

ToDiff Developer's Guide

ToDiff

Overview

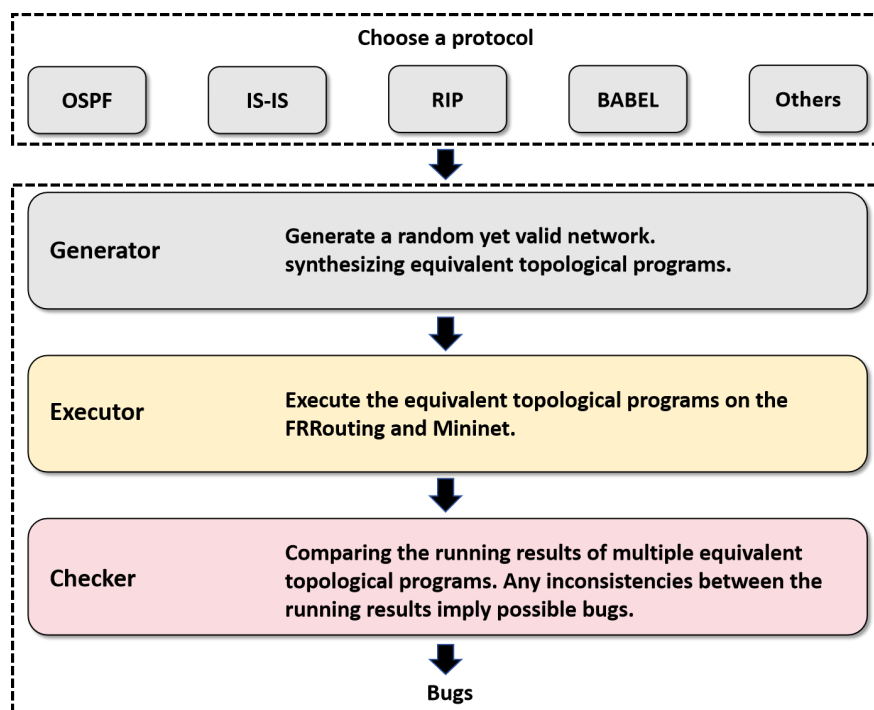
ToDiff is a prototype implementation of the technique known as ToDiff. It aims to validate Interior Gateway Protocols(a key class of routing protocols) via equivalent topology synthesis.

ToDiff performs differential testing in three steps:

Step 1: Generate valid yet random topologies along with their corresponding equivalent topological programs.

Step 2: Simulate the network, inject the topological programs into routing protocol implementations, and collect the execution output.

Step 3: Compare the outputs and analyze the root causes of any discrepancies.



These steps are handled by three separate components: `generator` , `executor` , and `checker` , as shown in the figure above.

Next, we will introduce these three components using the OSPF protocol as an example.

Extend protocols to validate

One should modify all three components step by step:

- Modify [generator/diffTopo](#).
- Modify [generator/lib/topo](#).
- Modify [generator/lib/generator](#).
- Modify [generator/lib/frontend](#).
- Modify [generator/lib/reducer](#).
- Modify [executor](#).
- Modify [checker](#).

Generator

The **generator** provides a complete framework for generating, simplifying, and managing equivalent topological programs for testing routing protocol implementations.

It is composed of one application and four library modules:

- **Application:** `diffTopo`
- **Libraries:** `lib/topo` , `lib/generator` , `lib/reducer` , `lib/frontend`

The application is the entry point of the generator, which serves as a driver to use functions in libraries to generate equivalent topological programs.

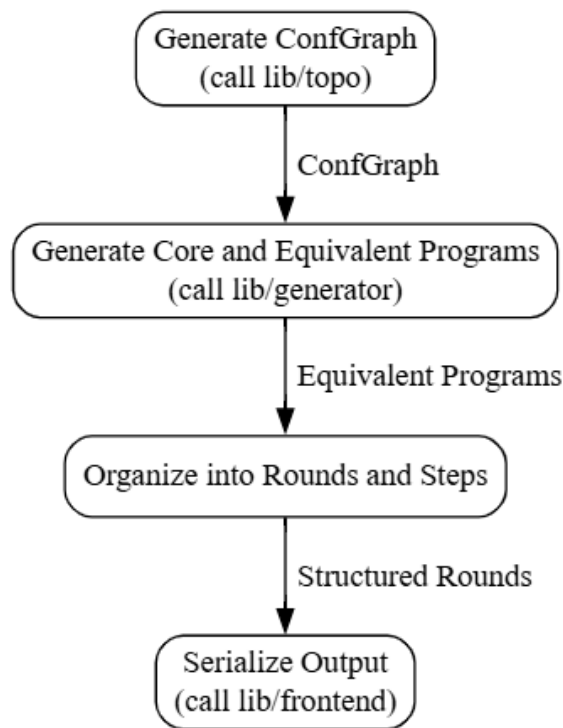
The output of the generator is the **testcase** consisting of multiple equivalent topological programs for testing, and will be simulated in executor.

Application: diffTopo

DiffTopo is the main application responsible for generating and organizing equivalent topological programs for routing protocol validation.

It coordinates the generation of topologies, the construction of equivalent programs, and the structured output of these programs into executable testcases.

Execution flow of diffTopo



1. Generate ConfGraph

- Call `lib/topo` to create a randomized or customized network topology (`ConfGraph`).

2. Generate Core and Equivalent Programs

- Call `lib/generator` to produce both physical and protocol equivalent programs based on the `ConfGraph`.

3. Organize Programs into Rounds and Steps

- Group each topological program into one round.
- Within each round, organize commands into steps with wait times for protocol convergence or transition.

4. Serialize Output

- Call `lib/frontend` to convert commands to characters.
- Store the generated testcase into a structured JSON file.

Testcase format

Each testcase is saved in a JSON structure with the following top-level format:

代码块

```
1  {
2    "geninfo": { ... },
3    "conf_name": "test1223444.json",
4    "step_nums": [1, 3],
```

```

5     "round_num": 2,
6     "routers": ["r0", "r1"],
7     "commands": [round0, round1]
8 }

```

- **geninfo:**
Auxiliary information recorded during testcase generation. See below for details.
- **conf_name:**
Name of the configuration testcase file.
- **step_nums:**
List of integers. Each element indicates the number of steps in a corresponding round.
- **round_num:**
Total number of rounds, equal to the number of generated equivalent programs.
- **routers:**
List of router names involved in the configuration.
- **commands:**
List of rounds, each containing a sequence of steps.

geninfo Structure

代码块

```

1  {
2      "core_commands": {
3          "router_name": "commands_str"
4      },
5      "configGraph": "DOT format string",
6      "configGraphAttr": "DOT format string",
7      "evaluate": {
8          "genGraphTime": 0.123,
9          "genEqualTime": 0.456,
10         "totalTime": 0.789,
11         "totalInstruction": 120
12     }
13 }

```

- **core_commands:**
Mapping from router names to their core topological program.

