**BEHAVIORAL MODEL VERSION 2:**

Behavioral model is at its roots a software switch compiled by the industry experts to emulate the environment of a programmable switch.

It basically provides a target for execution of p4 program.

It ingests a JSON data file which consists of table entries for the target switch and the pipeline that is to be emulated is also present in the JSON data that is generated by the p4 compiler.

Main aim of behavioral model is to prove that network programming can be made hardware agnostic.

Bmv2 is currently supporting v1 model architecture which has the following pipeline:

Diagram

Description automatically generated

The V1Model consists of six P4 programmable components:

* Parser
* Checksum verification control block
* Ingress Match-Action processing control block
* Egress Match-Action processing control block
* Checksum update control block
* Deparser

The switches provided by behavioral model are chronologically ordered based on their recommended use.

It is a c++ user-space software switch to emulate p4 data plane.

**WORK FLOW OF BMV2:**

**Diagram

Description automatically generated**

Innovation in the field of networking is algorithms being shifted from hardcoded into the hardware to a more software approach where the algorithms can be sent as programs to the routers or be executed at a remote location and the results being sent back to the requesting router.

Bmv2 comes with simple\_switch target, which implements the P4 v1.0 / v1.1 abstract device model using the bmv2 library.

For running p4 programs on different targets provided by Bmv2 and those can be specified in the compilation command.

After that we need separate commands to run bmv2 switch independently and then run the mininet environment inside the bmv2 and check the working of the switch.

A file called runtime\_CLI is used to populate the tables in the switch and there are appropriate commands for it which can be found in the runtime\_CLI readme file.

A commands.txt file can be made which can be fed to the runtime\_CLI file and will populate the tables accordingly and the switch will forward the packets accordingly.

The above approach is for the configuration that is not provided as runtime json file.

**For proper functioning of your p4 program:**

**topology.json:** This file specifies the topology that is to be used for the p4 program. It consists of all the switches that are to be present along with the address of their runtime json file. Hosts and their mac address and ip address and interface information along with the commands that is used to initialize the networking information. This should be present in a folder named topo.

**Screen Shot of a topology.json file:**

**Text

Description automatically generated**

**Si-runtime.json:** This file contains basic build information one of them being the compiled P4 program json file route and other necessary information. Along with this it consists of table entries which are the most important part of our network implementation. In its skeleton, table entries consist of table name, action name depending on the packet headers which is also a part of this json file, action name this points to the action that is to be executed when an entry in the table matches a particular expression mentioned in the key section of the table.

**Screen Shot of a Runtime JSON File:**

**Text

Description automatically generated**

**Makefile:** This file helps in the compilation process which is basically a collection of compilation commands. It cuts unnecessary slack of writing multiple commands to run our p4 logic on bmv2 switch.

**<Filename>.p4:** The heart of our network program file which guides the switch on how to implement data plane.

Well, we know that it is a software emulated switch with no actual physical ethernet ports, so at the roots it is implemented just as any other process and the ethernet ports are virtual interfaces to which the operating system might instruct them to act as ports and to process packet via the P4 program.

As we are treating it as a real switch we can actually monitor the packets going to and from these virtual ethernet ports using **tcpdump or tshark commands.**

**BMV2 Mininet Environment:**

This is generated using Mininet python API. All the necessary JSON files are provided as command line inputs and are used in initializing the environment.

**Analysis of P4\_mininet.py file:**

This file describes code for P4 hosts and P4 switch.

**P4 Host** is a class in this file which inherits the Host class of mininet python API and is used to provide an outline of the host type that we are going to use. It is called while defining the type of host that we are going to use in the topology while initializing the **Mininet class**. It has commands running to disable the ipv6 using the **cmd()** command.

It also has a method called **describe()** which prints out the interface information along with MAC address and IP address.

**defaultIntf().name 🡺** prints the name of the interface which is renamed to eth0 using the **rename** method of Host class.

