ToDiff Developer's Guide

ToDiff

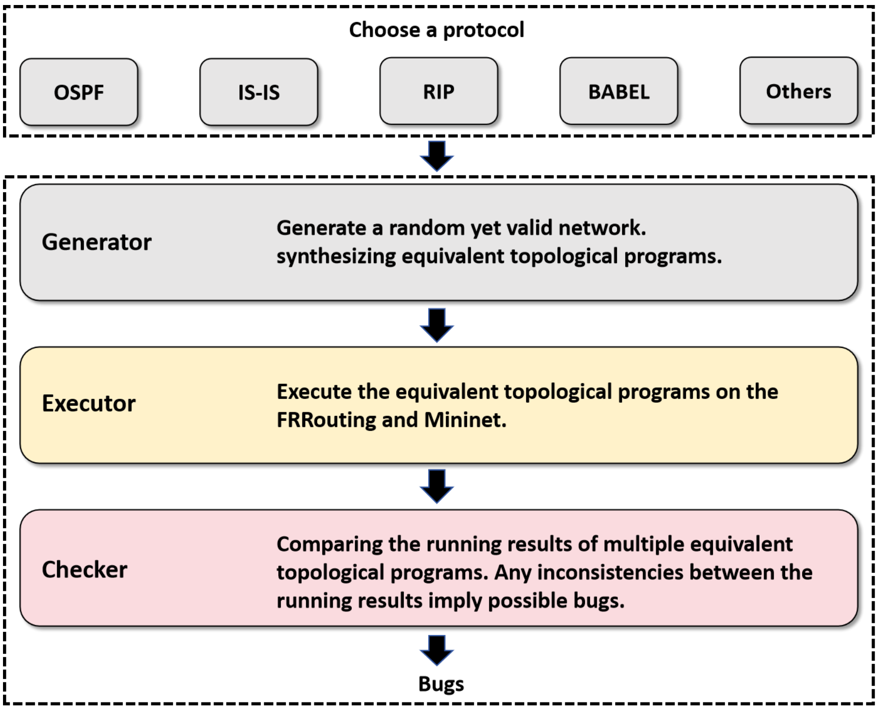
ToDiff is a prototype implementation of the technique known as ToDiff. It aims to validate Interior Gateway Protocols(a key class of routing protocols) via equivalent topology synthesis.

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 and show how to modify these components to support validing a new protocol.

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`

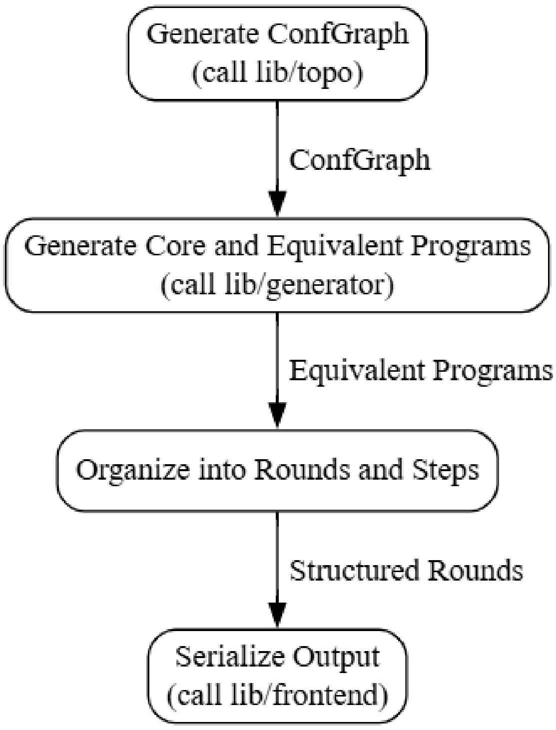
The output of the generator is the testcase consisting of multiple equivalent topological programs for testing, and will be simulated in executor.

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.

The main purpose of diffTopo is to generate multiple equivalent programs for a given network configuration and organize them into rounds and steps for controlled testing.

Execution flow



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 configuration commands into rounds.
  - Within each round, organize commands into steps with wait times for protocol convergence or transition.