- **configGraph:**
Physical network topology, represented in DOT graph format.
- **configGraphAttr:**
ConfGraph(including protocol attributes), represented in DOT format.
- **evaluate:**
Performance statistics during generation:
 - `genGraphTime` : Time for topology graph generation.
 - `genEqualTime` : Time for equivalent program generation.
 - `totalTime` : Total generation time.
 - `totalInstruction` : Number of instructions generated across all programs.

Round and Step Structure

Each round corresponds to one equivalent program. Each round is divided into multiple steps, allowing fine-grained control of the configuration sequence.

Example of a single step:

代码块

```
1  {
2      "step": 0,
3      "waitTime": 2,
4      "phy": ["physical layer commands"],
5      "ospf": ["ospf layer commands"]
6  }
```

- **step:**
The index of the step within the round.
- **waitTime:**
Time to wait after executing this step:
 - `>0` means a fixed wait of `>0` seconds.
 - `-1` means wait until protocol convergence is detected.
- **phy:**
List of physical topological commands applied in this step. (Detailed in [physical topological commands section](#))

- **ospf(<proto>):**

List of OSPF protocol topological commands applied in this step. (Detailed in [protocol topological commands section](#))

How to add a new protocol to the diffTopo?

1. `diffTopo.java`

Modify `diffTopo.java` to generate equivalent topological programs for the new protocol.

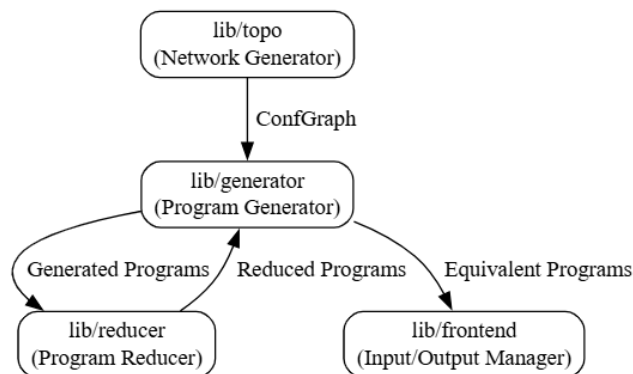
Lib: four components

The `lib` directory contains the core modules responsible for generating, simplifying, and managing equivalent topological programs for routing protocol validation.

It is structured into four major modules:

- `lib/topo`
- `lib/reducer`
- `lib/generator`
- `lib/frontend`

Their interaction is illustrated below:



- **lib/topo (Network Generator)**

- Generates randomized or customized network topologies, represented by `ConfGraph` structure.
- `ConfGraph` consists of both physical connectivity and protocol-specific configurations.
- `ConfGraph` is the initial input for program generation.

- **lib/generator (Program Generator)**

- Transforms a given `ConfGraph` into **core** and **equivalent topological programs**.
- Iteratively interacts with `lib/reducer` to generate equivalent programs

- **lib/reducer (Program Reducer)**
 - Simplifies generated programs by removing redundant or overridden commands.
 - Called by `lib/generator` to guide equivalent program synthesis.
 - Enhances efficiency and reduces test case size without affecting correctness.
- **lib/frontend (Input/Output Manager)**
 - Handles the storage and serialization of generated equivalent programs.

Next, we will give a brief introduction of all four components.

lib/topo

The topology generator (`lib/generator/topo`) is responsible for producing randomized network topologies tailored for testing routing protocols such as OSPF. The generator not only constructs physical network connectivity but also assigns necessary protocol-specific attributes (e.g., OSPF areas, Hello intervals).

Structure of topo module

代码块

```

1  lib/generator/topo/
2  |— driver/
3  |   |— topo.java           # Entry point for topology generation
4  |— item/
5  |   |— base/
6  |       |— Intf.java       # Basic Interface class
7  |       |— Router.java    # Basic Router class
8  |— pass/
9  |   |— attri/
10 |       |— ospfRanAttriGen.java # OSPF-specific attribute generator
11 |       |— <proto>RanAttriGen.java # Other protocol attribute generator
12 |   |— base/
13 |       |— ospfRanBaseGen.java # OSPF-specific base topology generator
14 |       |— <proto>RanBaseGen.java # other protocol base topology generator
15 |   |— build/
16 |       |— topoBuild.java    # Topology assembly and ConfGraph
17 |
   construction

```

- **driver/topo.java**

Top-level entry that parses input arguments, initializes generation context, and triggers topology generation passes.

- **item/base/**

Base definitions of network elements used in `pass/base` :

- `Router.java` , `Intf.java` are basic node and interface classes

- **pass/base/**

Generates the *topology skeleton* (nodes, links) without detailed attributes.

- For example, `ospfRanBaseGen.java` randomly generates an OSPF-compatible network structure.

- **pass/attri/**

Fills in *protocol-specific attributes* (e.g., OSPF area ID, hello intervals) for the generated topology.

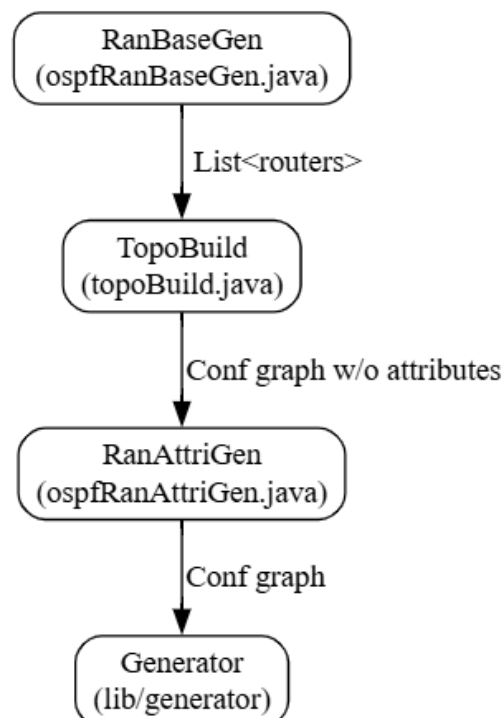
- `ospfRanAttriGen.java` is used for OSPF.

- **pass/build/**

Builds the complete topology object in form of Conf Graph.

- `topoBuild.java` converts base topology(represented in network elements defined in `item/base`) into a ConfGraph.

Execution flow of topo module



The `genGraph()` function in `topo.java` orchestrates the entire topology generation process.

It consists of three main stages, each with distinct responsibilities and data structures, as shown in the figure above.

1. **Generate base physical topology and protocol logical areas**

Call `base/ospfRanBaseGen.java` to generate physical topology and area division in `List<Routers>`.