**defaultIntf().IP() 🡺** prints the IP address of the host default interface.

**defaultIntf().MAC() 🡺** prints the MAC address of the host interface.

**P4 Switch** is class which gives outline of the P4 switch which we are going to employ for our P4 programs to run. It has **init () method** with following parameters:

1. **Name:** Name of the switch to be identified.
2. **Sw\_path:** Path to the executable binary file of Bmv2 target switch.
3. **Json\_path:** Path to the compiled P4 program in json format.
4. **Thrift\_port:** Port number through which the switch will accept configuration commands.
5. **Pcap\_dump:** Path to the directory where the packet sniffing files will be dumped.
6. **Log\_console:** If this flag is true then it prints the log messages to the console.
7. **Log\_file:** Path to the log directory where all the log message files will be stored.
8. **Device\_id:** device on which the switch will be executed.
9. **Enable\_Debugger:** enables debugger if true.
10. **\*\*Kwargs:** Any extra keyword arguments that are passed apart from the above mentioned are stored in this parameter.

All the above parameters are stored in the class’s own instance variable for further use with all the error checks necessary along with appropriate error messages.

**Check\_switch\_started(self, pid):** process id is as the switch is executed as an independent. While the process is running (pid exists), we check if the thrift server has been started. If the thrift server is ready, we assume that the switch was started successfully. This is only reliable if the thrift server is started at the end of the init process.

**start(self, controllers):** starting a p4 switch with the following args:

1. **-i:** This tag followed by port number followed by ‘@’ concatenated with interface name initializes the interface of the switch.
2. **-pcap:** This tag followed by the path to the pcap dump folder instructs the switch where to dump the packet sniffing files.
3. **–thrift-port:** This tag followed by port number will initialize the thrift port for this switch.
4. **–nanolog:** No idea about this tag.
5. **–device-id:** device id for the switch.
6. **–debugger:** This tag confirms that debugger should be kept on.
7. **–log-console:** This tag affirms that the messages can be displayed onto the console.

After this the P4 switch is started using this command:

elf.cmd(' '.join(args) + ' >' + self.log\_file + ' 2>&1 & echo $! >> ' + f.name)

This will start the P4 switch and will return a pid of the process where this switch is started as a new process.

**stop():** This method terminates the P4 switch by killing the process which is running the switch using the **cmd()** method of mininet python API.

**Analysis of 1swdemo.py:**

It is a single switch topology mininet environment with different mininet API used taking the configuration from the command prompt in the form of various tags:

**1)behavioral-exe(--behavioral-exe):** This tag gives the address to the binary executable of the target bmv2 switch.

**2)thrift port (--thrift-port):** Used for specifying thrift server port for table updates.

**3) number of hosts (--num-hosts):** number of hosts to be connected to switch

**4) mode (--mode):** not sure about what this means

**5) Json file path (--json):** specifies the json path which is to be executed on the switch

**6) pcap dump (--pcap-dump):** dump packets on interfaces to pcap files

Using the above arguments in terminal we can configure our environment.

**P4RuntimeSwitch.py Script Analysis:**

This is a BMv2 switch with gRPC support. The parameters are the same as the P4 switch class explained above. The only addition is grpc\_port parameter which takes in the grpc port number.

The command line arguments are also the same except there is one more argument called:

**--grpc-server-addr 0.0.0.0** concatenated with the grpc port number.

**Analysis of the utils directory programs from Tutorials Repository:**

**1)run\_exercise.py:** This program helps build the bmv2 mininet environment with all the files provided as arguments in the command line. Command line arguments are:

1. **–quite:** helps suppress the log messages.
2. **–t:** Path to the topology file
3. **-l:** mentioning the log directory where all the logging files will be stored.
4. **-p:** mention the pcap directory where all the packet sniffing files will be stored.
5. **-j:** specify the json file of the compiled P4 program that is to be executed on the switch.
6. **-b:** Path to the behavioral executable file

**Get args():** This function extracts all the arguments along with tags and passes it onto the exercise runner class which initializes the complete p4 mininet environment.

**ConfigureP4Switch Method:**

This class configures a P4 switch according to the switch executable type mentioned in the argument. The purpose is to ensure each switch’s thrift port is unique.

It has 2 different subclasses of switches one is of type which employs grpc for communication and the other one employs a simple P4 switch.

