JANOG58

CLIパーサーからの脱却

モデル駆動型プラットフォームによるマルチベンダー検証の完全自動化

KDDI株式会社
荻原 拓也、横山 直

Spark Your Journey
KDDI

今日のお話

KDDIでは膨大なIPネットワークの検証業務にモデルベースの自動化を導入しました
今回はその過程で生じた課題と取り組み、得られた成果や知見を共有します

これまでの検証業務

- 手順作成
- セットアップ
- 検証
- 結果判定

×1000~

膨大な件数の検証を外注で対応
→ **多大なコスト/期間が発生**

前回の発表(JANOG54)

- コード開発
- セットアップ
- 検証
- 結果判定

数件を検証

OpenConfigベースのテストフレームワークを用いてPoCを実施
→ 有用性を確認し、**導入を決定**

今回の内容

- コード開発
- セットアップ
- 検証
- 結果判定

500件超を開発/導入

500件超のシナリオを開発、業務に投入
→ **開発大規模化で課題発生**
→ **アーキテクチャを再検討**
→ **検証内製化、期間は1/10以下に**

検証業務と自動化の方針

KDDIのネットワークと検証業務の規模

KDDIのIPネットワークは数千万の加入者を収容する重要なサービス基盤です
その品質担保のため、検証業務は開発において大きなウェイトを占めています

サービスとネットワーク

サービス

- **収容サービス**: KDDIグループの通信サービス全般
  (au, auひかり, UQ mobile, BIGLOBE,,,)
- **加入者数**: XX M Subs

ネットワーク

- **トラフィック量**: XX Tbps
- **機器数**: 1000~
- **機種数**: 10~

検証要件と業務規模

検証規模

- **基本機能検証**: **1500件~**
- **複合/IOT(※)検証**: 200件~
- **検証期間**: **3~6ヵ月**

※ Inter-Operability Testing: 相互接続性試験

トリガー

- **沢山**
  (バージョンアップ, 更改, 構成変更,,,)

手動での検証はやってられない😇

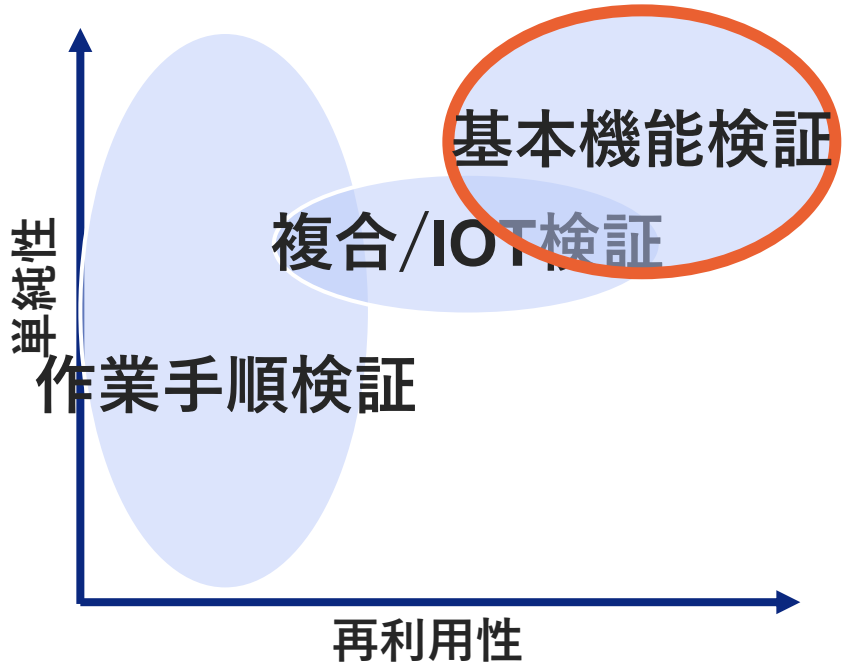
検証業務の分類と自動化対象

KDDIのIPネットワークにおける検証業務は大きく3つに分類できます
このうち、最も再利用性、単純性の高い基本機能検証を対象に自動化をしました

検証業務の分類

分類	位置づけと実施内容例
• **基本機能検証:**	機器単体の各種機能が期待通りに動作するか。 例: ネイバー確立, タイマー動作, MRAI動作,,,
• **複合/IOT検証:**	特定の機器間の接続性における各種機能や障害時の挙動が期待通りに動作するか。 例: 障害時の切替,トラフィック転送性能,,,
• **作業手順検証:**	商用機器に対する各種設定操作が期待通りに動作するか。 例: リンク増速, トラフィックシェーピング,,,

