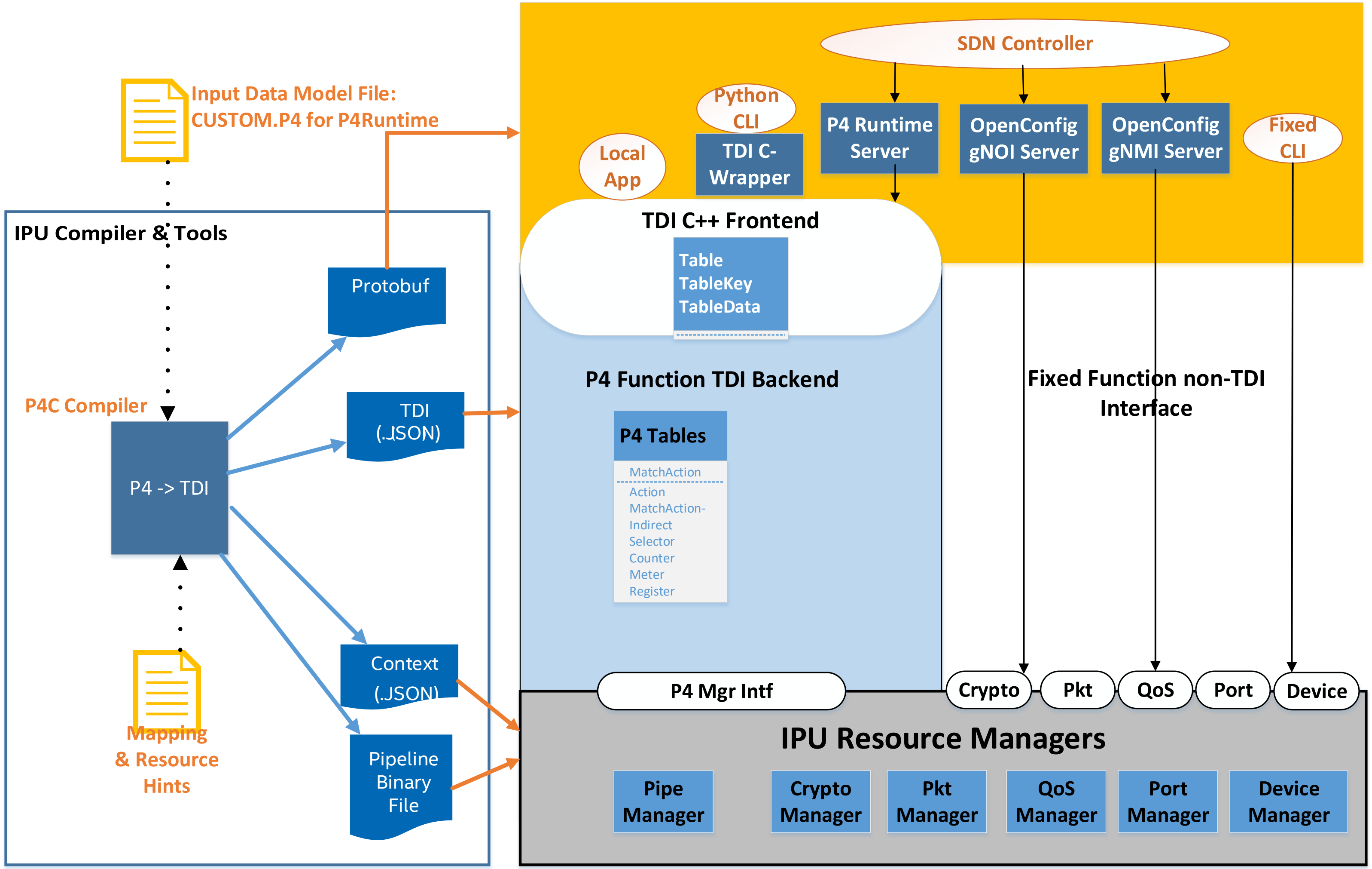
OpenConfig Co-existence with P4 using TDI

James Choi
Sayan Bandyopadhyay
Sabeel Ansari

Current P4 & Fixed Function Control Plane

- P4 function support end-to-end model-based program independent(PI) protocol or API.
- Minimal updates to the protocol or driver API with new feature addition.

- Fixed function (non-P4) based support model based remote protocol, but the local APIs are custom named interfaces.
- Require updates to the named driver API and the interfacing gRPC server with new feature addition.