4. **Serialize Output**
   - 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

```
{
    "geninfo": { ... },
    "conf_name": "test1223444.json",
    "step_nums": [1, 3],
    "round_num": 2,
    "routers": ["r0", "r1"],
    "commands": [round0, round1]
}
```

- **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

```
{
```

```
    "core_commands": {
        "router_name": "commands_str"
    },
    "configGraph": "DOT format string",
    "configGraphAttr": "DOT format string",
    "evaluate": {
        "genGraphTime": 0.123,
        "genEqualTime": 0.456,
        "totalTime": 0.789,
        "totalInstruction": 120
    }
}
```

- **core_commands**:
  Mapping from router names to their core configuration commands (pre-equivalent program).
- **configGraph**:
  Physical network topology, represented in DOT graph format.
- **configGraphAttr**:
  Configuration graph (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

```
{
    "step": 0,
    "waitTime": "2s",
    "phy": ["physical layer commands"],
    "ospf": ["ospf layer commands"]
}
```

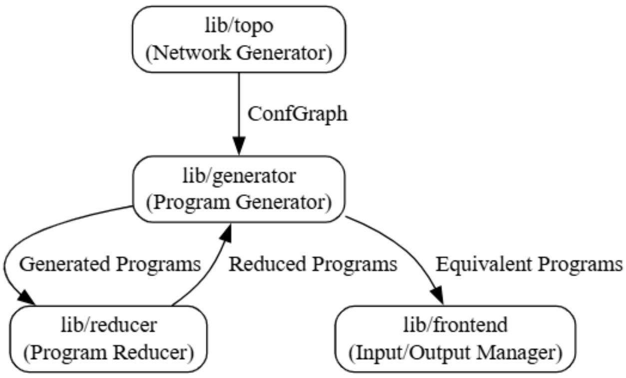
- **step**:
  The index of the step within the round.
- **waitTime**:
  Time to wait after executing this step:
  - `"2s"` means a fixed wait of 2 seconds.
  - `"-1s"` means wait until protocol convergence is detected.
- **phy**:
  List of physical configuration commands applied in this step.
- **ospf(<proto>)**:
  List of OSPF protocol configuration commands applied in this step.

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 components:

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

Their interaction is illustrated below:



- **lib/topo (Network Generator)**
  - Generates randomized or customized network topologies (`ConfGraph`).
  - Encodes both physical connectivity and protocol-specific configurations.
  - Provides the initial input for program generation.

- **lib/generator (Program Generator)**
  - Transforms a given `ConfGraph` into **core** and **equivalent topological programs**.
  - Initially generates programs based on `ConfGraph`.
  - Iteratively interacts with `lib/reducer` to generate equivalent programs
- **lib/reducer (Program Reducer)**
  - Simplifies generated programs by removing redundant or overrided commands.
  - Provides **reduced versions** of programs to `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