2. **Build detailed topology graph structures**

Call `build/topoBuild.java` to build `ConfGraph` according to `List<Routers>` generated in stage 1.

3. **Assign protocol attributes**

Call `attri/ospfRanAttriGen.java` to assign attributes of the protocols to the `ConfGraph` which are complied with the protocol specification.

Key data structure: ConfGraph

The `lib/item/conf` module defines the **ConfGraph**, which serves as the protocol network model within the ToDiff framework.

`ConfGraph` is a directed graph whose nodes represent both:

- Physical device (e.g., Routers, Switches, Hosts)
- Protocol configuration(e.g., OSPF processes, OSPF interfaces, OSPF area summaries)

The edges in the graph represent relationships between nodes, such as:

- Physical connections (links between routers or hosts)
- Logical ownership or binding (e.g., a router owning an OSPF process)

Through this abstraction, `ConfGraph` can simultaneously capture both the physical topology and the protocol-specific configurations in the OSPF network.

The `lib/item/conf` module is organized into the following subdirectories:

Subdirectory	Purpose
edge/	Defines the relationship between two nodes via <code>RelationEdge</code>
graph/	Provides the <code>ConfGraph</code> implementation and related graph utilities
node/	Defines all node types used in <code>ConfGraph</code> , including both physical nodes and protocol-specific nodes

The `node/` subdirectory defines all the **node types** that appear in the ConfGraph, which are divided into two major categories:

- Physical Nodes (`node/phy/`)

Class	Description
Router	Represents a physical router device.
Host	Represents a host device (e.g., client machine).
Switch	Represents a switch device (may be abstracted).
Intf	Represents a physical network interface, attached to a Router or Host.

Main characteristics:

- Physical nodes maintain a list of interfaces (`Intf`).
 - Routers have basic properties such as `routerId` and a reference to area IDs (for initial generation purposes).
 - Interfaces (`Intf`) may hold IP addresses and state (up/down).
- Protocol-Specific Nodes (`node/ospf/` , etc.)

Each routing protocol (such as OSPF) has its own specialized node types, located under corresponding protocol folders.

Taking `node/ospf/` as an example:

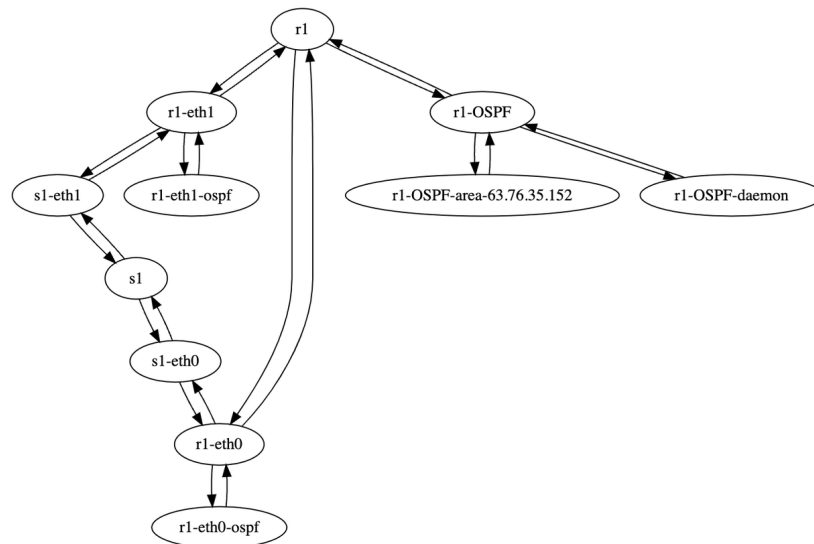
Class	Description
OSPF	Represents an OSPF process associated with a router.
OSPFAreaSum	Represents an area aggregation node for OSPF (summarizing routers within the same area).
OSPFIntf	Represents an OSPF-level interface configuration, attached to a physical interface.
OSPFDaemon	Stores daemon-level settings for the OSPF process

Main characteristics:

- Protocol nodes encapsulate **protocol-specific attributes** as their fields.

- For example:
 - `OSPFIntf` stores Hello interval, Dead interval, Priority, Cost for OSPF.
 - `OSPFAreaSum` stores Area ID and area-wide settings.
- These nodes are **connected via edges** from their corresponding physical nodes.

Example



In the ConfGraph shown above, we have one router `r1` and one switch `s1`. Router `r1` has two interface `r1-eth0` and `r1-eth1`; switch `s1` has two interface `s1-eth0` and `s1-eth1`. Interface `r1-eth0` is connected to `s1-eth0` and `r1-eth1` is connected to `s1-eth1`, the edge Type is physically connected. Router `r1` has one OSPF daemon, thus it connects to node `r1-OSPF` with edge type OSPF BELONGS. Each interface has one OSPF interface node, namely `r1-eth1-ospf` and `r1-eth0-ospf`.

How to add a new protocol to the topo module?

1. Add Base Generation

- Create `<proto>RanBaseGen.java` under `pass/base/`
- Implement random generation logic respecting protocol requirements.

2. Add Attribute Generation

- Create `<proto>RanAttriGen.java` under `pass/attri/`
- Assign necessary attributes.

3. (Optional) Add Specialized Node Classes

- Extend `Router.java` or `Intf.java` if protocol needs extra metadata.

4. Integrate into Driver

- Modify `topo.java` to dispatch to a new base and attribute passes if `<proto>` is selected.

lib/generator

The `lib/generator` module is responsible for generating equivalent topological programs based on the provided `ConfGraph`.

It first translates a given `ConfGraph` into a minimal topological program, which we refer to as `core program`, then employs an equivalent topology synthesis algorithm to synthesize equivalent topological programs for differential testing.

Structure of generator module

The `lib/generator` module is organized into several subdirectories, each responsible for different phases of program generation.

The overall structure is as follows:

代码块

```
1  lib/generator/
2  |— driver/
3  |   └─ generate.java      # Main driver to orchestrate the generation
                             process
4  |— phy/
5  |   └─ pass/
6  |      └─ genPhyCorePass.java # Generate core physical operations from
                             ConfGraph
7  |      └─ genPhyEqualPass.java # Generate equivalent physical operations
8  |— ospf/
9  |   └─ pass/
10 |      └─ genCorePassOspf.java # Generate core OSPF protocol operations
                             from ConfGraph
11 |      └─ genEqualPass.java    # Generate equivalent OSPF operations
12 |      └─ shrinkCorePass.java  # Shrink and optimize core operation
                             sequences
13
```

Execution flow of generator module

The `lib/generator` module transforms a given `ConfGraph` into executable equivalent topological programs through a structured sequence of passes.

The goal is to generate configuration programs that correctly set all attributes specified in the ConfGraph, ensuring that the resulting network configuration matches the ConfGraph exactly.