TDI (Table Driven Interface)

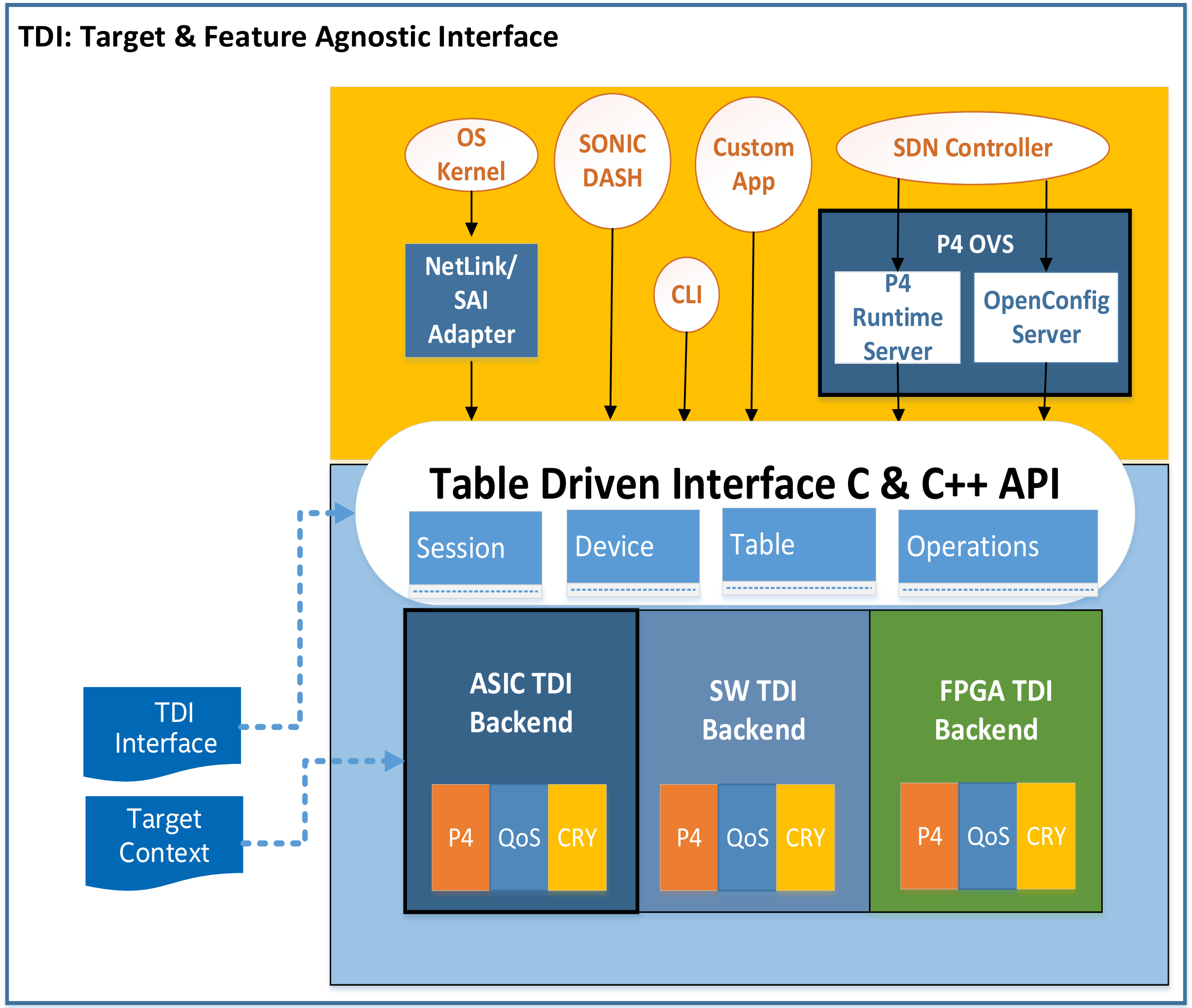
Table Driven Interface (TDI) is a Target Agnostic Interface. Present under p4lang/tdi

- A common frontend for TDI exists in open-source which can be used by control plane applications and target specific backend.
- Every P4 runtime entity is represented as one or multiple tables P4(MatchAction, ActionProfile,…)
- tdi.json is a json-based contract between TDI frontend and control plane on how these tables look like. Similar to p4info in p4runtime
- P4 Compiler generates a tdi.json for P4 entities.

TDI is Feature Agnostic and can support fixed function (non-P4 features) as a set of tables.

1. Not easy to create the TDI.JSON for the fixed functions. These files are hand-written today.
2. Attributes are device specific and not aligned to a standard.

Want to take advantage of the table abstraction of TDI to support fixed functions in similar way as P4 functions.