検証シナリオの試行回数と複雑性



基本機能検証の自動化が最もコスパがよいと判断 😎

自動化フレームワークの選定

KDDIの要件を踏まえ、OpenConfigベースのテストフレームワーク ONDATRAを候補に選定しました

KDDIの要件

モデルベース/マルチベンダー対応

- 多種のデバイスを統一的に検証したい

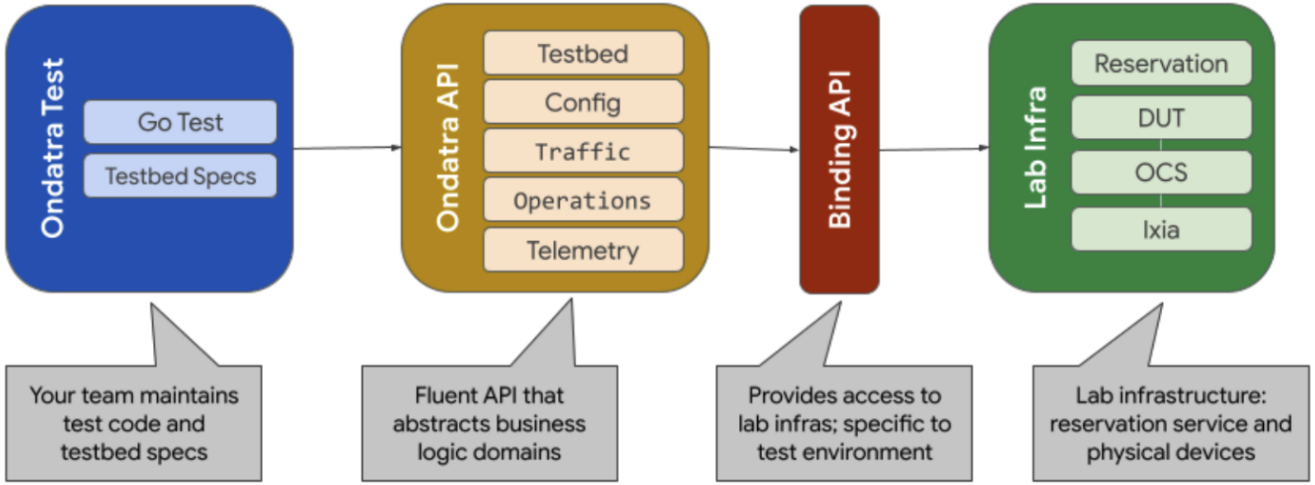
テスト環境の抽象化

- 限られたリソースの中で、仮想を含めこれらを組み合わせ透過的に検証したい

