the basic BEBA API, the most important extension required in the AbstractBEBANode class is to extend the add_state and add_transition methods with extra arguments that will describe the conditions of the new XFSM BEBA model. These extensions require one additional data structure in the AbstractBEBANode class that keeps a mapping between the states and their conditions. We leave this task as a future work.
6 Discussions

6.1 Proposed abstraction: at which “level”?

Our abstraction resides at a very low level, as in practice it specifies a machine language which directly configures the hardware. In this, albeit technically very different, we are quite similar in spirit to the original OpenFlow abstraction, envisioned as an abstraction of the switch’s Flow table component. Independence of the underlying platform is therefore not accomplished via a compiler, such as in the case of emerging data plane programming languages such as P4 [25], but it is accomplished by decoupling the actual identification and (XFSM-based) combination of the hardware primitives from how they are implemented inside the switch (the proposed BEBA full XFSM hardware design being a possible implementation, but not nearly the unique). As long as two platforms expose the same set of (header) matching facilities, forwarding actions, enabling functions for evaluating conditions, and a same instruction set for the update functions, and as long as they do support the XFSM state transition execution logic, the application’s description in terms of XFSM can be ported across platforms. Loosely speaking, our abstraction is closer to a “bytecode” or to an assembly language rather than to a practical programming language. While we believe this is a strength of our proposal, we also clearly recognize the importance of offering to the applications’ developers a higher level and more user-friendly programming language (P4 being an obvious candidate, see next discussion) and relevant compilers which transform a higher level description into the proposed BEBA XFSM-based machine language.

6.2 Use of P4 for the description of BEBA advanced stages.

Even if the details of the reference P4 hardware architecture is not publicly available, the language has the goal of being expressive enough to describe any configurable architecture for packet forwarding. Therefore, even if out of the scope of this codument, and for just the purpose to stimulate possible discussion (i.e. we don’t claim this document to provide any specific contribution in this direction), we nevertheless made our own preliminary attempt to understand how the proposed BEBA advanced architecture could be described using P4, i.e. if BEBA advanced could be used as a further platform’s target for a P4 program. To this purpose, we have defined a reusable BEBA_advanced.p4 library. From such experience, we gathered a twofold impression. On one side, the current version of the language already permits to describe/support key BEBA advanced functionalities. Moreover, the ability to store data in persistent registries permits to properly describe, using P4, the “computational loop” characterizing BEBA architecture. On the other side, we suffered from the lack, among the P4 constructs, of an explicit state/context table and a relevant clean way to store and access per-flow data. In our BEBA_advanced.p4 library, a table functionally equivalent to our Context table was actually constructed by combining arrays of registers with hash keys generators which are provided as P4 language primitives. However, besides the obvious stretch (P4 registers are generic, and not specifically meant to be deployed on a per-flow basis), this
construction also suffers from hash collisions, a non-trivial problem if constrained to be addressed while the packet is flying through the pipeline. The availability of a tailored context/state table structure in P4 would greatly simplify the support of BEBA advanced as the target platform for P4.

### 6.2.1 BEBA_advanced.p4 library

The P4 language (Programming Protocol-independent Packet Processors) [50] is an open source language which allows the specification of packet processing logic with. P4 is protocol and target independent. A forwarding element is represented by the following P4 abstract model:

The parser can be configured by specifying the header format of packets the switch will deal with. The ingress/egress pipeline of match+action tables is configured by specifying the available set of matching fields, the type of match and the permitted actions. Custom actions can be defined by the user and built by combining packet processing primitives.

The user must also configure the “Control flow” which specifies the imperative description of the table application order.

A forwarding device is configured in two steps. In the first step, at boot time, the user has to define the specification of the parser, match+action tables and the control flow through the pipelines. Such a configuration is provided as a P4 program which is compiled to a specific target.
The output of this first phase is the specific target configuration and a set of API which can be used by a controller in the second phase, at run-time, to actually populate pipeline’s tables with flow entries.

P4 language offers a set of stateful memories (counters, meters and registers) to maintain a state across subsequent packets. In particular registers are memories which can be read and written by an action.