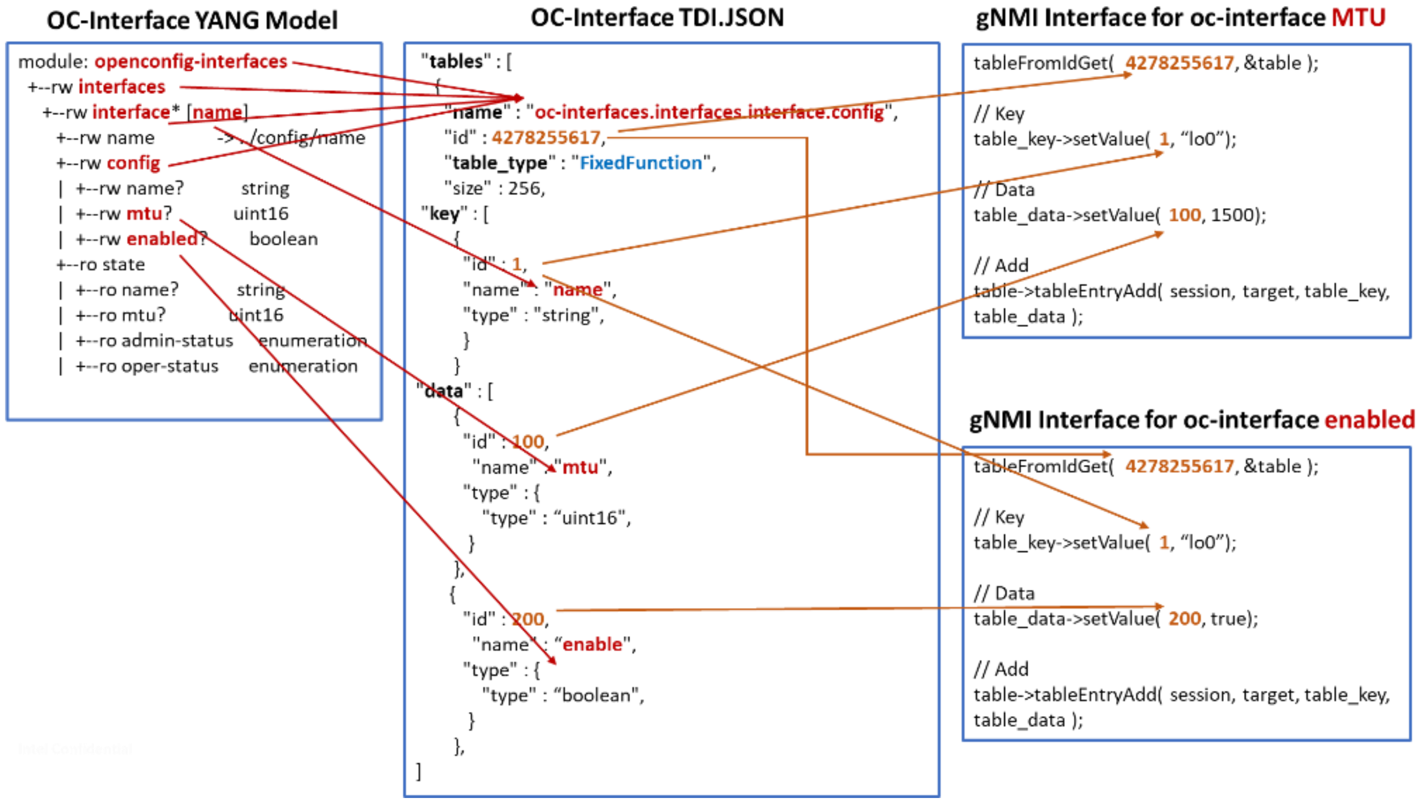
OpenConfig and gNMI

OpenConfig defines various standard data models for network management using YANG modeling language.

- The models follow consistent style, making conducive to develop tool to convert to TDI.JSON.
  - Consistent naming practices -> TDI table and attribute names derived from YANG model hierarchy and attribute names
  - Scalar and List -> Maps to keyless and keyed TDI tables
  - Config and State container split -> Config and State split TDI tables
  - Augmenting models -> Additional tables with inherited key
  - Notifications -> Map to TDI Notifications at module level.

gNMI is a gRPC-based protocol for OC data models from a target device (https://github.com/openconfig/gnmi) and implemented using OC tool-generated code stub.