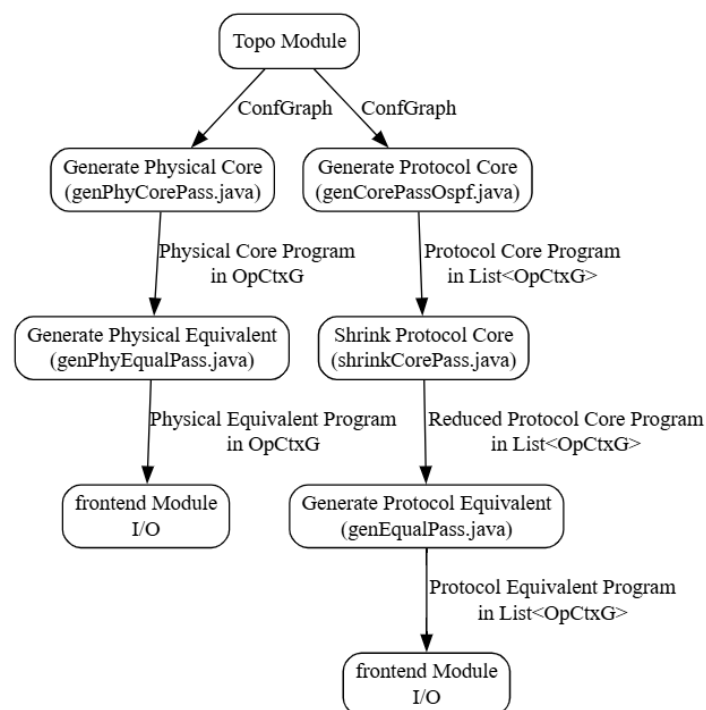
The entire generation process is divided into two independent layers:

- **Physical Layer:**
Responsible for generating commands related to physical topology, which we refer to as physical topological commands, including device creation, link establishment, and interface activation.
- **Protocol Layer:**
Responsible for generating commands input to routing protocols, which we refer to as protocol topological commands.

Each layer follows a similar pipeline of **Core Program Generation → Equivalent Program Generation**:

- **ConfGraph → Core Programs:**
A **Core Program** is the minimal set of topological commands necessary to fully realize the intended network state.

From a single ConfGraph, **multiple Core Programs** can be derived.
- **Core Programs → Equivalent Programs:**
Each Core Program is expanded into **multiple Equivalent Programs** by injecting additional commands, ensuring that all attributes in ConfGraph are eventually correctly set.



a) Generating physical layer program

- **Generate Physical Core (`genPhyCorePass.java`)**

- **Input:** ConfGraph
- **Output:** Physical core program (`OpCtxG`)
- **Description:**
Generate physical topological commands(physical topological program) like adding routers, switches, links, and activating interfaces to establish the topology described in ConfGraph. (Detailed in [physical topological commands section](#))
- **Details:**
For each router, switch, or host node in the `ConfGraph` , generate a `NODEADD` operation.
For each pair of connected interfaces, generate a `LINKADD` operation.
For each active interface, generate an `INTFUP` operation to bring the interface up.
For each router which runs the protocol, generate a `PROTOCOLUP` operation.
- **Generate Physical Equivalent (`genPhyEqualPass.java`)**
 - **Input:** Core physical program (`OpCtxG`)
 - **Output:** Physical equivalent program (`OpCtxG`)
 - **Description:**
Expand the physical core by inserting additional operations (e.g., interface flaps, link removals and restorations) without changing the final network structure.
 - **Details:**
Insert additional operations such as:
 - Temporary shutdown and reactivation of interfaces
 - Link removal and re-establishment
 - Node deletion and rebooting
 Carefully maintain operation dependencies (e.g., a link can only be removed if its interfaces exist).

b) Generating protocol layer program

- **Generate Protocol Core (`genCorePass0spf.java`)**
 - **Input:** ConfGraph
 - **Output:** Protocol core programs (`List<OpCtxG>`)
 - **Description:**
Generate minimal OSPF topological program according to the protocol manual.
 - Developers manually define how to translate ConfGraph attributes into commands.

- Multiple Cores can exist for the same ConfGraph because different syntaxes can realize the same configuration semantics.

Example:

- **Core 1:** `interface r1-eth0 ip address 1.1.1.1; ip ospf area 1`

- **Core 2:** `interface r1-eth0 ip address 1.1.1.1; router ospf network 1.1.1.1 area 1`

These represent two different core programs achieving the same OSPF area assignment, adding interface `r1-eth0` to area 1.

- **Details:**

For each router requiring OSPF:

- Create a `R OSPF` operation to start the OSPF process.
- Generate a `RID` operation to set the router ID.

For each interface configured for OSPF:

- Create operations to set hello interval, dead interval, priority, and cost.
- Generate network advertisements (`NETWORK` operations) or area assignments (`IP_OSPF_AREA`).

- **Shrink Protocol Core (`shrinkCorePass.java`)**

- **Input:** Core protocol programs (`List<OpCtxG>`)
- **Output:** Reduced protocol core programs (`List<OpCtxG>`)
- **Description:**

Some generated commands may explicitly set attributes to their default values (e.g., hello interval = 10 seconds). These operations are unnecessary and can be safely removed to simplify the program.

Shrinking ensures that the core remains minimal and only retains essential operations.

- **Generate Protocol Equivalent (`genEqualPass.java`)**

- **Input:** Reduced protocol core programs (`List<OpCtxG>`)
- **Output:** Protocol equivalent programs (`List<OpCtxG>`)
- **Description:**

Expand each shrunk core into multiple equivalent programs by inserting topological commands (temporary value modifications, removals, reactivations) while guaranteeing that the final protocol attributes remains equivalent to the original ConfGraph.

- **Details:**

Use an equivalent topology synthesis algorithm (See Pseudo Code in paper Section 4.2) to synthesize equivalent programs with bounded length.

`actionRulePass` , `applyRulePass` , `movePass` and `NormalController` are helper functions to do the synthesis process.

Key data structures

- **OpCtxG (lib/item/IR/OpCtxG.java)**

- **Description:**

A group of operations (`OpCtx`) representing a topological program.

- **Key Points:**

- Main container for Core and Equivalent Programs.
 - Supports merging, cloning, and output generation.

- **OpAnalysis (lib/item/opg/OpAnalysis.java)**

- **Description:**

Associates an operation with a state (INIT, ACTIVE, REMOVED) during equivalent generation.

- **Key Points:**

- Tracks activation/removal transitions.
 - Enables controlled mutation and restoration of operations.

- **OpOspf (lib/item/IR/OpOspf.java)**

- **Description:**

Represents protocol topological commands.