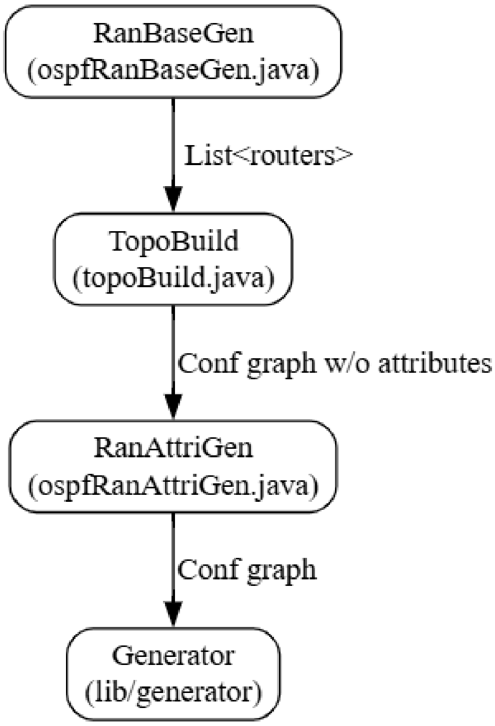
```
lib/generator/topo/
├── driver/
│   └── topo.java          # Entry point for topology generation
├── item/
│   └── base/
│       ├── Intf.java      # Basic Interface class
│       ├── Router.java    # Basic Router class
├── pass/
    ├── attri/
    │   ├── ospfRanAttriGen.java   # OSPF-specific attribute generator
    │   ├── <protol>RanAttriGen.java # Other protocol attribute generator
    ├── base/
    │   ├── ospfRanBaseGen.java    # OSPF-specific base topology generator
    │   ├── <protol>RanBaseGen.java    # other protocol base topology generator
```

```
    ├── build/
    │   └── topoBuild.java        # Topology assembly and ConfGraph 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:
  - `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` assembles base topology and attributes into a Conf Graph.

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.

Stage	Description	Src File	Output
1	Generate base physical topology and protocol logical areas	base/ospfRanBaseGen.java	List<Routers>
2	Build detailed topology graph structures (via topoBuild)	build/topoBuild.java	ConfGraph w/o attributes
3	Assign additional protocol attributes (via RanAttriGen)	attri/ospfRanAttriGen.java	ConfGraph with attributes

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 nodes (e.g., Routers, Switches, Hosts)
- Protocol configuration nodes (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 needed for differential testing.

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

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

The `node/` subdirectory defines all the **node types** that appear in the ConfGraph.
It is 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.

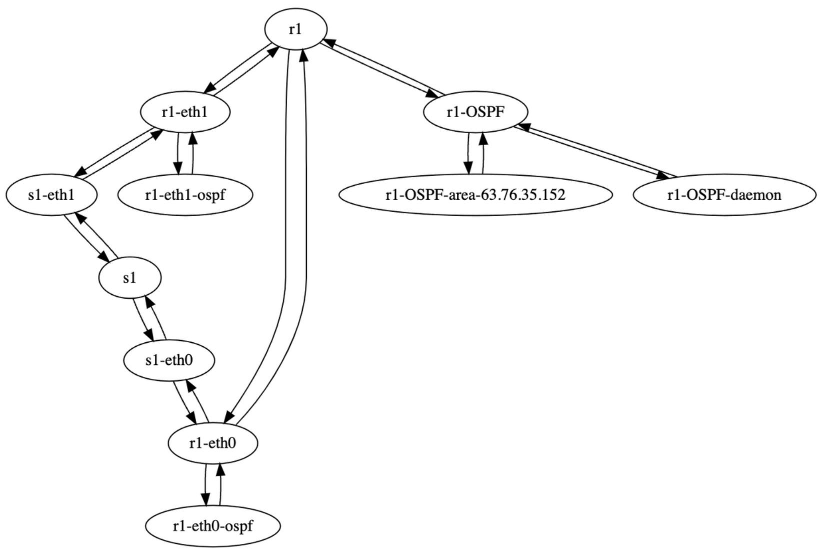
Taking `node/ospf/` as an example:

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 this ConfGraph, we have one router `r1` and one switch `s1`. `r1` has two interface `r1-eth0` and `r1-eth1`, `s1` has two interface `s1-eth0` and `s1-eth1`. Interface `r1-eth0` is connected to `s1-eth0` and `r1-eth1` is connectd to `s1-eth1`, the edge Type is physical connectd. Router `r1` has one OSPF daemon, thus it connects to node `r1-OSPF` with edge type of 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

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

```
lib/generator/
├── driver/
│   └── generate.java         # Main driver to orchestrate the generation process
├── phy/
│   ├── pass/
│   │   ├── genPhyCorePass.java  # Generate core physical operations from ConfGraph
│   │   └── genPhyEqualPass.java # Generate equivalent physical operations
├── ospf/
│   ├── pass/
```