オープンな実装

- 自分達で理解し、拡張できるツールを使いたい

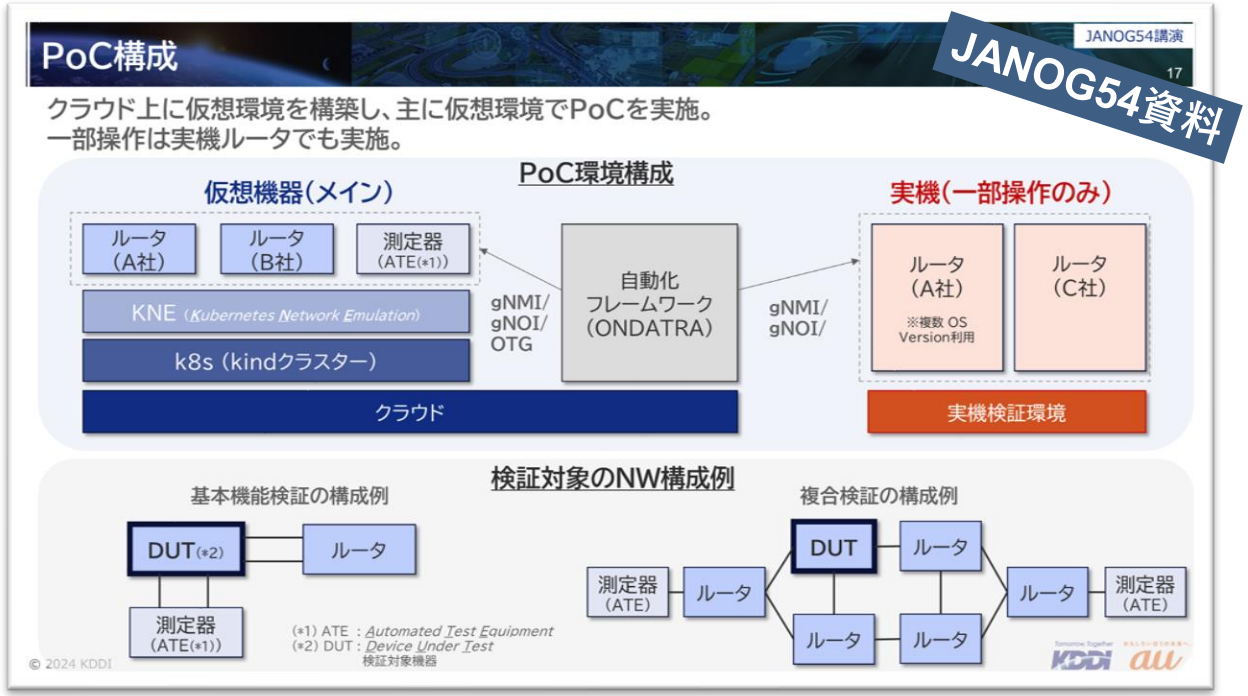
ONDATRAの概要

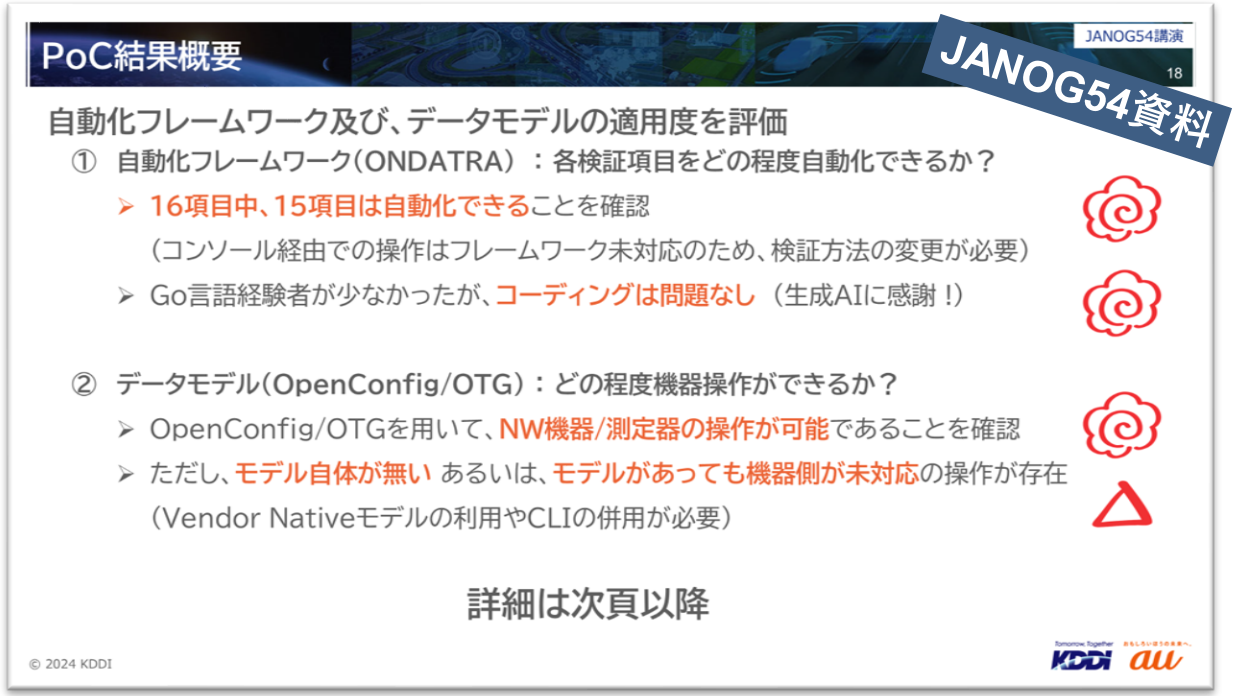


- Google製のネットワークテストフレームワーク
- 汎用モデルである**OpenConfig/OTG APIをネイティブサポート**
- bindingにより**実環境とテスト上の論理トポロジーを分離可能**
- **OSSとして公開**されている