- With model-based TDI driver interface, the mapping YANG model to TDI.JSON mapping logic can be utilized to automate mapping to TDI in gNMI server (e.g. Stratum)



OC-Interface YANG Model

```
module: openconfig-interfaces
  +--rw interfaces
    +--rw interface* [name]
      +--rw name          -> ../config/name
      +--rw config
      |  +--rw name?        string
      |  +--rw mtu?         uint16
      |  +--rw enabled?     boolean
      +--ro state
      |  +--ro name?        string
      |  +--ro mtu?         uint16
      |  +--ro admin-status   enumeration
      |  +--ro oper-status    enumeration
```

OC-Interface TDI.JSON

```
"tables" : [
  {
   "name" : "oc-interfaces.interfaces.interface.config",
   "id" : 4278255617,
   "table_type" : "FixedFunction",
   "size" : 256,
 "key" : [
    {
     "id" : 1,
     "name" : "name",
     "type" : "string",
     }
    }
 "data" : [
    {
     "id" : 100,
      "name" : "mtu",
     "type" : {
       "type" : “uint16",
      }
    },
   {
     "id" : 200,
      "name" : “enable",
     "type" : {
       "type" : “boolean",
      }
    },
]
```

gNMI Interface for oc-interface MTU

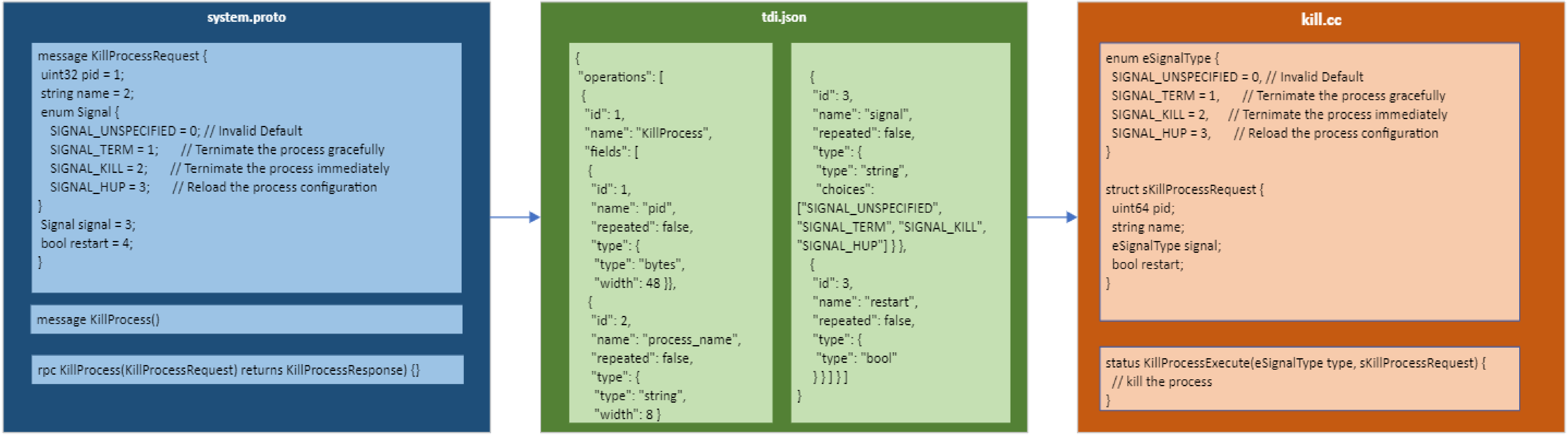
```
tableFromIdGet( 4278255617, &table );

// Key
table_key->setValue( 1, “lo0”);

// Data
table_data->setValue( 100, 1500);

// Add
table->tableEntryAdd( session, target, table_key,
table_data );
```

gNMI Interface for oc-interface enabled

```
tableFromIdGet( 4278255617, &table );

// Key
table_key->setValue( 1, “lo0”);

// Data
table_data->setValue( 200, true);

// Add
table->tableEntryAdd( session, target, table_key,
table_data );
```