We tried to understand how the BEBA advanced architecture could be described using P4. By combining arrays of registers with hash keys generators we are able to obtain a table functionally equivalent to our Context table: such a table can be built using an array of registers to store the state and the flow data variables. Hash keys generators produce the index to access those registers by computing a hash function on a user-defined set of fields (the lookup/update scopes).

In addition to packet header fields, P4 defines metadata as additional pseudo-header fields available in the pipeline for matching or moving state between tables or providing informations such as ingress port or a time stamp. The main difference from standard header fields is that they are implicitly removed before the packet leaves the switch.

We created a library, BEBA_advanced.p4, which can be included by another P4 program to easily express stateful packet processing. After packet headers have been parsed, a metadata field with the default context (state, flow data variables, global data variables and conditions all set to 0) is added. The first table (lookup context table), defined in the library, loads the corresponding flow context (state and flow data variables) and the global data variables into the packet metadata fields. Conditions are specified and evaluated in the control flow (as shown in figure) and their result is written into packet metadata fields. The second table (the actual XFSM table), defined by the user, matches on state, conditions and other user-specified fields and executes user-defined actions (built by combining P4 primitives) which eventually update state, flow data variables, global data variables and forward or drop the packet.

### METADATA

We defined this metadata which allows user-defined flow table to match on the flow context lookup operation. In addition to the state field, there are m+1 fields for the flow data variables, p+1 fields for the global data variables and n+1 fields for the conditions. 

\[ \text{STATE\_MAP\_SIZE} = \log_2 \text{of context table’s size} \] and is a user-defined value.

```pseudocode
header_type BEBA_t {
    fields {
        lookup_state_index : STATE_MAP_SIZE; // state map index
        update_state_index : STATE_MAP_SIZE; // state map index
    }
}
```
state : 16; // state
    R0: 32;
    R1: 32;
    R2: 32;
    Rm: 32;
    G0: 32;
    G1: 32;
    G2: 32;
    Gp: 32;
    c0:1;
    c1:1;
    c2:1;
    cn:1;
}
}
metadata BEBA_t beba;

**CONTEXT LOOKUP TABLE**

This table has a default action which computes the hash of user-defined header fields to create the index to access the registers and append the flow context (state and flow data variables) to the packet by setting the corresponding metadata fields. Global registers metadata fields are set by reading the corresponding registers, which are not flow-dependent. Finally, conditions metadata fields are left with value 0. `CONTEXT_TABLE_SIZE` is the user-defined size of the context table (it must be $2^{\text{STATE_MAP_SIZE}}$).

```c
action lookup_context_table() {
    //store the new hash value used for the lookup
    modify_field_with_hash_based_offset(beba.lookup_state_index, 0, l_hash, CONTEXT_TABLE_SIZE);
    //Using the new hash, we perform the lookup reading the reg_state[idx]
    register_read(beba.state, reg_state, beba.lookup_state_index);

    register_read(beba.R0, reg_R0, beba.lookup_state_index);
    register_read(beba.R1, reg_R1, beba.lookup_state_index);
    register_read(beba.R2, reg_R2, beba.lookup_state_index);
    register_read(beba.R3, reg_R3, beba.lookup_state_index);

    register_read(beba.G0, reg_G, 0);
    register_read(beba.G1, reg_G, 1);
    register_read(beba.G2, reg_G, 2);
    register_read(beba.G3, reg_G, 3);
}
table context_lookup {
    actions {
    lookup_context_table;
    _nop;
    }
}
CONDITIONS

Conditions are defined and evaluated in the control flow. If X-th condition is evaluated as True, the corresponding set_cX_table is applied and thus cX metadata field is set to 1. If the condition is not met, cX metadata field is kept to the default value 0. For example, the table applied when 2-th condition is met is defined as