ONDATRA による自動化の検証

ONDATRAを用いて実際のシナリオを検証し、その有用性を確認しました





- 仮想/物理を含めた**テストベッドの抽象化が容易**なことを確認
- **測定器の制御も可能**であることを確認

- **OpenConfigによるマルチベンダ対応**を確認
- OpenConfig非対応部分も代替手法があることを確認

ONDATRAの導入を決定 👌

大規模化で見えた課題

拙速な大規模展開による副作用

カバレッジを優先してアーキテクチャや保守性、利用性を考慮せず進めた結果、
技術的負債が課題として健在化し、テストコードの有用性が損なわれていきました

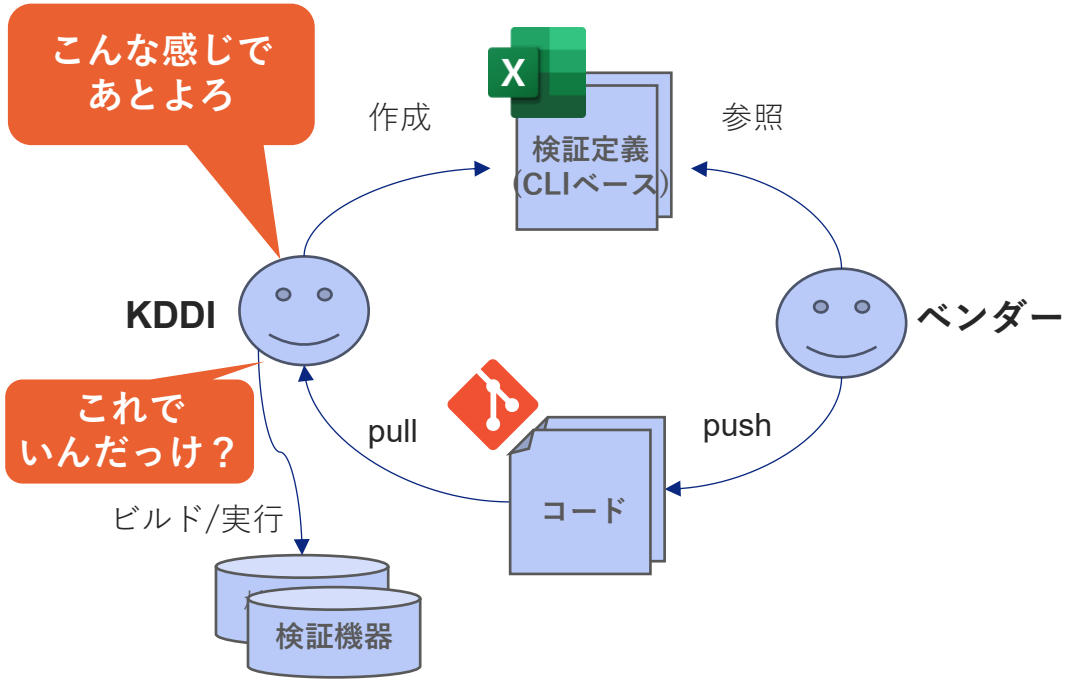
当初の実装	発生した課題
ベンダー間の差異の吸収 • シナリオ内で雑にswitch/if分岐	**課題1: マルチベンダー対応によるシナリオの肥大化** • シナリオにデバイス依存の実装が染み出し • **変更/改修負荷が増大、シナリオの可読性も消滅**
OpenConfig非互換な操作 • CLIパーサ + gomiko(Netmiko的なもの)でゴリ押し	**課題2: CLIパーサーのメンテナンス問題** • **難解な正規表現で実装の妥当性評価/検証が困難**に • VerUp等に伴う出力変化の影響範囲が特定できない
テストの実行方法 • go test で手動実行	**課題3: テスト実行のしにくさと属人化** • **職人のシェル芸に依存した再現性の低いテスト**に

拙速な開発で技術的負債が蓄積、テストコードの価値を棄損 😭

開発スキームの問題

当初はCLIベースの既存手順をもとにコード開発をすべて業務委託していました
結果、KDDIは実装の詳細を把握せず、技術的負債の蓄積を助長しました

当初の体制



起きたこと

- **デザイン/保守性の要件なし**
  - → コメントのない難読な正規表現
  - →スピード重視の実装/再利用性の低いコード
- **中身を見ない**
  - → 雑なsleep待機や甘い判定でたまにこける
  - → 本当にすべき検証になっているのか不明
  - → 自分達で直せない