OpenConfig and gNOI

- OpenConfig defines various standard network operations as microservices to be executed on a target device using protobuf definitions. (https://github.com/openconfig/gnoi)
- gNOI Operations are shallow RPCs with narrow scoped set of services and messages.
- gNOI operation representations are diverse and requires a more diverse set of vocabularies to describe.
- To support more operational vocabularies, TDIOperation and TDINotification have been added to TDI specification.
- Following outlines one possible model to service a KillProcess operation.



```
message KillProcessRequest {
 uint32 pid = 1;
 string name = 2;
 enum Signal {
   SIGNAL_UNSPECIFIED = 0; // Invalid Default
   SIGNAL_TERM = 1;      // Ternimate the process gracefully
   SIGNAL_KILL = 2;     // Ternimate the process immediately
   SIGNAL_HUP = 3;      // Reload the process configuration
 }
 Signal signal = 3;
 bool restart = 4;
}
```

```
message KillProcess()
```

```
rpc KillProcess(KillProcessRequest) returns KillProcessResponse) {}
```

```
{
 "operations": [
  {
   "id": 1,
   "name": "KillProcess",
   "fields": [
    {
     "id": 1,
     "name": "pid",
     "repeated": false,
     "type": {
      "type": "bytes",
      "width": 48 }},
    {
     "id": 2,
     "name": "process_name",
     "repeated": false,
     "type": {
      "type": "string",
      "width": 8 }
```

```
    {
     "id": 3,
     "name": "signal",
     "repeated": false,
     "type": {
      "type": "string",
      "choices":
["SIGNAL_UNSPECIFIED",
"SIGNAL_TERM", "SIGNAL_KILL",
"SIGNAL_HUP"] } },
    {
     "id": 3,
     "name": "restart",
     "repeated": false,
     "type": {
      "type": "bool"
     } } ] } ]
}
```

```
enum eSignalType {
  SIGNAL_UNSPECIFIED = 0, // Invalid Default
  SIGNAL_TERM = 1,      // Ternimate the process gracefully
  SIGNAL_KILL = 2,     // Ternimate the process immediately
  SIGNAL_HUP = 3,      // Reload the process configuration
}

struct sKillProcessRequest {
  uint64 pid;
  string name;
  eSignalType signal;
  bool restart;
}
```

```
status KillProcessExecute(eSignalType type, sKillProcessRequest) {
  // kill the process
}
```

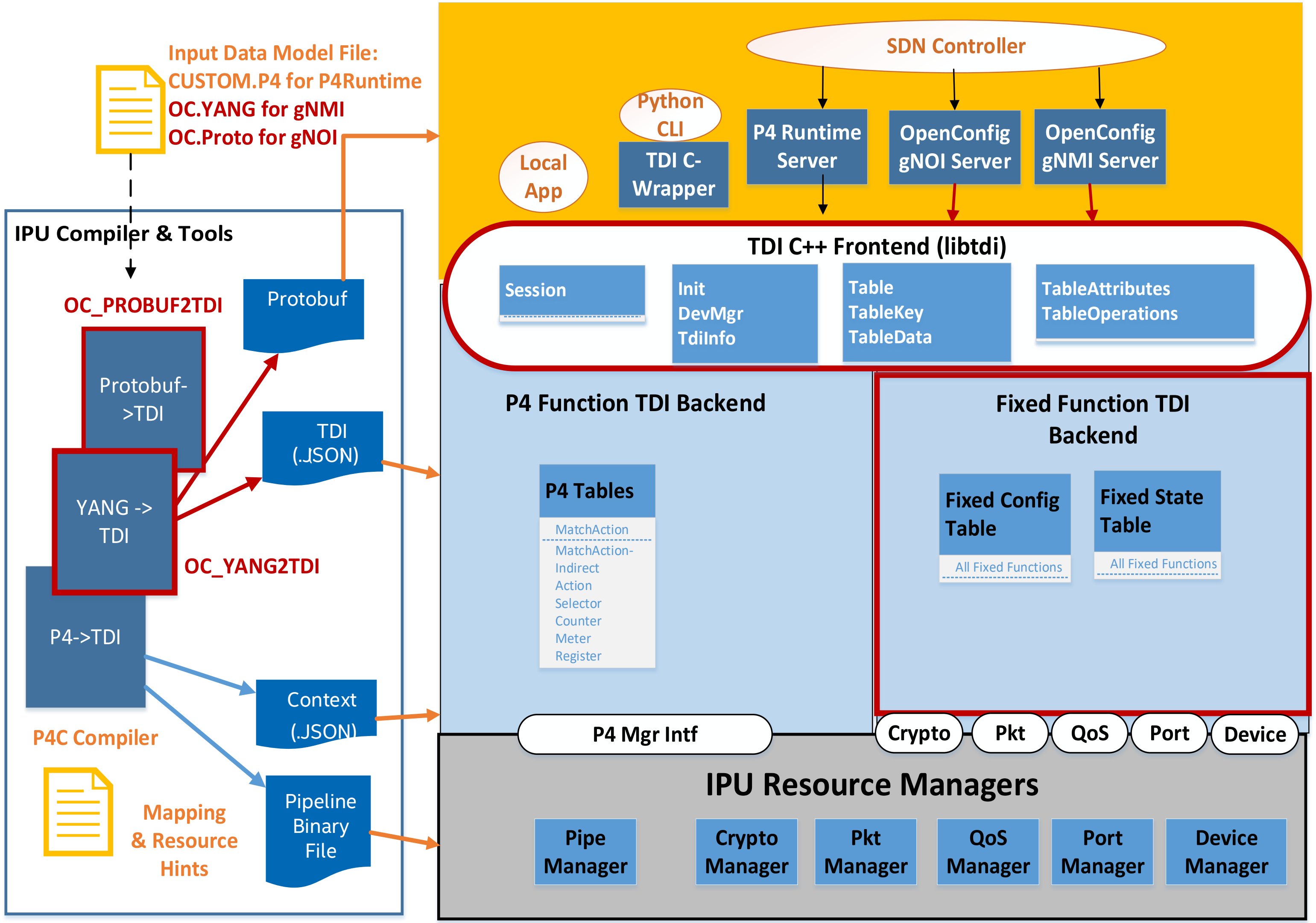
Uniform Control Plane for P4 & Fixed Functions Using TDI