- **Key Points:**

- Covers process startup, router ID, network announcements, and interface parameters.
 - Used in protocol core and equivalent generation.

- **OpPhy (lib/item/IR/OpPhy.java)**

- **Description:**

Represents physical topological commands.

- **Key Points:**

- Includes node creation, link setup, and interface activation.
 - Used in physical core and equivalent generation.

How to add a new protocol to generator?

1. Implement protocol topological program generation

- Create a new core generation pass, e.g., `genCorePass<Protocol>.java`, under `ospf/pass/`.
- Parse `ConfGraph` and generate minimal operations (`OpCtxG`) to realize the protocol configuration.
- Developers need to manually map `ConfGraph` attributes to protocol topological commands based on the protocol's specification.

2. Implement physical topological program generation

- Add `ProtocolUP` and `ProtocolDown` commands in `genPhyCore` pass
- Modify `genPhyEqual` Pass to support these two newly added commands.

3. Integrate into driver

Update `generate.java` to dispatch to the new protocol' s core generation and equivalent generation passes based on the selected protocol.

lib/frontend

The `lib/frontend` module is to construct, for protocol's configuration instructions, its instruction types and the corresponding lexical templates used to generate equivalent instructions. It handles the storage and serialization of generated equivalent programs.

Structure of frontend module

The `lib/frontend` module consists of three parts, each with a different function.

The overall structure is as follows:

代码块

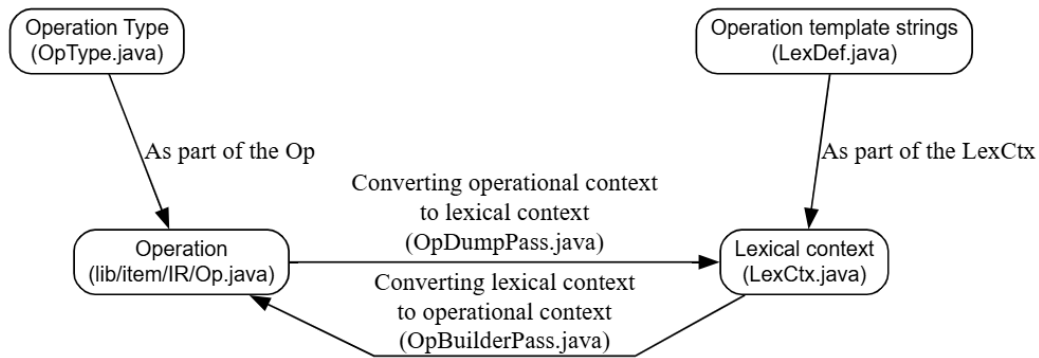
```
1  lib/frontend/
2  |— driver/
3  |   |— IO.java           # Read and write operation
4  |— lexical/
5  |   |— LexDef.java      # Parse template strings, extracting parameter and
   |   range information
6  |   |— OpType.java      # Define the types of OSPF operations
7  |— pass/
8  |   |— LexToStrPass.java # Convert lexical contexts to strings
```

```

9 | | — OpDumpPass.java      # Convert an operational context to a lexical
  | | context
10 | | — OpBuilderPass.java  # Convert lexical contexts to operational contexts
11 | | — StrToLexPass.java   # Convert strings contexts to lexical

```

Execution flow of frontend module



The `lib/frontend` module is to construct, for protocol's configuration instructions, its instruction types and the corresponding lexical templates used to generate equivalent instructions.

All instructions that can be generated need to have their instruction type defined in `OpType.java`, and their corresponding lexical template defined in `LexDef.java`.

All newly generated instructions can be transformed into lexical templates by `OpDumpPass.java`, and then into strings by `LexToStrPass.java`. The reverse is also true for parsing a string into the operation by `StrToLexPass.java` and `OpBuilderPass.java`.

That is, there is an interoperability bridge between the configuration instructions used in real scenarios and the abstract operational instructions. An operation instruction is a data structure, while a configuration instruction is just a string.

Key data structures

- **OpType (lexical/OpType.java)**
 - **Description:**
Protocol's configuration operation type.
 - **Key Points:**
 - It includes router operations and interface operations.
 - All set operations have their corresponding unset operations.
- **LexDef (lexical/LexDef.java)**

- **Description:**

Parsing template strings, extracting parameter and range information.

- **Key Points:**

- An operation corresponds to a template string: `LexDef`. The attribute `preprocess` represents the mapping of operations and lexical template. That is, a lexical template represents an instruction. Since instructions can have optional arguments, such as `num`, `name`, etc., a lexical template corresponds to multiple specific instructions.
- `LexDef` contains four parts, regular expression: `Re`, template string: `Template`, parameter list: `args`, parameter value range: `argsRange`. With these four attributes, it is possible to determine a specific, executable operation instruction.
- `Re` ensures that only the parameter information in an operation is used, without affecting other fields. `Template` ensures that the desired operation is used. `args` includes multiple types: `num`, `name`, `ID`, `IP`, `IPRANGE`, etc. In a template string, the content of the parentheses immediately following the parameter type is the range of the parameter, but some parameters do not have ranges.
- There may be more than one parameter of the same type in operation instruction. For example, there may be more than one `num`, but each may have a different range of values. To achieve exact matching, we use `num`, `num2`, etc., to distinguish between them.
- For an instruction whose structure is a string, `LexDef` first matches the corresponding instruction type with a regular expression, then extracts the parameter information in it and stores it in a data structure. On the other hand, for an operation, `LexDef` finds the corresponding lexical template by its instruction type, and then fills it with parameter information to get a complete instruction string.

How to add a new protocol to frontend?

Determine the instructions that need to be generated based on the protocol's development documentation. Then define the operation types and lexical templates for these instructions. Update `OpType.java` and `LexDef.java`.

- `OpType.java`: The type of operation for each instruction needs to be defined by yourself and is usually expressed in the capitalized form of the instruction. The instructions need to be divided into router operations and interface operations. Please design the division function in the last part of the document

- `LexDef.java` : Define its corresponding lexical template according to the type of operation defined in a. For the parameter part of the instruction, convert it to the specified way. For example, `(1-120)` is converted to `{NUM(1-120)}`. `ip ospf network (broadcast|non-broadcast)` is converted to `ip ospf network {NAME(broadcast,non-broadcast)}`.
- You can define new parameter types, such as NET, but you need to modify the documentation that parses the parameter types, and you can search globally to find what needs to be changed.

lib/reducer

The `lib/reducer` module can simplify generated programs by removing redundant or overridden commands. It can also transform simplified generated programs to a `ConfGraph`. And it defines some reduced rules to simplify generated programs.

Structure of reducer module

The `lib/reducer` module consists of three parts, each with a different function.

