TR181 Node Comparator User Guide

Table of Contents

1. Introduction
2. Installation
3. Quick Start
4. Comparison Scenarios
5. Configuration
6. Command Line Interface
7. Advanced Usage
8. Best Practices

Introduction

The TR181 Node Comparator is a powerful tool for analyzing and comparing TR181 data model implementations across different sources. It supports three primary comparison scenarios:

1. **CWMP vs Custom Subset**: Compare TR181 nodes from a CWMP source against your custom subset definitions
2. **Custom Subset vs Device**: Validate that a device correctly implements your custom TR181 subset
3. **Device vs Device**: Compare TR181 implementations between two different devices

Installation

Prerequisites

- Python 3.8 or higher
- pip package manager

Install from Source

```
git clone <repository-url>
cd tr181-node-comparator
pip install -e .
```

Install Dependencies

```
pip install -r requirements.txt
```

Quick Start

1. Basic CWMP Extraction

```
import asyncio
from tr181_comparator import CWMPExtractor

async def extract_cwmp_nodes():
    # Configure CWMP connection
    config = {
```

```
        'endpoint': 'http://192.168.1.1:7547/cwmp',
        'username': 'admin',
        'password': 'admin123',
        'timeout': 30
    }

    # Create extractor and extract nodes
    extractor = CWMPExtractor(config)
    nodes = await extractor.extract()

    print(f"Extracted {len(nodes)} TR181 nodes")
    for node in nodes[:5]:  # Show first 5 nodes
        print(f"  {node.path}: {node.data_type} ({node.access})")

# Run the extraction
asyncio.run(extract_cwmp_nodes())
```

2. Create a Custom Subset

```
from tr181_comparator import SubsetManager, TR181Node, AccessLevel

async def create_custom_subset():
    # Define custom nodes
    nodes = [
        TR181Node(
            path="Device.WiFi.Radio.1.Channel",
            name="Channel",
            data_type="int",
            access=AccessLevel.READ_WRITE,
            description="WiFi channel number"
        ),
        TR181Node(
            path="Device.WiFi.AccessPoint.1.SSID",
            name="SSID",
            data_type="string",
            access=AccessLevel.READ_WRITE,
            description="WiFi network name"
        )
    ]

    # Save subset
    subset_manager = SubsetManager("my_wifi_subset.json")
    await subset_manager.save_subset(nodes)
    print("Custom subset saved successfully")

asyncio.run(create_custom_subset())
```

3. Basic Comparison

```
from tr181_comparator import ComparisonEngine, CWMPExtractor, SubsetManager

async def basic_comparison():
    # Extract from CWMP
```

```
    cwmp_extractor = CWMPExtractor({
        'endpoint': 'http://device.local:7547/cwmp',
        'username': 'admin',
        'password': 'password'
    })
    cwmp_nodes = await cwmp_extractor.extract()

    # Load subset
    subset_manager = SubsetManager('my_wifi_subset.json')
    subset_nodes = await subset_manager.extract()

    # Compare
    engine = ComparisonEngine()
    result = await engine.compare(cwmp_nodes, subset_nodes)

    # Display results
    print(f"Comparison Summary:")
    print(f"  Total CWMP nodes:{result.summary.total_nodes_source1}")
    print(f"  Total subset nodes:{result.summary.total_nodes_source2}")
    print(f"  Common nodes:{result.summary.common_nodes}")
    print(f"  Differences:{result.summary.differences_count}")

    if result.only_in_source1:
        print(f"\nNodes only in CWMP ({len(result.only_in_source1)}):")
        for node in result.only_in_source1[:5]:
            print(f"  {node.path}")

    if result.differences:
        print(f"\nDifferences found ({len(result.differences)}):")
        for diff in result.differences[:5]:
            print(f"  {diff.path}: {diff.property} differs")

asyncio.run(basic_comparison())
```