業務委託に頼り実装の詳細を把握しなかったことで課題を助長😨

課題1：マルチベンダー対応によるシナリオ肥大化

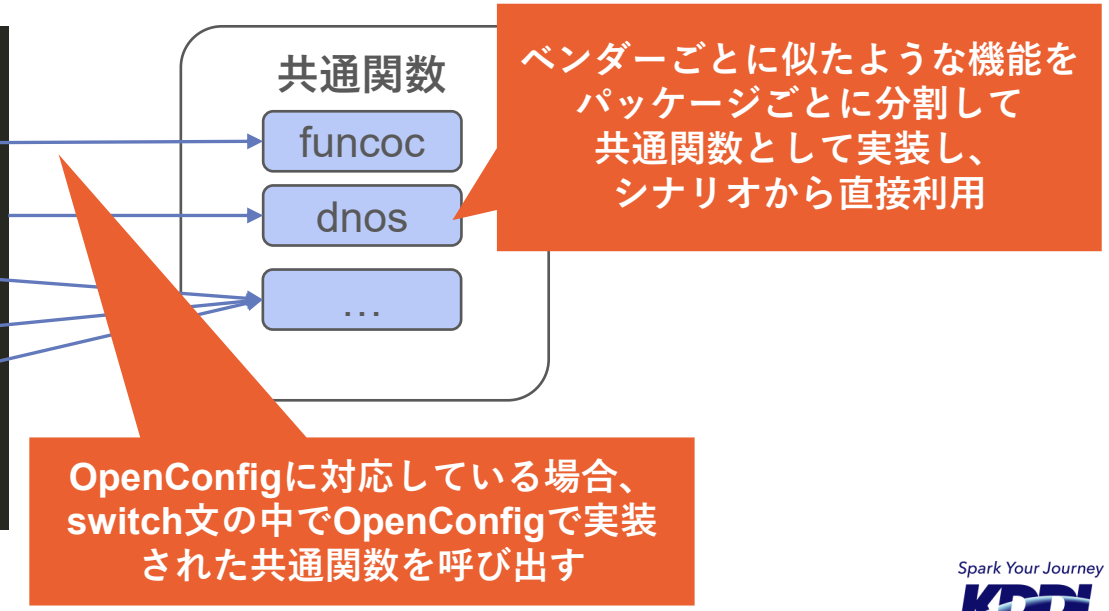
**OpenConfigでカバーしきれない操作に対し、
シナリオコード内に直接デバイスごとのif/switch分岐を記述。**

何が起きたか？

- ベンダーやOSが追加されるたびに、全テストシナリオに分岐を追加する羽目に。
- 「何を設定して、何をテストしているのか」という本来のビジネスロジックが、ベンダー固有の泥臭い処理に埋もれてしまい、コードの可読性が崩壊。
- ベンダーごとの実装をシナリオに記述したため、改修コストが激増。

```
vendorNameDut := ondatra.DUT(t, parameter.Dut.Name).Vendor().String()
switch vendorNameDut {
case "CISCO":
  funcoc.ChkIfMtu(t, parameter.Dut.Name, parameter.Dut.Port1Name, defaultMtu)
case "DRIVENETS":
  dnos.ChkIfIPv4Mtu(t, parameter.Dut.Name, parameter.Dut.Port1Name, defaultMtu-ethHeaderSize)
case "JUNIPER":
  junos.ChkIfLogicalInterfaceMtu(t, parameter.Dut.Name, parameter.Dut.Port1Name, defaultMtu)
case "ARISTA":
  eos.ChkIfEthernetMtu(t, parameter.Dut.Name, parameter.Dut.Port1Name, defaultMtu)
case "NOKIA":
  sros.ChkIfPortMtu(t, parameter.Dut.Name, parameter.Dut.Port1Name, defaultMtu)
default:
  t.Fatalf("Unimplemented vendor: %s", vendorNameDut)
}
```