The overall structure is as follows:

代码块

```

1  lib/reducer/
2  |— driver/
3  |   |— reducer.java          # Entry point for reducer
4  |— pass/
5  |   |— baseExecPass.java     # Record the current interface and router at
   runtime
6  |   |— ospfArgPass.java      # Parse the operation and converting the
   execution effect of the operation to ConfGraph
7  |   |— ospfDaemonExecPass.java # Parse the router operation
8  |   |— ospfIntfExecPass.java  # Parse the interface operation
9  |   |— phyExecArgPass.java    # Parse the physical operation
10 |   |— reducePass.java       # Reduction of the set of operations
11 |— semantic/
12 |   |— ConflictRedexDef.java  # Define conflict reduce rules between
   operations
13 |   |— CtxOpDef.java         # Define the parent operation to which each
   operation type belongs
14 |   |— OverrideRedexDef.java  # Define override reduction rules for the
   operation

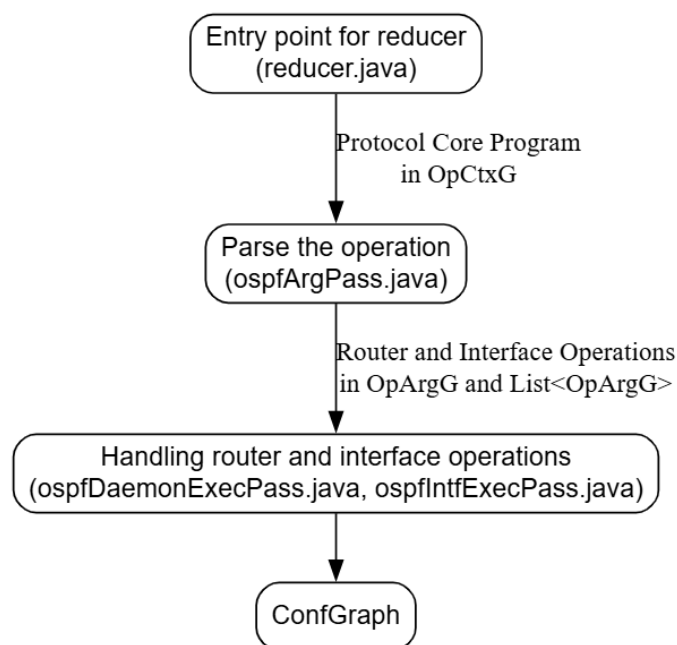
```

Execution flow of reducer module

The reducer module can simplify generated programs by removing redundant or overridden commands. `reducer.java` is an entry point for reducer. It can get simplified generated programs or transform them into a `ConfGraph`.

`reducePass.java` is to simplify generated programs by removing redundant or overridden commands.

`ospfArgPass.java` transforms them into a `ConfGraph`. It needs to use `ospfDaemonExecPass.java` and `ospfIntfExecPass.java`.



`semantic/` defines some reduced rules to simplify generated programs

Key data structures

- **ConflictRedexDef (semantic/ConflictRedexDef.java)**
 - **Description:**
Define conflict reduce rules between operations
 - **Key Points:**
 - Define conflict rules. If the operation creates a conflict, it should be removed.
- **OverrideRedexDef (semantic/OverrideRedexDef.java)**
 - **Description:**
Define override reduction rules for the operation
 - **Key Points:**

- Define override rules. The effect of the latter command can override the previous one.
- **UnsetRedexDef (semantic/UnsetRedexDef.java)**
 - Define the mapping between the Set Operation and the corresponding Unset Operation.
 - **Key Points:**
 - Define unset rules. Unset Operation can undo the effect of the corresponding instruction.

How to add a new protocol to generator?

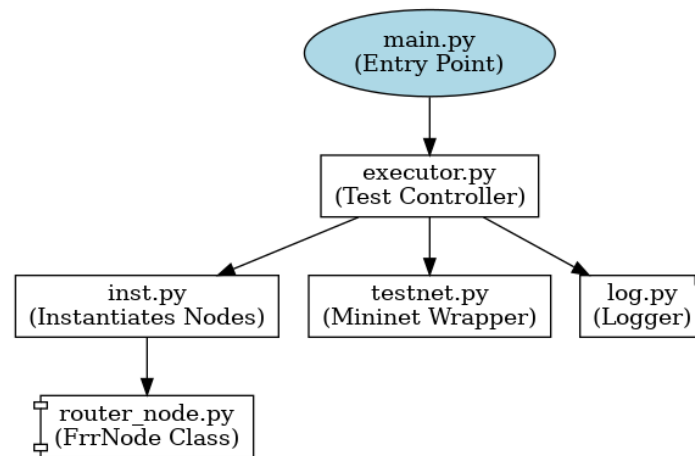
1. Add new protocol in `reducer.java`.
2. Update `baseExecPass.java` Add the current interface and router at runtime for new protocol. Update `phyExecArgPass.java`. Add parsing of physical node operations for new protocols.
3. Add `<proto>ArgPass.java`, `<proto>DaemonExecPass.java` and `<proto>IntfExecPass.java`, etc. `<proto>DaemonExecPass.java` parses the router operation and `<proto>IntfExecPass.java` parses the interface operation. The effects of the execution of all operations should correspond to the ConfGraph, making sure that the ConfGraph and the set of operations are corresponding. It is necessary to iterate through all the operation types, defining separately the effects they produce on the ConfGraph.
4. Update `ConflictRedexDef.java`. Conflict rules are customized to the specifics of the protocol operation. Update `OverrideRedexDef.java`. Override rules are customized to the specifics of the protocol operation. Update `UnsetRedexDef.java` and `CtxOpDef.java` for new protocol.

Executor

The executor is responsible for simulating the network and running protocol implementations inside Docker containers based on Mininet.

This document explains how to extend the ToDiff executor (`restful_mininet`) to support the new routing protocol. We take OSPF protocols as an example.