Comparison Scenarios

Scenario 1: CWMP vs Custom Subset

This scenario helps you understand which standard TR181 nodes are available in your CWMP device but not included in your custom subset.

```
async def cwmp_vs_subset_comparison():
    # Configure CWMP source
    cwmp_config = {
        'endpoint': 'http://192.168.1.1:7547/cwmp',
        'username': 'admin',
        'password': 'admin123'
    }

    # Extract from both sources
    cwmp_extractor = CWMPExtractor(cwmp_config)
    subset_manager = SubsetManager('device_subset.json')
```

```
    cwmp_nodes = await cwmp_extractor.extract()
    subset_nodes = await subset_manager.extract()

    # Perform comparison
    engine = ComparisonEngine()
    result = await engine.compare(cwmp_nodes, subset_nodes)

    # Analyze missing standard nodes
    print("Standard TR181 nodes missing from your subset:")
    for node in result.only_in_source1:
        if not node.is_custom:  # Only show standard nodes
            print(f"  {node.path} ({node.data_type})")

    # Analyze custom nodes not in CWMP
    print("\nCustom nodes in subset not available via CWMP:")
    for node in result.only_in_source2:
        if node.is_custom:
            print(f"  {node.path} ({node.data_type})")

asyncio.run(cwmp_vs_subset_comparison())
```

Scenario 2: Custom Subset vs Device Implementation

This scenario validates that your device correctly implements the TR181 nodes defined in your custom subset.

```
from tr181_comparator import EnhancedComparisonEngine, HookBasedDeviceExtractor, RESTAP

async def subset_vs_device_validation():
    # Load your custom subset
    subset_manager = SubsetManager('my_device_spec.json')
    subset_nodes = await subset_manager.extract()

    # Configure device connection
    device_config = DeviceConfig(
        name="production_device",
        type="rest",
        endpoint="http://192.168.1.100/api/tr181",
        authentication={
            'type': 'bearer',
            'token': 'your-api-token'
        },
        timeout=30
    )

    # Set up device extractor
    hook = RESTAPIHook()
    device_extractor = HookBasedDeviceExtractor(device_config, hook)
    device_nodes = await device_extractor.extract()

    # Perform enhanced comparison with validation
    enhanced_engine = EnhancedComparisonEngine()
    result = await enhanced_engine.compare_with_validation(
```

```
        subset_nodes,
        device_nodes,
        device_extractor
    )

    # Generate comprehensive report
    summary = result.get_summary()

    print("Device Implementation Validation Report")
    print("=" * 50)
    print(f"Subset nodes: {summary['basic_comparison']['total_differences']}")
    print(f"Device nodes: {summary['basic_comparison']['extra_in_device']}")
    print(f"Missing implementations: {summary['basic_comparison']['missing_in_device']}
    print(f"Validation errors: {summary['validation']['nodes_with_errors']}")
    print(f"Validation warnings: {summary['validation']['total_warnings']}")

    # Show specific validation issues
    if result.validation_results:
        print("\nValidation Issues:")
        for path, validation_result in result.validation_results:
            if not validation_result.is_valid:
                print(f"  ERROR -{path}:")
                for error in validation_result.errors:
                    print(f"    {error}")
            elif validation_result.warnings:
                print(f"  WARNING -{path}:")
                for warning in validation_result.warnings:
                    print(f"    {warning}")

asyncio.run(subset_vs_device_validation())
```