**Exercise Topo Class:**

This is the mininet topology class for the P4 tutorial exercises which is configured using the topology.json file provided.

**\_\_init\_\_():** This method takes input the path to executable switch file along with dictionaries of switch and hosts which is parsed from json file and stored in dictionary.

Later in the code it is looped through to find the number of hosts and links are established between the hosts and switches as stored in the dictionary. Mininet python API method **addLink()** is used to establish virtual links in the mininet environment between the hosts and the switches present. Same is used for establishing links between the switches. Also to initialize hosts we use **addHost ()** method and for switches we use **addSwitch ()** method for simple switches or custom made switches written in python can be used our application per se.

There is a helper **parseLink ()** function which is used to identify the linking using string functions to parse the link represented in json array format.

**Class Exercise Runner:**

This class is the essence of how bmv2 and mininet are executed hand in hand. It takes input the topology in json format along with compiled p4 program in the form of json as well. Furthermore, it also take the path of log file and pcap dump file.

The **init()** method parses the json file and initializes appropriate dictionaries for further use by the topology generator method.

**run\_exercise()** method is used to actually run the mininet bmv2 environment creating the network, initializing the hosts and running commands that will populate the tables in hosts and switches which will be used for packet forwarding. **start()** is a sub method which is a mininet API method which actually starts the topology. As mentioned earlier that hosts are to be programmed using the commands that are mentioned in the json file. Same goes with the json file of switches.

**Parse\_Links()** method takes input the description of the form [node1, node2, latency, bandwidth] where latency and bandwidth being optional, parses these descriptions into dictionaries and store them as an array of dictionaries with the above mentioned fields.

**create\_network():** This method creates the mininet network object and store it as self.net. Here mininet topology instance stored as self.topo. Mininet instance stored as self.net

**program\_switch\_p4runtime(self, sw\_name, sw\_dict):**

The parameters are the object of the class, the name of the switch that is to be configured along with all the commands that are being parsed as dictionary from the topology json file. Then the json file is passed along with configuration details such as address of the switch along with port number, device\_id, json config file, working directory, path to the proto\_dump file.

This function is for those switches that has a json file giving all the important configuration details.

**program\_switch\_cli(self, sw\_name, sw\_dict):**

This method will start up the CLI and use the contents of the command files as input.

The switch here will run as a subprocess with a thrift port for command communication.

**program\_switches(self):**

This method will program each switch using the BMv2 CLI and or P4runtime, depending if any command or runtime JSON files were provided for the switches. It will search through the switches dictionary for appropriate fields and decide on the type of switch function to be called.

**program\_hosts(self):**

execute any command provided in the topology.json file on each mininet host.

We know that the json file is type of a collection of dictionaries and in that each host is sub dictionary with key as the host name and command array as the value of the dictionary.

We will loop through this host collection which is stored in a variable called hosts. We will get the host object from the mininet topology and then run through its commands executing it using the mininet python API function **cmd()** which takes the command that is to be executed as input.

**do\_net\_cli(self):**

Starts up the mininet CLI and prints some helpful output. It assumes a mininet instance is stored as self.net and self.net.start() has been called which is a method that will start the mininet topology.

**get\_args():**

This method parses all the command line arguments that are present when this python script is called. The tags are as follows:

1. -q: This tag asserts that the log messages are to suppressed.
2. -t: The file path followed by this tag points to the json file that meticulously elicits the topology that is to be initialized by the mininet.
3. -l: Specifies the path to the log directory where all the log messages are to be stored for further reference.
4. -p: Specifies path to the dump directory where all the sniffing files are to be stored.
5. -j: Specifies path to the compiled p4 program which is also in the form of json.
6. -b: Specifies the binary executable file path of the BMv2 target switch.

First the get\_args() function is called which will extract all the necessary information from the command line. Following it will be the instantiation of Exercise Runner class which will help build the topology and initialize the BMv2 switch with tables and actions that are to be performed when a packet is encountered which will be governed by the P4 program. Following which will be the call to run\_exercise() method which do all the magic of mininet environment and will establish a test environment in mininet.