- For both P4 and Fixed functions, generate the TDI.JSON from standard model files.
- Utilize tools to generate data model artifacts and code to help automate or make easier feature additions.

- To support OC gNMI in similar way as P4, OC_Yang2TDI (new tool) automates generating Fixed Function artifacts (TDI.json) for non-P4 tables

- To support gNOI using TDI with more operational requests, TDIOperation and TDINotification are added to TDI.

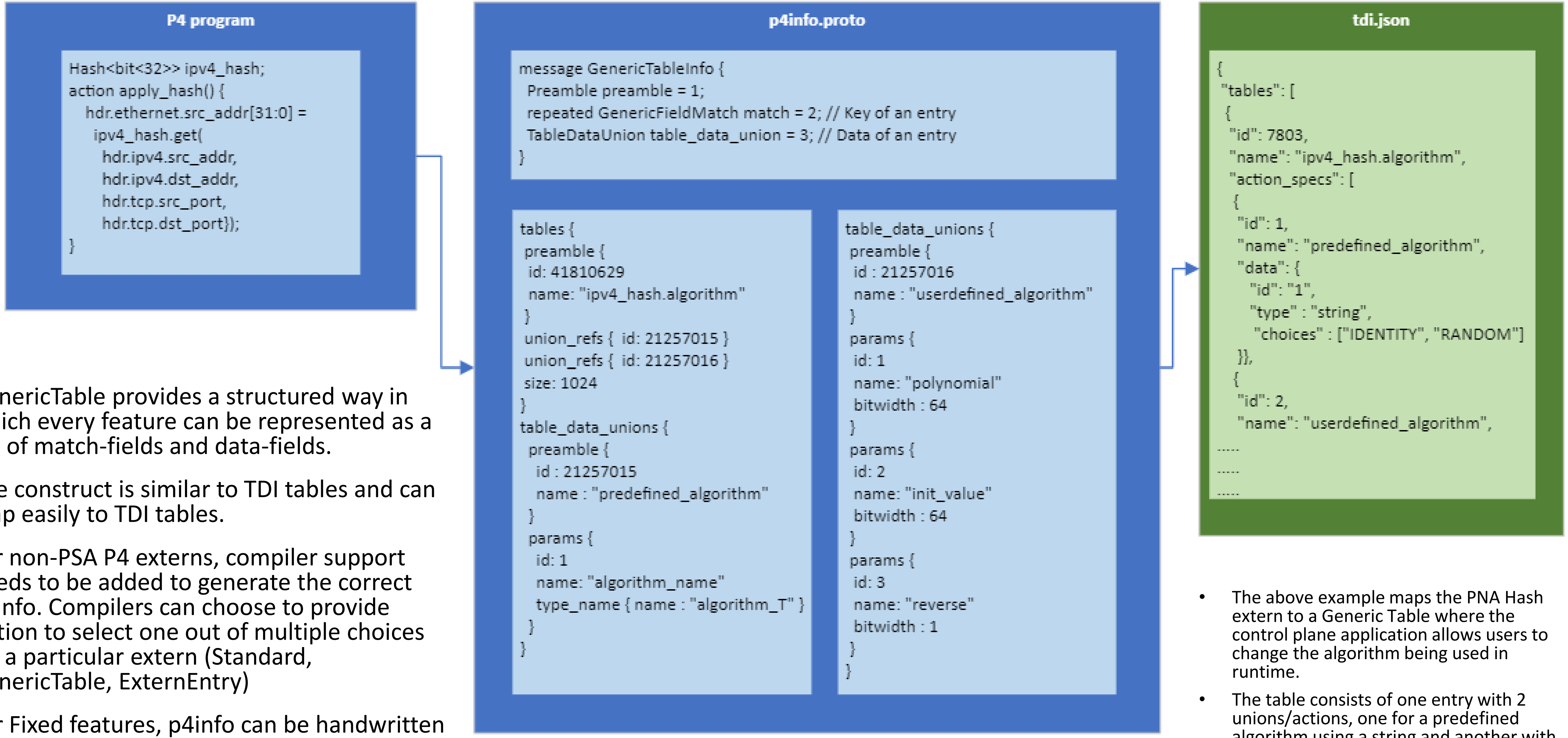
- Using both P4Runtime and OpenConfig gNMI & gNOI is most common deployment mechanism; however, we propose a new method that may help to simplify this further.