Scenario 3: Device vs Device Comparison

This scenario compares TR181 implementations between two different devices to identify configuration and capability differences.

```
async def device_vs_device_comparison():
    # Configure first device (e.g., development device)
    dev_config = DeviceConfig(
        name="dev_device",
        type="rest",
        endpoint="http://192.168.1.10/api",
        authentication={'type': 'basic', 'username': 'admin', 'password': 'dev123'}
    )

    # Configure second device (e.g., production device)
    prod_config = DeviceConfig(
        name="prod_device",
        type="rest",
        endpoint="http://192.168.1.20/api",
        authentication={'type': 'basic', 'username': 'admin', 'password': 'prod456'}
    )
```

```
    # Extract from both devices
    dev_hook = RESTAPIHook()
    prod_hook = RESTAPIHook()

    dev_extractor = HookBasedDeviceExtractor(dev_config, dev_hook)
    prod_extractor = HookBasedDeviceExtractor(prod_config, prod_hook)

    dev_nodes = await dev_extractor.extract()
    prod_nodes = await prod_extractor.extract()

    # Compare devices
    engine = ComparisonEngine()
    result = await engine.compare(dev_nodes, prod_nodes)

    print("Device-to-Device Comparison Report")
    print("=" * 40)
    print(f"Development device nodes: {result.summary.total_nodes_source1}")
    print(f"Production device nodes: {result.summary.total_nodes_source2}")
    print(f"Common nodes: {result.summary.common_nodes}")
    print(f"Configuration differences: {result.summary.differences_count}")

    # Show capabilities unique to each device
    if result.only_in_source1:
        print(f"\nCapabilities only in development device:")
        for node in result.only_in_source1[:10]:
            print(f"  {node.path}")

    if result.only_in_source2:
        print(f"\nCapabilities only in production device:")
        for node in result.only_in_source2[:10]:
            print(f"  {node.path}")

    # Show configuration differences
    if result.differences:
        print(f"\nConfiguration differences:")
        for diff in result.differences[:10]:
            print(f"  {diff.path}: {diff.source1_value} vs {diff.source2_value}")

asyncio.run(device_vs_device_comparison())
```

Configuration

Configuration File Format

The system supports JSON and YAML configuration files. Here's a complete example:

```
{
  "system": {
    "logging": {
      "level": "INFO",
      "file": "tr181_comparator.log",
      "rotation": "daily"
    },
```

```
  "performance": {
    "max_concurrent_connections": 5,
    "connection_timeout": 30,
    "retry_attempts": 3
  }
},
"devices": [
  {
    "name": "main_gateway",
    "type": "cwmp",
    "endpoint": "http://192.168.1.1:7547/cwmp",
    "authentication": {
      "type": "basic",
      "username": "admin",
      "password": "admin123"
    },
    "timeout": 30,
    "retry_count": 3
  },
  {
    "name": "wifi_ap",
    "type": "rest",
    "endpoint": "http://192.168.1.10/api/tr181",
    "authentication": {
      "type": "bearer",
      "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
    },
    "hook_config": {
      "api_version": "v2",
      "custom_headers": {
        "X-Device-Type": "WiFi-AP"
      }
    }
  }
],
"subsets": [
  {
    "name": "wifi_subset",
    "path": "subsets/wifi_parameters.json",
    "description": "WiFi-related TR181 parameters"
  },
  {
    "name": "device_info",
    "path": "subsets/device_info.yaml",
    "description": "Basic device information parameters"
  }
],
"export": {
  "default_format": "json",
  "include_metadata": true,
  "output_directory": "reports",
```

```
    "timestamp_format": "ISO8601"
  }
}
```

Loading Configuration

```python
from tr181_comparator import SystemConfig

# Load from file
config = SystemConfig.load_from_file('config.json')

# Access configuration
for device_config in config.devices:
    print(f"Device: {device_config.name} ({device_config.type})")
```

Command Line Interface

Basic Usage

```bash
# Compare CWMP source against subset
tr181-compare cwmp-vs-subset \
  --cwmp-endpoint http://device.local:7547/cwmp \
  --cwmp-username admin \
  --cwmp-password password \
  --subset-file my_subset.json \
  --output-format json \
  --output-file comparison_result.json

# Compare subset against device
tr181-compare subset-vs-device \
  --subset-file device_spec.json \
  --device-endpoint http://device.local/api \
  --device-type rest \
  --auth-token your-token \
  --validate \
  --output-format text

# Compare two devices
tr181-compare device-vs-device \
  --device1-endpoint http://dev.local/api \
  --device1-type rest \
  --device1-auth admin:dev123 \
  --device2-endpoint http://prod.local/api \
  --device2-type rest \
  --device2-auth admin:prod456 \
  --output-format xml
```