Structure of executor

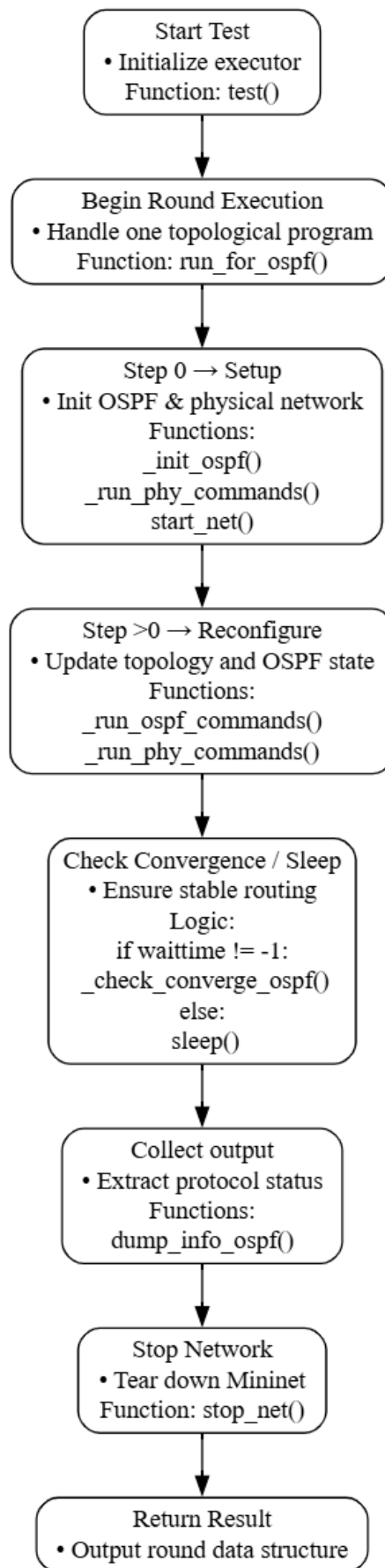


As shown in the figure, the `restful_mininet` module consists of several subcomponents:

- **Top-level**
 - `main.py` : Entry point for CLI execution. Passes arguments (e.g., test file, protocol) into the `executor`.
- **exec/ : Execution Logic**
 - `executor.py` : Main controller for protocol testing. Reads test configuration, sets up topology, and launches daemons.
 - `inst.py` : Parse the network build commands to build the testing network.
- **net/ : Mininet-Based Network Construction**
 - `testnet.py` : Initializes a Mininet network, manages router/host/switch objects.
- **node/ : Node Definitions**
 - `router_node.py` : Core class for FRRouting router nodes. Responsible for creating `/etc/frr`, writing configuration files, and launching routing daemons.
 - `host_node.py`, `switch_node.py` : Host and switch classes
- **util/ : Utility Functions**
 - `log.py` : Color-coded logging wrappers for Mininet.

Execution flow of OSPF protocol

Then we provide a detailed explanation of how the OSPF routing protocol is tested in the executor. The overall logic is shown in the following graph.



1. Entry Point: `executor.test()`

The testing process begins with the invocation of the `test()` method in the `executor` class. This method serves as the orchestration point for executing one or more rounds of

differential testing.

In each round, the executor update of physical network and topological programs defined in test case `testXXXX.json`.

Depending on the specified protocol, a protocol-specific round handler is invoked. For OSPF, the method is:

代码块

```
1 executor._run_for_ospf(r)
```

where `r` denotes the current round number.

2. Input: test case from generator

Each test case(i.e, `testXXXX.json`) consists of multiple test rounds. Each test round corresponds to a single topological program, which defines the sequence of physical and protocol topological commands required to simulate a specific network configuration and test scenario.

- The configuration file provides `commands[r]` for each round.
- Each `commands[r]` includes multiple steps that simulate changes over time.
- These include:
 - `phy` : physical topology commands
 - `ospf` : OSPF command sequences
 - `waitTime` : stabilization delay

3. Step-by-Step testing workflow

Each test round consists of multiple testing steps. Each step we update simulated network via physical topology commands and update topological program to the routing implementation under test.

The testing logic is in `_run_for_ospf(r)` function which distinguishes between the first step and subsequent steps.

- **Step 0: Initial Topology and Protocol Setup**
 - **Function:** ``executor._init_ospf(router_name, ospf_ops)``
 - Writes `ospfd.conf` to `/etc/frr/`
 - Registers `ospfd` in `/etc/frr/daemons`
 - Prepares daemon for launch

- **Function:** ``executor._run_phy_commands(net, ctx, commands[i]['phy'])``
 - Applies physical topology configuration. This internally calls:

代码块

```
1 MininetInst.run()
```

from **File:** ``restful_mininet/exec/inst.py``, which parses and applies:

- `node` commands: add/remove routers or switches
- `intf` commands: manage interfaces and IPs
- `link` commands: manage connectivity

- **Function:** ``TestNet.start_net()``
 - Starts the Mininet-based virtual network

• Step > 0: Dynamic Reconfiguration

- **Function:** ``_run_phy_commands(...)``
 - Applies dynamic network modifications
- **Function:** ``_run_ospf_commands(net, router_name, ospf_ops)``
 - Injects updated OSPF configurations using `vtysh`

4. Wait Phase

- **Condition:** ``commands[i]['waitTime'] == -1``
 - If true, executor assumes implicit convergence behavior and explicitly invokes:

代码块

```
1 self._check_converge_ospf(net)
```

- **Function:** ``_check_converge_ospf(net)``
 - Checks convergence of OSPF protocol by querying route states and LSA propagation
 - If false, a fixed delay is applied:

```
time.sleep(waitTime)
```

5. Output: Result Collection Phase

- **Function:** `FrrNode.dump_info_ospf()` in `router_node.py``
 - Extracts the current OSPF state from each router

The collected data is stored under the `res[i]['watch']` structure, which is then written to the `testXXXXX_res.json`` file.

Key structure: topological commands

Physical topological commands

Physical topological commands define operations on the network topology simulated by Mininet. These commands are interpreted by the executor through the `MininetInst` class in `restful_mininet/exec/inst.py`.

Each command is a string using a domain-specific syntax with the following categories:

(1) Node Commands

Syntax:

代码块

```
1 node <node_name> <type> <add|del|set> [...]
```

- `<node_name>`: name of the node, e.g., `r1`, `s1`, `h1`
- `<type>`: one of `router`, `switch`, or `host`
- `<action>`:
 - `add`: add the node to the network
 - `del`: remove the node
 - `set`: set the protocol implementation running in the device.

Example:

```
1 node r1 router add
2 node r1 router set OSPF up
```

- Adds router `r1` and enables OSPF on it.

(2) Interface Commands

Syntax:

代码块

```
1 intf <node_name> <intf_name> <add|del|up|down|set> [...]
```

- `<intf_name>`: should follow the format `<node_name>-ethX`
- `<action>`:
 - `add` / `del`: add or remove the interface
 - `up` / `down`: bring interface up/down
 - `set net <ip>`: assign IP address to the interface

Example:

代码块

```
1 intf r1 r1-eth0 add
2 intf r1 r1-eth0 set net 10.0.0.1/24
```

(3) Link Commands

Syntax:

代码块

```
1 link <node1> <intf1> <node2> <intf2> <add|del|up|down|set> [...]
```

- Creates or configures a virtual link between two interfaces.

Example:

代码块

```
1 link r1 r1-eth0 r2 r2-eth0 add
2 link r1 r1-eth0 r2 r2-eth0 up
```

- Adds and enables a link between `r1` and `r2`.

Protocol topological commands

The protocol topological commands(topological program) are split into multiple parts and input to the routing implementation step by step.