```
│   │   ├── genCorePassOspf.java  # Generate core OSPF protocol operations from ConfGraph
│   │   ├── genEqualPass.java     # Generate equivalent OSPF operations
│   │   ├── shrinkCorePass.java   # Shrink and optimize core operation sequences

```

Execution flow of generator module

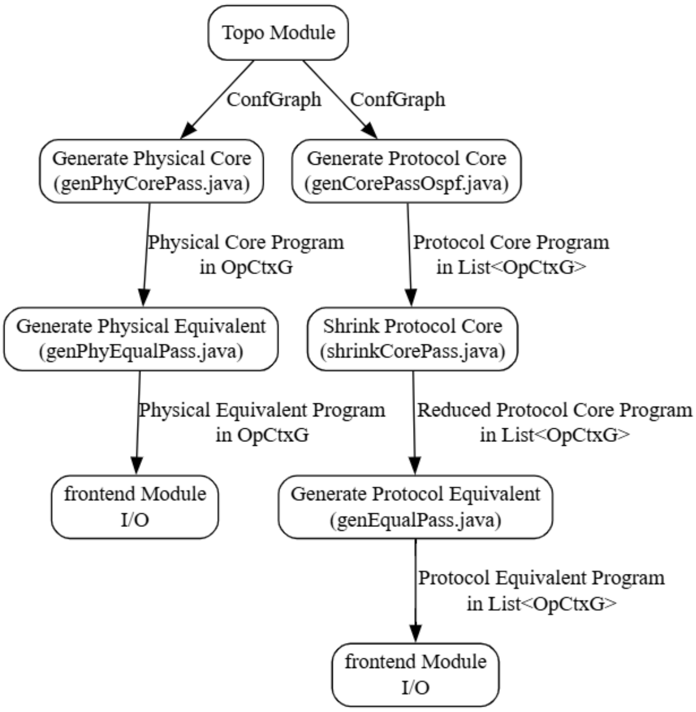
The `lib/generator` module transforms a given `ConfGraph` into executable equivalent topological programs through a structured sequence of passes.
The goal is to generate configuration programs that correctly set all attributes specified in the ConfGraph, ensuring that the resulting network configuration matches the ConfGraph exactly.

The entire generation process is divided into two independent layers:

- **Physical Layer**:
  Responsible for generating configurations related to physical topology, including device creation, link establishment, and interface activation.
- **Protocol Layer**:
  Responsible for generating configurations related to routing protocols (e.g., OSPF), such as protocol-specific parameters.

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

- **ConfGraph → Core Programs**:
  From a single ConfGraph, **multiple Core Programs** can be derived.
  A **Core Program** is the minimal set of configuration commands necessary to fully realize the intended network state.
- **Core Programs → Equivalent Programs**:
  Each Core Program is expanded into **multiple Equivalent Programs** by injecting safe intermediate changes, 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 basic physical operations like adding routers, switches, links, and activating interfaces to establish the topology described in ConfGraph.
  - **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 non-destructive 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 (`genCorePassOspf.java`)**
  - **Input**: ConfGraph
  - **Output**: Protocol core programs (`List<OpCtxG>`)
  - **Description**:
    Generate minimal OSPF configuration commands 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:
      - `interface r1-eth0 ip address 1.1.1.1; ip ospf area 1`
      - `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 `ROSPF` 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 safe configuration changes (temporary value modifications, removals, reactivations) while guaranteeing that the final protocol attributes remains equivalent to the original ConfGraph.
  - **Details:**

    Use an equivalent synthesis algorithm (See Pesudo 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 configuration 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-specific configuration operations.
- **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 topology configuration operations.
  - **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-level 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 CLI commands based on the protocol's specification.
2. **Implement physical-level 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

Structure of frontend module

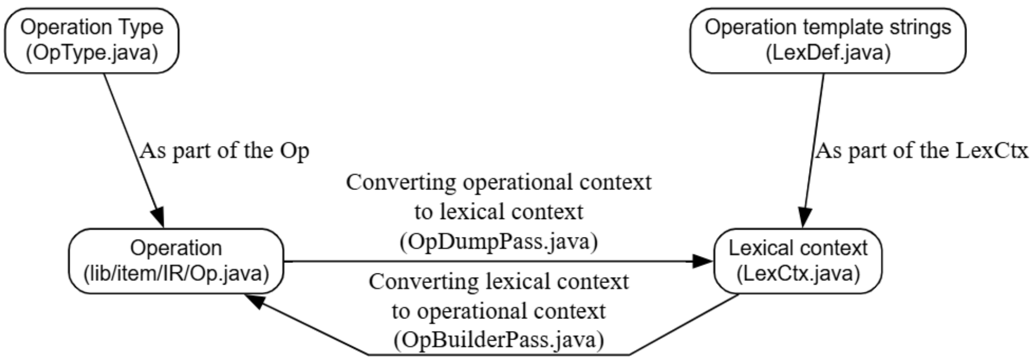
The `lib/frontend` module consists of three parts, each with a different function.

The overall structure is as follows:

代码块