ベンダーが追加されるたびに肥大化する巨大なswitch文の例
＊わかりやすくするためにコードを簡略化

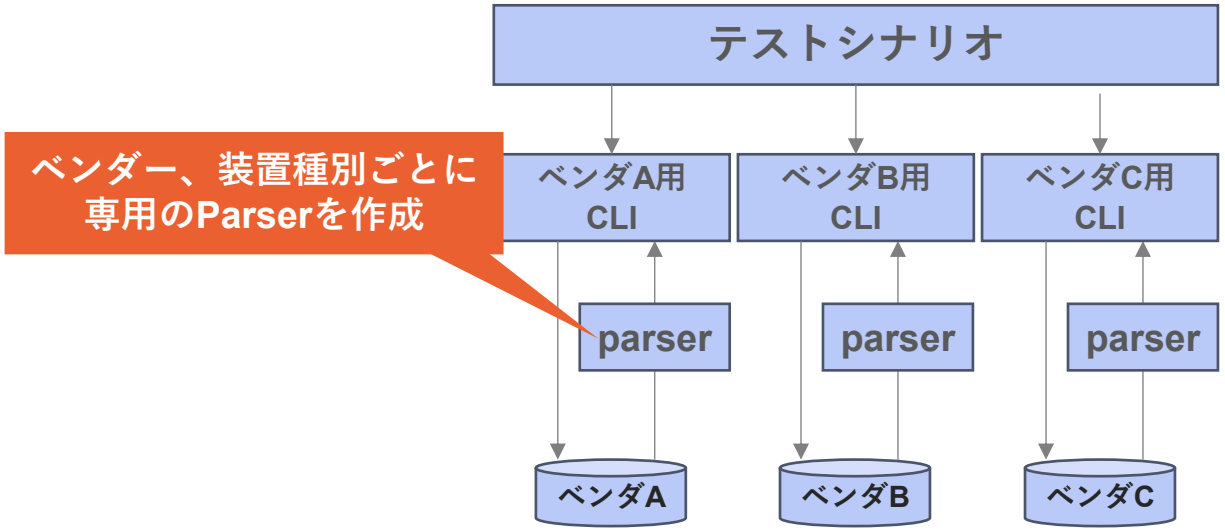


課題2：CLIパーサーのメンテナンス問題

OpenConfig非互換な操作をCLIパーサーとコマンド投入処理で実装。

何が起きたか？

- 同じOSでもプラットフォームや状態によって出力は変わり、パースエラー/意図しないPASSが発生
- 構造化されていないテキストデータ処理→実装者以外理解不能で複雑な正規表現が大量に発生
- 実機確認なしにコードの動作や出力詳細が分からず、デバッグや開発のボトルネックに




```
// Get BGP Neighbor Last Established
output, _ := d.SendCommand("show bgp summary")
// 正規表現のフォーマット文字列を変数化
pattern := `(?m)^\s*%s\s+\d+\s+\d+\s+\d+\s+\d+\s+\d+\s+\d+\s+\d+\s+\d+\s+(\d+):(\d+):(\d+)`
re := regexp.MustCompile(fmt.Sprintf(pattern, regexp.QuoteMeta(neighborIP)))
match := re.FindStringSubmatch(output)
```

＊複雑な正規表現のイメージ

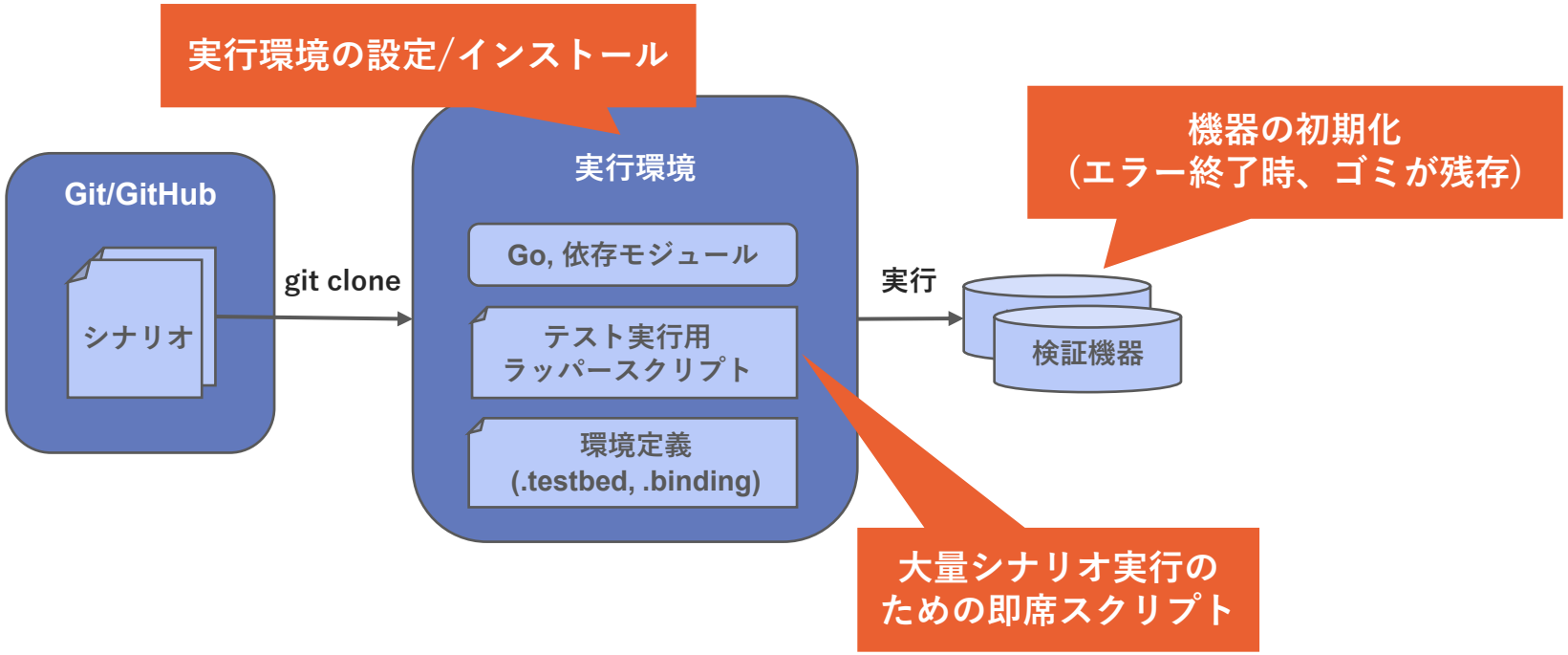
CLI+パーサによる自動化はスケール、品質、保守全てで限界がある💀
ソフトウェア制御に適したモデル/APIベースの自動化に移行すべき🤔