How to add a new Protocol to executor?

We take OSPF as an example to list all the functions need to be added to the executor.

Function Overview Table

File	Function	Description
<code>executor.py</code>	<code>test()</code>	Entry point that dispatches to the OSPF test logic based on the protocol argument.
<code>executor.py</code>	<code>_run_for_ospf(r)</code>	Executes a full test round (i.e., one topological program) for OSPF.
<code>executor.py</code>	<code>_init_ospf(router_name, ospf_commands)</code>	Writes OSPF config files and prepares the daemon in <code>/etc/frr/</code> .
<code>executor.py</code>	<code>_run_ospf_commands(net, router_name, ospf_commands)</code>	Applies live OSPF configuration using <code>vttysh</code> .
<code>executor.py</code>	<code>_check_converge_ospf(net)</code>	Verifies convergence through FRRouting commands.
<code>router_node.py</code>	<code>dump_info_ospf()</code>	Extracts protocol state (LSAs, neighbors, interfaces) from each router.
<code>router_node.py</code>	<code>stop_ospfd()</code>	Stops the OSPF daemon cleanly.
<code>router_node.py</code>	<code>load_ospf()</code>	

		Starts the OSPF daemon using <code>/etc/frr/ospfd.conf</code> .
<code>inst.py</code>	<code>_run_node_cmd(tokens)</code>	Parses commands like <code>node r1 router set OSPF up/down</code> and invokes <code>load_ospf()</code> or <code>stop_ospfd()</code> .

Writing Support for a New Protocol <proto>

You should follow the logic shown in OSPF and do the minimal change as possible as you can.

1. ``executor.py``

- Add `_run_for_<proto>()`: The main testing logic for the protocol.
- Add `_init_<proto>()`: Writes configuration file (e.g., `/etc/frr/eigrpd.conf`).
- Add `_run_<proto>_commands()`: Injects live configuration via `vttysh`.
- Add `_check_converge_<proto>()`: (Optional) Checks convergence conditions specific to the protocol.

2. ``router_node.py``

- Add `load_<proto>()`: Start the protocol daemon using its config.
- Add `stop_<proto>d()`: Stop the daemon cleanly.
- Add `dump_info_<proto>()`: Extract meaningful state to JSON from `vttysh`.

3. ``inst.py``

- Extend `_run_node_cmd()` to recognize:

代码块

```
1 node r1 router set <PROTO> up
2 node r1 router set <PROTO> down
```

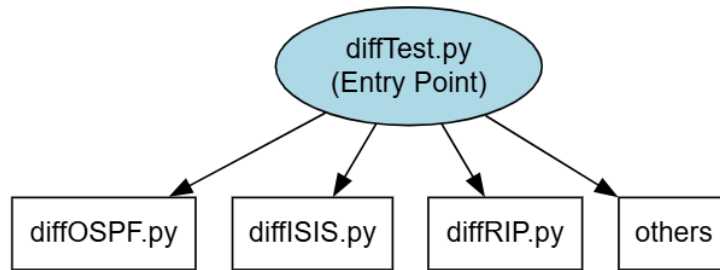
- Route to `load_<proto>()` and `stop_<proto>d()` accordingly.

Checker

The checker is responsible for comparing the results of running multiple equivalent topological programs and outputting the differences in the results.

This document explains how to extend the ToDiff checker (`restful_mininet`) to support the new routing protocol. We take OSPF protocols as an example.

Structure of checker



- `diffTest.py` : Entry point for checker. Passes parameters (e.g., `protocol`, `short_circuit`) into the `checker`.
- `diffOSPF.py` et al. Execute the corresponding difference function according to the specified protocol parameter.

Execution Flow of OSPF Protocol

1. Entry Point: `diffTest.checkTests()`

When executing the `diffTest.py` file, it first calls function `checkTests`. This function reads the test results of all the executors and then calls the difference functions for the different protocols according to the specified parameter: `protocol`. Parameter `short_circuit` refers to "Stop checking as soon as the first diff is found".

2. Input: Test results of all the executors

Each test result (i.e. `testXXXXX_res.json``) is generated by the corresponding test case (i.e. `testXXXX.json``). It includes test case and output information.

Output information includes:

- Daemon: Show information on a variety of general `protocol` and configuration information (Different protocols have different kinds of information).
- Interface: Show state and configuration of `protocol` the specified interface or all interfaces.
- Neighbor: Show state and information of `protocol` specified neighbor or all neighbors.
- Route: Show the routing table, as determined by the most recent SPF calculation.
- Running-config: Show the configuration information of router.
- Other protocol-specific information.

3. Executing the corresponding difference function

Depending on the protocol, different programs under folder `diffTestUtil` are executed.

For example, for the OSPF protocol, it executes `diff0SPF.py`.

The `checkTest` function is first executed in the `diff0SPF.py` file.

For each test round, the output information is compared differentially. The comparisons are of the output information listed in the previous step. Different kinds of information require special comparison functions. For the OSPF protocol, it needs to run `check_runningConfig`, `check_ospfIntfs`, `check_neighbors`, etc.

Before comparing information, it is necessary to remove information entries that change when executed differently. For the OSPF protocol, it needs to run `shrink_ospfDaemon`, `shrink_ospfIntfs`, etc.

4. Output: Result Collection Phase

The results after differential testing are written in the ``testXXXXX.json.txt`` file.

The information shown in the file is what makes the difference in the test results.

How to add a new Protocol to checker?

We take OSPF as an example to list all the functions that need to be added to the checker.

Function Overview Table

File	Function	Description
<code>diffTest.py</code>	<code>checkTests()</code>	Reading the test results of all the executors and then calling the difference functions for the different protocols according to the specified parameter: <code>protocol</code> .
<code>diffTestUtil/ diff<proto>.py</code>	<code>checkTest()</code>	Comparing various information in test results.
<code>diffTestUtil/ diff<proto>.py</code>	<code>check_<proto>_XXXX()</code>	Comparing specific information in test results.
<code>diffTestUtil/ diff<proto>.py</code>	<code>shrink_<proto>_XXXX()</code>	Deleting information entries that change when executed differently.

Writing Support for a New Protocol <proto> (e.g., EIGRP)

You should follow the logic shown in OSPF and do the minimal change as possible as you can.

1. ``diffTest.py``

- Add <proto> in choices
- Add Conditional Statements <proto> in `checkTests()`

2. ``diffTestUtil/diff<proto>.py``

- Add new file(e.g., diffEIGRP.py)
- `checkTest()` for EIGRP.
- Add `check_<proto>_XXXX()` .Comparing specific information in test results.Here different functions are added depending on the protocol.
- Add `shrink_<proto>_XXXX()` .Deleting information entries that change when executed differently.Which information needs to be deleted needs to be determined based on the output of the protocol run.