```
lib/frontend/
├── driver/
│   └── IO.java         # Read and write operation
├── lexical/
│   ├── LexDef.java   # Parse template strings, extracting parameter and range information
│   └── OpType.java  # Define the types of OSPF operations
├── pass/
│   ├── LexToStrPass.java  # Convert lexical contexts to strings
│   ├── OpDumpPass.java     # Convert an operational context to a lexical context
│   ├── OpBuilderPass.java  # Convert lexical contexts to operational contexts
│   ├── 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`.

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.

How to add a new protocol to generator?

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`.

Here are some notes：

- All set operations should have their corresponding unset operations for `OpType.java` and `LexDef.java`.
- The operations need to be divided into router operations and interface operations.
- 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

1. 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

```
lib/reducer/
├── driver/
│   └── reducer.java          # Entry point for reducer
├── pass/
```

```
│   ├── baseExecPass.java   # Record the current interface and router at runtime
│   ├── ospfArgPass.java    # Parse the operation and converting the execution effect of the operation to ConfGraph
│   ├── ospfDaemonExecPass.java  # Parse the router operation
│   ├── ospfIntfExecPass.java # Parse the interface operation
│   ├── phyExecArgPass.java    # Parse the physical operation
├── semantic/
│   ├── ConflictRedexDef.java  # Define conflict reduce rules between operations
│   ├── CtxOpDef.java    # Define the parent operation to which each operation type belongs
│   ├── OverrideRedexDef.java  # Define override reduction definitions for the operation
│   ├── UnsetRedexDef.java   # Define the mapping between the Set Operation and the corresponding Unset Operation.
```

Execution flow of reducer module

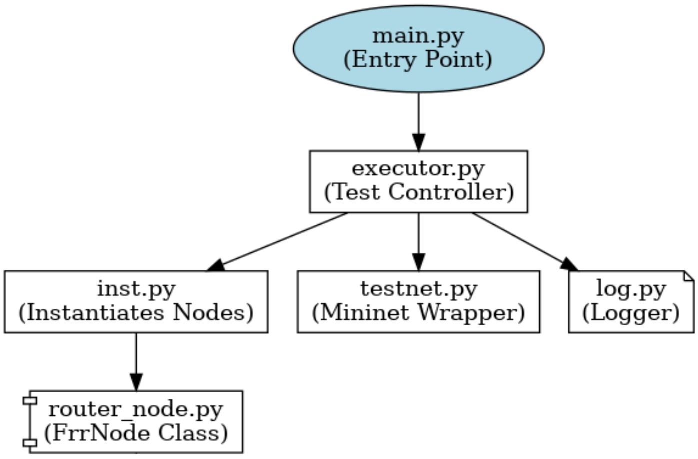
The reducer module

Executor

The executor is responsible for simulating the network and running protocol daemons 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