課題3：実行のしにくさと属人化

自動化スクリプトを動かすために、複雑なローカル環境の構築が必要。

何が起きたか？

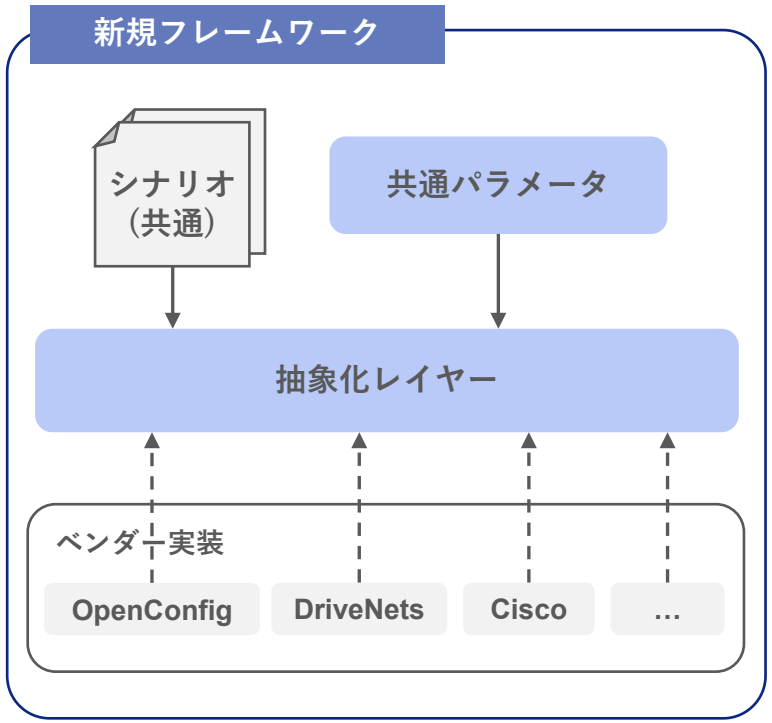
- 自動化環境は作成者しか操作できず、ログの確認方法も限定的で属人化が進行。
- 初期化処理がシナリオ内にあるため、途中でエラーが起きると機器に不要な設定が残り、後続テストが失敗する問題が発生（冪等性の欠如）。
- 再現性がなく、実行できる人が限られているためボトルネックが発生。