P4Runtime GenericTables

- Present p4runtime interface
  - P4 tables and Externs like MatchAction, Counters are supported on p4runtime
  - P4runtime native API contains support for some forwarding functionalities which can be fixed functions in targets
    - Multicast,
    - Cloning, Mirroring
- Problem :
  - The fixed p4runtime support isn't extensible. Any enhanced functionality like ECMP group management in Multicast is not natively possible.
  - No support possible if vendor wants to provide Traffic management and shaping over p4runtime.
  - No native support for non-PSA externs like PNA or vendor specific P4 architectures. ExternEntry is present but is a very open ended mechanism without any specifications, left for vendors to decide runtime and info protos.
- Proposal
  - GenericTableEntry in p4runtime (in progress)
  - GenericTable to be used for non-PSA externs.
  - Can also be used for fixed features where the vendor/user identifies a use to keep them on p4runtime. For example, users might want to configure all forwarding related features over a single interface.

P4Runtime GenericTables Continued



P4 program

```
Hash<bit<32>> ipv4_hash;
action apply_hash() {
   hdr.ethernet.src_addr[31:0] =
     ipv4_hash.get(
       hdr.ipv4.src_addr,
       hdr.ipv4.dst_addr,
       hdr.tcp.src_port,
       hdr.tcp.dst_port});
}
```

p4info.proto

```
message GenericTableInfo {
  Preamble preamble = 1;
  repeated GenericFieldMatch match = 2; // Key of an entry
  TableDataUnion table_data_union = 3; // Data of an entry
}
```

```
tables {
 preamble {
  id: 41810629
  name: "ipv4_hash.algorithm"
 }
 union_refs {  id: 21257015 }
 union_refs {  id: 21257016 }
 size: 1024
}
table_data_unions {
  preamble {
    id : 21257015
    name : "predefined_algorithm"
  }
  params {
    id: 1
    name: "algorithm_name"
    type_name { name : "algorithm_T" }
  }
}
```

```
table_data_unions {
 preamble {
  id : 21257016
  name : "userdefined_algorithm"
 }
 params {
  id: 1
  name: "polynomial"
  bitwidth : 64
 }
 params {
  id: 2
  name: "init_value"
  bitwidth : 64
 }
 params {
  id: 3
  name: "reverse"
  bitwidth : 1
 }
}
```