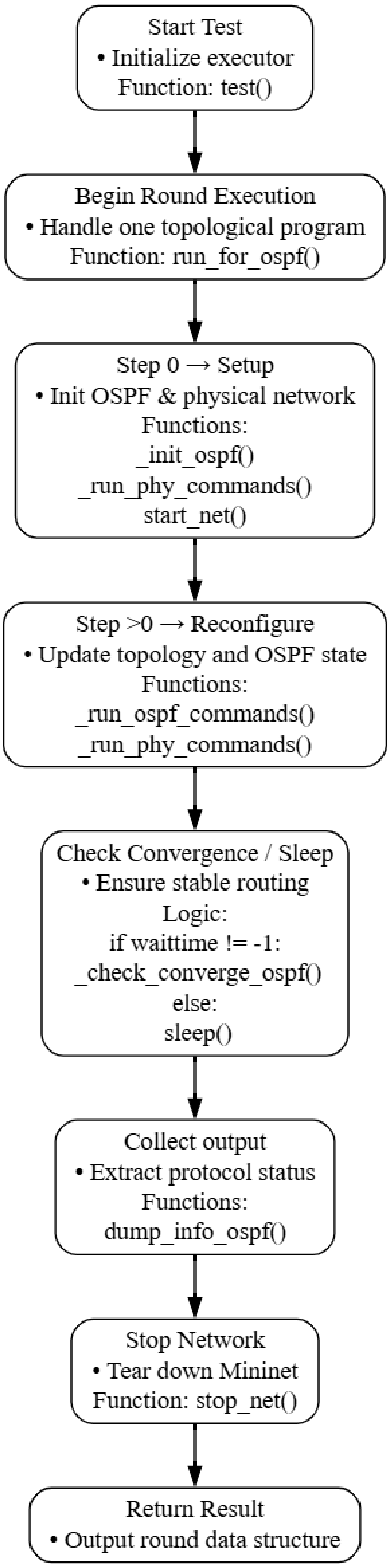


As shown in the figure, the `restful_mininet` module consists of several subcomponents:

- **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` : Prase 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 and academically structured 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

```
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 operations 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 toplogy 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:

  `python      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 Phse**

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

代码块

```
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:

    代码块

    ```
    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: topology commands

Physical topology commands

physical toplogy commands (also referred to as `phy` 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

```
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`: change the state or properties

Example:

代码块

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

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

(2) Interface Commands

Syntax:

代码块

```
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

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

(3) Link Commands

Syntax:

代码块

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

- Creates or configures a virtual link between two interfaces.

Example:

代码块

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

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

Protocol topology commands

The protocol topology commands(a.k.a topological program) are split into multiple parts and input to the routing implemenation 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
`executor.py`	`test()`	Entry point that dispatches to the OSPF test logic based on the protocol argument.
`executor.py`	`_run_for_ospf(r)`	Executes a full test round (i.e., one topological program) for OSPF.
`executor.py`	`_init_ospf(router_name, ospf_commands)`	Writes OSPF config files and prepares the daemon in `/etc/frr/`.
`executor.py`	`_run_ospf_commands(net, router_name, ospf_commands)`	Applies live OSPF configuration using `vtysh`.
`executor.py`	`_check_converge_ospf(net)`	Verifies convergence through FRRouting commands.
`router_node.py`	`dump_info_ospf()`	Extracts protocol state (LSAs, neighbors, interfaces) from each router.
`router_node.py`	`stop_ospfd()`	Stops the OSPF daemon cleanly.
`router_node.py`	`load_ospf()`	Starts the OSPF daemon using `/etc/frr/ospfd.conf`.

`inst.py`	`_run_node_cmd(tokens)`	Parses commands like `node r1 router set OSPF up/down` and invokes `load_ospf()` or `stop_ospfd()`.

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. **`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 `vtysh`.
- 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 `vtysh`.

3. **`inst.py`**

- Extend `_run_node_cmd()` to recognize:

  代码块

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

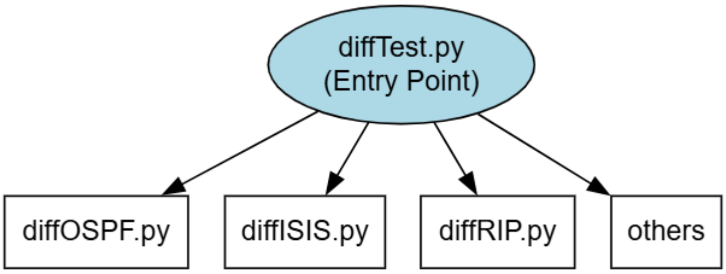
- Route to `load_<proto>()` and `stop_<proto>d()` accordingly.

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 `diffOSPF.py` .

The `checkTest` function is first executed in the `diffOSPF.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
`diffTest.py`	`checkTests()`	Reading the test results of all the executors and then calling the difference functions for the different protocols according to the specified parameter: `protocol`.
diffTestUtil/ `diff<proto>.py`	`checkTest()`	Comparing various information in test results.
diffTestUtil/ `diff<proto>.py`	`check_ospf_XXXX()`	Comparing specific information in test results.
diffTestUtil/ `diff<proto>.py`	`shrink_ospf_XXXX()`	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.