```c
action set_cond2_true() {
    modify_field(beba.c2, 1);
}
```
```c
table set_c2_true {
    actions {
        set_cond2_true;
    }
}
```

UPDATE FUNCTIONS

Since update functions are application-dependent and P4 does not allow to operate directly on registers, user has to define his custom flow table (matching on state and conditions metadata) with his custom actions operating on metadata fields. Finally, in order to store these modifications into registers, we can think about two different approaches:

1. Adding a table (defined in the library) at the end of the pipeline to update the state, flow data variables and global data variables into registers from metadata fields.
2. Ask the user to add a piece of code at the end of his custom actions to explicitly update the value of registers from metadata fields.

6.3 Structural limitations and possible extensions

While (we believe) very promising, our proposed approach is not free of structural concerns. If, on one side, limitations in the set of supported enabling functions and ALU functions for registry updates may be easily addressed with suitable extensions, and integration of more flexible packet header parsing (following [24]) is not expected to bring significant changes in the architecture, there are at least three pragmatic compromises which we took in the design, and which suggest future research directions. The first, and major one, resides in the fact that state transitions are “clocked” by packet arrivals: one and only one state transition associated to a flow can be triggered only if a packet of that flow arrives; asynchronous events, such as timers' expiration, are not supported. So far we have partially addressed this limitation with, on one side, the decoupling between lookup and update functions (the cross-flow state handling feature), and on the other side with programming tricks such as the handling of time performed while implementing the token bucket example. But further flexibility in this direction is a priority in our future research work. A second shortcoming is the deployment of ALU processing only in the Update Logic Block. This decision was done in favour of a cleaner abstraction and a simpler implementation. However, (programmable) arithmetic and logic
operations would be beneficial also while evaluate conditions (e.g., A − B > C) which, in the most general case, may require to be postponed to the next packet (the update function can store A − B in a registry, and the next condition can use such register). A third, minor, shortcoming relates to the fact that all updates occur in parallel. This prevents the programmer to pipeline operations, i.e. use (in the same transition step) the output of an instruction as input to a next one. While this issue is easily addressed by deploying multiple Update Logic Blocks in series, this would increase the latency of the BEBA advanced loop.
7 Future works

This document presented advanced abstractions and programming interfaces for the BEBA switch. While the presented work surely enhances the switch capabilities, as it happens in many cases, good abstractions are finally confirmed over time, through deployments and extensive use. The BEBA switch is likely to undergo a similar process, thus, we expect that, as new use cases are being implemented, the defined abstractions will require refinements and further improvements.

However, we envision a number of future directions for the line of work that was started by this project, which go beyond the natural evolution of the defined abstractions. We currently identify five directions that are only partially, or not at all, covered in the context of this project, but that we still consider of great interest.

7.1 Programmable ASICs

During the execution of the BEBA project a number of new initiatives were kicked off to allow the creation of programmable switching ASICs. The main and most concrete example in this direction is probably the work lead by the P4.org consortium. The consortium is defining a language, P4, which allows a programmer to define the switching pipeline the switch should implement. Notice that the target switch may be implemented in any technology, e.g., ASIC, FPGA, software, etc.

Since a BEBA switch XFSM abstraction introduces some new functions that are not typically offered by a switch pipeline, an interesting research direction is to understand whether such advanced functions can be provided on top of a more general programmable switch. For example, the BEBA switch requires multiple ALUs for supporting per-flow mathematical actions, is it possible to integrate the operations of such ALUs in a more general architecture of a programmable ASIC?

7.2 Hybrid software-hardware switches

In recent years a trend in embedding programmable hardware in general purpose servers has emerged. CPU companies like Intel announced integrated FPGA in their new generation processors; likewise, network interface card vendors are integrating FPGAs in their products. The integration of FPGAs is considered as a response to the inability of general purpose CPUs to grow further their performance as they did over the past decades. In fact, it has been observed that the Moore law may finally come to an end as the physical limits of the current CPU production process are hit.