tdi.json

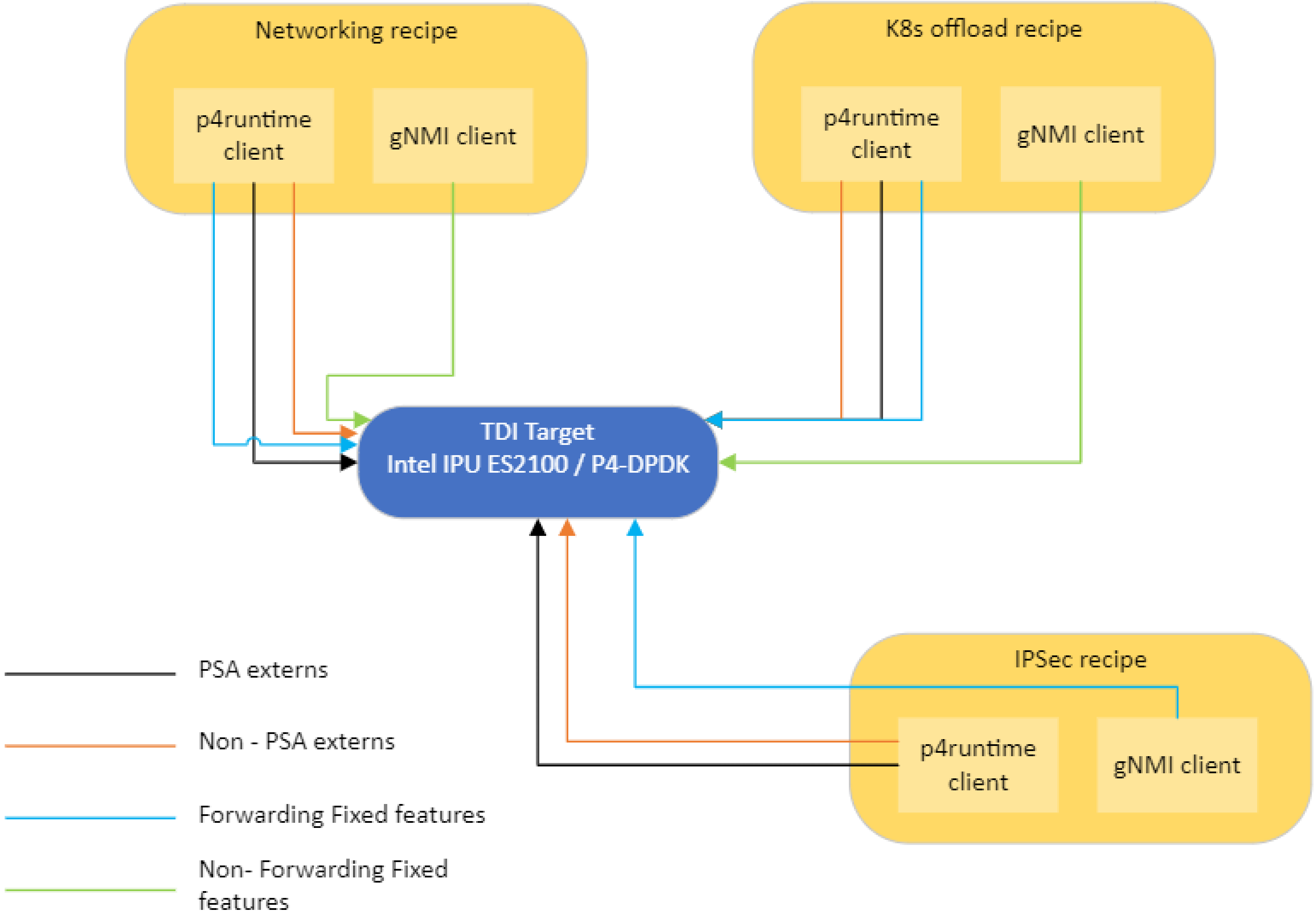
```
{
 "tables": [
  {
   "id": 7803,
   "name": "ipv4_hash.algorithm",
   "action_specs": [
    {
     "id": 1,
     "name": "predefined_algorithm",
     "data": {
       "id": "1",
       "type" : "string",
        "choices" : ["IDENTITY", "RANDOM"]
     }},
    {
     "id": 2,
     "name": "userdefined_algorithm",
.....
.....
.....
```

- GenericTable provides a structured way in which every feature can be represented as a set of match-fields and data-fields.
- The construct is similar to TDI tables and can map easily to TDI tables.
- For non-PSA P4 externs, compiler support needs to be added to generate the correct p4info. Compilers can choose to provide option to select one out of multiple choices for a particular extern (Standard, GenericTable, ExternEntry)
- For Fixed features, p4info can be handwritten to map easily to the corresponding tdi.json

- The above example maps the PNA Hash extern to a Generic Table where the control plane application allows users to change the algorithm being used in runtime.
- The table consists of one entry with 2 unions/actions, one for a predefined algorithm using a string and another with parameters used to define a CRC algorithm

P4Runtime GenericTable Usecase

- Users might prefer using p4runtime to program forwarding related fixed features using p4runtime since it is already used to program forwarding tables
  - Networking and K8s recipes in IPDK (Infrastructure Programmer Development Kit) use p4runtime to program some forwarding related fixed features like Multicast and Mirroring.
  - IPSec uses gNMI to program some forwarding related fixed features like SADB(Security Association Database) which contains the crypto-tag/SPI (Security Parameter Index)/crypto-algorithm information.
- Non-PSA P4 externs cannot be mapped easily from p4runtime. GenericTable proposes to fill this gap
- Enhanced Forwarding fixed features like Multicast with ECMP groups can also be easily supported over p4runtime with GenericTables



What’s next?

- Enhancements to OC_YANG2TDI tool with config file to allow different variations in table generation.
- Development of a tool to generate TDI.JSON from gNOI protobuf definitions.
- Taking GenericTables in P4 API WG forward.
  - PR Link : https://github.com/p4lang/p4runtime/pull/419
- A tool to convert an existing tdi.json to its p4info representation
  - Tdi.json -> p4info for vendors who have a fixed tdi.json for fixed features
  - P4info.txt -> tdi.json for device developers who want to generate a tdi.json from a new handwritten p4info.txt of a GenericTable. P4 compiler does this from P4 programs, but there is no present way in which developers can do this translation for purely handwritten p4info.txt

Thank You!

James Choi
Sayan Bandyopadhyay
Sabeel Ansari