Advanced CLI Options

```bash
# Use configuration file
tr181-compare --config config.json cwmp-vs-subset \
  --cwmp-device main_gateway \
  --subset wifi_subset
```

```
# Enable detailed logging
tr181-compare --log-level DEBUG --log-file debug.log \
  subset-vs-device --subset-file spec.json --device-name test_device

# Export multiple formats
tr181-compare device-vs-device \
  --device1-name dev_device \
  --device2-name prod_device \
  --output-formats json,xml,text \
  --output-directory reports/
```

Advanced Usage

Custom Validation Rules

```
from tr181_comparator import TR181Validator, ValidationResult, ValueRange

class CustomValidator(TR181Validator):
    def validate_custom_constraints(self, node: TR181Node) -> ValidationResult:
        result = ValidationResult()

        # Custom validation for WiFi channels
        if "WiFi.Radio" in node.path and "Channel" in node.name:
            if node.value and node.value not in [1, 6, 11]:
                result.add_warning(f"WiFi channel {node.value} may cause interference")

        # Custom validation for string lengths
        if node.data_type == "string" and node.value:
            if len(node.value) > 64:
                result.add_error(f"String value too long: {len(node.value)} > 64")

        return result

# Use custom validator
validator = CustomValidator()
validation_result = validator.validate_node(node)
```

Custom Device Hooks

```
from tr181_comparator import DeviceConnectionHook, DeviceConfig

class SNMPHook(DeviceConnectionHook):
    def __init__(self):
        self.session = None

    async def connect(self, config: DeviceConfig) -> bool:
        # Implement SNMP connection
        from pysnmp.hlapi import *

        self.session = SnmpEngine()
        # Configure SNMP session...
        return True
```

```
    async def get_parameter_names(self, path_prefix: str = "Device.") -> List[str]:
        # Implement SNMP parameter discovery
        # Walk SNMP MIB tree and map to TR181 paths
        pass

    async def get_parameter_values(self, paths: List[str]) -> Dict[str, Any]:
        # Implement SNMP GET operations
        pass

# Register custom hook
from tr181_comparator import DeviceHookFactory
DeviceHookFactory.register_hook('snmp', SNMPHook)

# Use custom hook
device_config = DeviceConfig(
    name="snmp_device",
    type="snmp",
    endpoint="192.168.1.1",
    authentication={
        'community': 'public',
        'version': '2c'
    }
)
```

Batch Processing

```
async def batch_device_comparison():
    devices = [
        'http://device1.local/api',
        'http://device2.local/api',
        'http://device3.local/api'
    ]

    subset_manager = SubsetManager('reference_spec.json')
    reference_nodes = await subset_manager.extract()

    results = []

    for device_url in devices:
        try:
            device_config = DeviceConfig(
                name=f"device_{device_url.split('//')[1].split('.')[0]}",
                type="rest",
                endpoint=device_url
            )

            hook = RESTAPIHook()
            extractor = HookBasedDeviceExtractor(device_config, hook)
            device_nodes = await extractor.extract()

            engine = EnhancedComparisonEngine()
            result = await engine.compare_with_validation(
```

```
                reference_nodes,
                device_nodes,
                extractor
            )

            results.append({
                'device': device_url,
                'summary': result.get_summary(),
                'timestamp': datetime.now()
            })

        except Exception as e:
            print(f"Failed to process {device_url}: {e}")

    # Generate batch report
    print("Batch Comparison Report")
    print("=" * 30)
    for result in results:
        summary = result['summary']
        print(f"\nDevice: {result['device']}")
        print(f"  Validation errors:{summary['validation']['nodes_with_errors']}")
        print(f"  Missing nodes:{summary['basic_comparison']['missing_in_device']}")
        print(f"  Extra nodes:{summary['basic_comparison']['extra_in_device']}")

asyncio.run(batch_device_comparison())
```

Best Practices

1. Connection Management

- Always use connection timeouts to prevent hanging operations
- Implement retry logic for unreliable network connections
- Close connections properly to avoid resource leaks

```
async def robust_extraction():
    extractor = None
    try:
        config = {
            'endpoint': 'http://device.local:7547/cwmp',
            'timeout': 30,
            'retry_count': 3
        }
        extractor = CWMPExtractor(config)

        # Validate connection before extraction
        if await extractor.validate():
            nodes = await extractor.extract()
            return nodes
        else:
            raise ConnectionError("Device validation failed")

    except Exception as e:
```

```
            print(f"Extraction failed: {e}")
            return []
        finally:
            if extractor and hasattr(extractor, 'disconnect'):
                await extractor.disconnect()
```