The BEBA project has been exploring both software and hardware implementation of the BEBA switch. The two currently separated prototypes could indeed represent ideal building blocks to start exploring the opportunity to implement a hybrid software-hardware switch that could meet the performance expectations of future server deployments. In particular, while the
possibility of combining hardware and software forwarders for the implementation of a SDN switch has been explored for the case of “stateless” switches [41], a BEBA switch with its XFSM abstraction introduces some new relevant challenges. In fact, it has been already observed that the feedback loop introduced to perform state updates within the switch can bring forwarding consistency issues in specific cases. When moving to a more complex hardware-software implementation, while performance may increase, the consistency issues may be exacerbated. Exploring such challenges and providing relevant solutions is still an open research field.

Notice that similar solutions have been already proposed for the case of standard OpenFlow switches [102,103], but the stateful BEBA abstractions widen the problem space.

### 7.3 End-host stack abstractions

While during the BEBA project we focused on the implementation of a programmable (stateful) switch abstraction, we realized that the provided XFSM abstraction is suitable for even more complex use cases. For example, one could implement some typical Middlebox operations, e.g., session reconstruction, intrusion detection, etc.

However, supporting these advanced use cases requires the introduction of new types of forwarding actions and data structures. Identifying and implementing those is a complex design task, since it once more involves a process of abstraction and generalization of the offered functions, in order to enable a broader set of use cases beyond those that are already addressed today with ad hoc functions. In this direction, we have been already partially exploring the implementation of some well-known GNU/Linux interfaces such as Iptables, however, more complex functions, e.g., those that require TCP sessions handling, have been left out of scope.

This research direction, dealing with typical end-host functions, would also be focused mainly on software design, and it could actually re-shape the way an end-host’s network stack is implemented. Furthermore, this line of work would nicely fit in the research line that is considering implementing specialized network stacks to provide better performance and flexibility for the implementation of end-host network applications [44,45,46]. That is, one could even think to re-implement some network stack’s functions using the enhanced BEBA switch abstractions.

Another, more challenging objectives could actually even consider to offer the BEBA-based network stack to the control of a central entity. I.e., this would include end-hosts’ network stack in set of programmable devices a e.g., SDN controller, manages [47].

### 7.4 High-level programming abstractions

One of the issues highlighted during the advanced BEBA abstractions design is the complexity of programming the switch’s pipeline. This has been an observation made in the past to the
OpenFlow API as well, leading to the effort of defining more high-level languages to program the switches [48].

In BEBA, as in the standard OpenFlow case, the defined programming abstractions actually expose some of the pipeline implementation complexity, with the need to take into account different bits and pieces distributed within the switch implementation. Such complexity may be not avoidable when creating more expressive and powerful abstractions, however, there could be tools that help a programmer in reducing the complexity level, finally making the provided programming interfaces easier to use.

In the BEBA project, the activity of defining a higher level abstraction has been already started and some preliminary results have been presented in this document. However, the introduction of an XFSM abstraction and the definition of new advanced actions require further investigation. Also, the work so far focused on the handling of a single switch, while abstractions that could capture the definition of network-wide FSMs, i.e., those that are executed over more than one switch, have not been explored yet.

### 7.5 Composition and optimization

The BEBA project focused on the definition of programs to be expressed as FSM to be executed in a switch. As it usually happens in real networks, the same switch may be required to run different functions, i.e., different FSMs. In the current BEBA approach, the combination of different FSMs running on the same switch is performed manually. Furthermore, the current (strawman) approach for composition leverages the definition of multiple stateful stages, whose availability is guaranteed mainly just in software switches.