2. Error Handling

- Handle different types of errors appropriately
- Provide meaningful error messages to users
- Log errors for debugging purposes

```
from tr181_comparator.errors import ConnectionError, ValidationError

async def safe_comparison():
    try:
        # Perform comparison operations
        result = await engine.compare(source1, source2)
        return result

    except ConnectionError as e:
        print(f"Connection failed: {e}")
        print("Check device connectivity and credentials")

    except ValidationError as e:
        print(f"Data validation failed: {e}")
        print("Check data format and TR181 compliance")

    except Exception as e:
        print(f"Unexpected error: {e}")
        print("Check logs for detailed error information")
```

3. Performance Optimization

- Use batch operations when possible
- Implement caching for frequently accessed data
- Monitor memory usage with large datasets

```
async def optimized_large_comparison():
    # Process in batches to manage memory
    batch_size = 1000
    all_results = []

    for i in range(0, len(large_node_list), batch_size):
        batch = large_node_list[i:i + batch_size]
        batch_result = await engine.compare(batch, reference_nodes)
        all_results.append(batch_result)

        # Optional: Clear cache between batches
        if hasattr(engine, 'clear_cache'):
            engine.clear_cache()
```

```
    # Combine results
    combined_result = combine_comparison_results(all_results)
    return combined_result
```

4. Data Validation

- Always validate input data before processing
- Use appropriate validation levels based on use case
- Document validation rules and constraints

```
async def validated_subset_creation():
    nodes = []

    # Validate each node before adding
    for node_data in input_data:
        node = TR181Node(**node_data)

        # Validate node definition
        validator = TR181Validator()
        validation_result = validator.validate_node(node)

        if validation_result.is_valid:
            nodes.append(node)
        else:
            print(f"Invalid node {node.path}:")
            for error in validation_result.errors:
                print(f"  {error}")

    # Save validated subset
    if nodes:
        subset_manager = SubsetManager('validated_subset.json')
        await subset_manager.save_subset(nodes)
```

5. Reporting and Documentation

- Include metadata in reports for traceability
- Use appropriate output formats for different audiences
- Document comparison criteria and validation rules

```
async def comprehensive_reporting():
    result = await engine.compare_with_validation(subset_nodes, device_nodes)

    # Generate different report formats for different audiences

    # Technical report (JSON)
    technical_report = {
        'metadata': {
            'timestamp': datetime.now().isoformat(),
            'subset_source': subset_manager.get_source_info(),
            'device_source': device_extractor.get_source_info(),
            'comparison_engine': 'EnhancedComparisonEngine v1.0'
        },
```

```
            'summary': result.get_summary(),
            'detailed_results': {
                'missing_nodes': [node.path for node in result.basic_comparison.only_in_s
                'extra_nodes': [node.path for node in result.basic_comparison.only_in_sou
                'validation_errors': [(path, res.errors) for path, res in result.validati
            }
        }

        # Executive summary (text)
        executive_summary = f"""
        TR181 Compliance Report
        ======================

        Device: {device_extractor.get_source_info().identifier}
        Specification: {subset_manager.get_source_info().identifier}
        Date: {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}

        Summary:
        - Total parameters validated: {len(subset_nodes)}
        - Compliance issues found: {len([r for _, r in result.validation_results if not r
        - Missing implementations: {len(result.basic_comparison.only_in_source1)}
        - Extra implementations: {len(result.basic_comparison.only_in_source2)}

        Recommendation: {'PASS' if len([r for _, r in result.validation_results if not r.
        """

        # Save reports
        with open('technical_report.json', 'w') as f:
            json.dump(technical_report, f, indent=2)

        with open('executive_summary.txt', 'w') as f:
            f.write(executive_summary)

asyncio.run(comprehensive_reporting())
```