As future direction, it would be of great interest the exploration of the available options to combine several independent FSMs into a single BEBA switch’s stateful stage. Notice that similar problems were explored for standard OpenFlow switches and that the literature about FSM composition is also vast [49]. Still, the specific application of such techniques to the BEBA case is left unexplored so far.

Furthermore, once a composition of different FSMs has been generated and implemented on top of a single switch, there could be also a chance to run optimizations over the obtained switch’s configuration. For example, one could look into methods that reduce the amount of required entries in the switch’s flow tables. These optimizations would differ from traditional FSM optimization techniques, such as state minimization ones, in that the final FSM resulting from the optimization may be dynamically updated, e.g., because the switch is re-programmed or because a new FSM has to be added to the switch’s configuration.
APPENDIX Forwarding Consistency Use case (UC06)

In this appendix we briefly present the implementation of one more use case using the programming abstractions introduced in Section 5. We chose the Forwarding Consistency use case (UC06) that shows how a BEBA node can be effectively used as a consistent load balancer. We consider a scenario where the switch interconnects a client zone (port 0) with a set of 4 servers in a server zone, connected to ports 1-4 of the switch. When there is traffic heading to these servers (with destination IP address in 200.0.0/24), the switch decides the destination server to serve a flow based upon its source IP address.

The state machine of this use case is shown in Figure 12 while the implementation is depicted in Figure 13. We assume N=4 in the state machine figure, as there are 4 servers available in the server zone. The original implementation of this use case is located in the BEBA repository, however the code does not fit into one page, hence we decided to omit it from this deliverable.

The code in Figure 13 follows a similar structure with the MAC Learning implementation. The first transition occurs for flows heading to the client zone, hence the switch forwards to port 0. Then, the loop performs a round-robin assignment of IP addresses to servers using a modulo 4 (the number of available servers) operation. This leads the flows targeting 200.0.0.0/24 to transition to states 1-4, which are associated with the appropriate forwarding. Finally, we send the configuration to the switch by calling the apply method.

Figure 12: BEBA FSM for the Forwarding Consistency use case (UC06).
from ryu.abstraction.beba import AbstractBEBA

### Number of ports of the BEBA node under test

\[ N = 5 \]

### Create a BEBA node

\[
\text{node} = \text{AbstractBEBA}(\text{number}\_\text{of}\_\text{ports}=N)
\]

### Add an FSM to the node

\[
\text{fsm} = \text{node.add}\_\text{fsm}(\text{table}\_\text{id}=0, \\
\text{lookup}\_\text{scope}=\text{"ipv4}\_\text{src"}, \\
\text{update}\_\text{scope}=\text{"ipv4}\_\text{src"})
\]

### Packets coming from the server zone (IP prefix 200.0.0/24) should be forwarded to the client zone (port 0)

\[
\text{fsm.add}\_\text{transition}(\text{state}=0, \\
\text{key}=\text{match}(/ipv4\_\text{src}=200.0.0.0/24), \\
\text{next}\_\text{state}=0, \\
\text{action}=\text{fwd}(0), \\
\text{priority}=100)
\]

### Packets coming from the client zone and target the servers' subnet (200.0.0/24) are load-balanced in a round-robin fashion according to their source IP address

\[
\text{for counter in range}(1, 255): \\
\text{next}\_\text{state} = \text{counter} \text{ mod} 4 \text{ if next}\_\text{state} == 0 \text{ else next}\_\text{state}+1 \\
\text{fsm.add}\_\text{transition}(\text{state}=0, \\
\text{key}=\text{match}(/ipv4\_\text{dst}=200.0.0.0/24, \\
/ipv4\_\text{src}=192.168.0.**+str(counter)), \\
\text{next}\_\text{state}=\text{next}\_\text{state}, \\
\text{action}=\text{fwd}(\text{next}\_\text{state}), \\
\text{priority}=10)
\]

\[
\text{fsm.add}\_\text{transition}(\text{state}=1, \\
\text{key}=\text{match}(), \\
\text{next}\_\text{state}=1, \\
\text{action}=\text{fwd}(1), \\
\text{priority}=10)
\]

\[
\text{fsm.add}\_\text{transition}(\text{state}=2, \\
\text{key}=\text{match}(), \\
\text{next}\_\text{state}=1, \\
\text{action}=\text{fwd}(2), \\
\text{priority}=10)
\]

\[
\text{fsm.add}\_\text{transition}(\text{state}=3, \\
\text{key}=\text{match}(), \\
\text{next}\_\text{state}=1, \\
\text{action}=\text{fwd}(3), \\
\text{priority}=10)
\]

\[
\text{fsm.add}\_\text{transition}(\text{state}=4, \\
\text{key}=\text{match}(), \\
\text{next}\_\text{state}=1, \\
\text{action}=\text{fwd}(4), \\
\text{priority}=10)
\]

### Apply the FSM by launching the controller

\[
\text{node.apply()}
\]

---

**Figure 13: BEBA Forwarding Consistency (UC06) implementation using the new abstract API.**
References


BEBA software repository: https://bitbucket.org/beba-eu