課題への対応: 再設計とプラットフォーム化

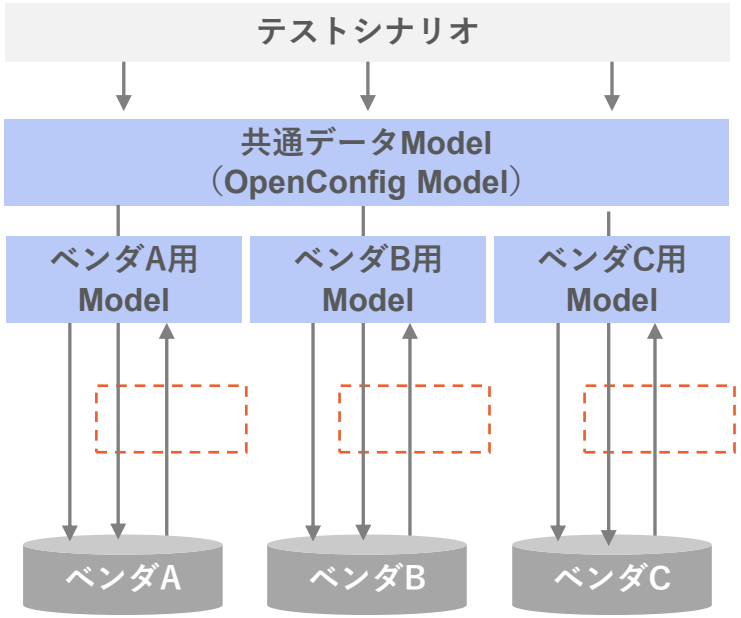
課題に対する全体方針

1. シナリオの抽象化



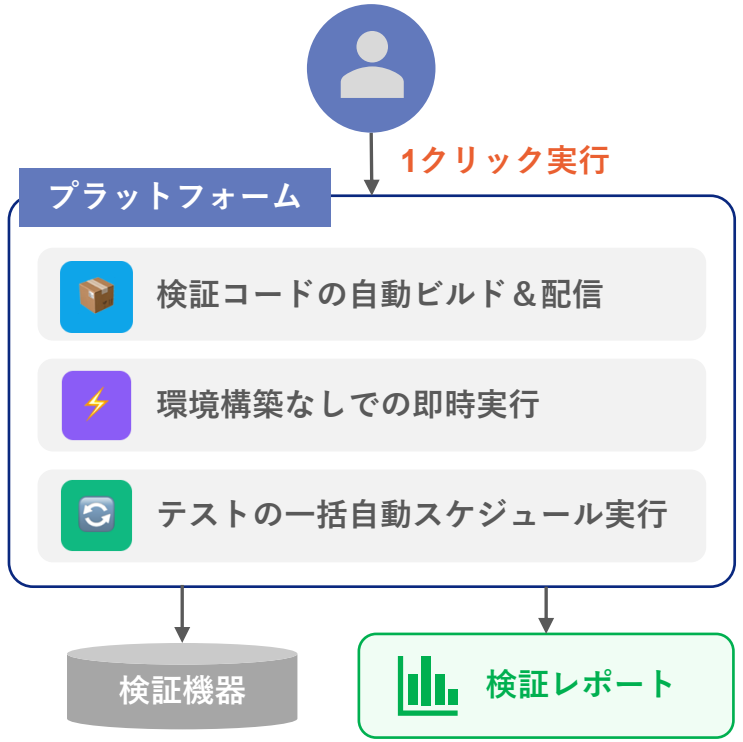
- 規約による構造の明確化
- ベンダー非依存のシナリオ設計
- トポロジー変数の外部化

2. CLIパーサの削減



- YANGモデル駆動開発への転換
- ygot/ygnmiによるコード自動生成
- 型安全かつ可読性の高い実装

3. ユーザビリティの向上



- Buildと環境構築のポータブル化
- CI連携による検証のワンクリック化
- 検証結果、レポートの自動生成

詳細は当日共有
デモを予定

議論

議論内容

? NW検証の実施項目について

- **どのようなプロセスや考え方で検証業務を定義し実施していますか？**
- **検証業務の組織間での共通化や外部化についてどう考えますか？**
  - Google の Feature Profilesや、検証コードの機器ベンダーとの共有など

? NW自動化の組織的アプローチについて

- **自動化における3つのロール（要件定義、開発/保守、利用/実行）を チームとしてどう分業・連携させていますか？**
  - 「分業によるコミュニケーションやスキルギャップの壁」を乗り越える工夫はありますか？
- **自動化人材の育成において、「NWエンジニアへのSW教育」と「SWエンジニアへのNW教育」のどちらが近道でしょうか？**
  - 生成AIやコーディングエージェントの普及は、どちらの学習コストを引き下げていると感じますか？

? NW自動化のソフトウェアアプローチについて

- **CLIパーサーベースの自動化/コードベースでどのような課題を持っていますか？**
- **マルチベンダー検証の自動化において、ベンダー間の差異をどのように管理していますか？**
- **モデルベースのAPIやインターフェースを持つネットワーク機器が増えてきている中で、どこまでこれを実運用し始めていますか？**

Spark Your Journey

KDDI

「つなぐチカラ」を進化させ、
誰もが思いを実現できる社会をつくる。

KDDI VISION 2030

参考資料

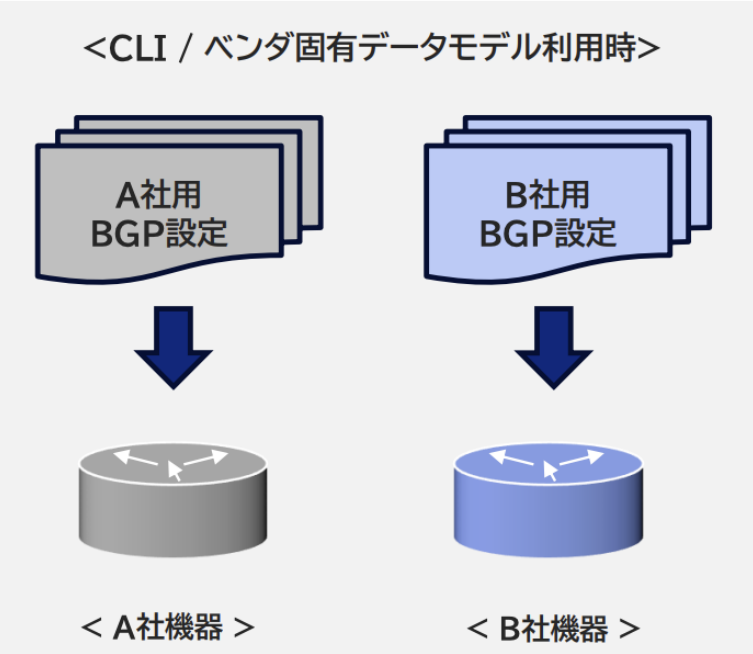
OpenConfigとは？

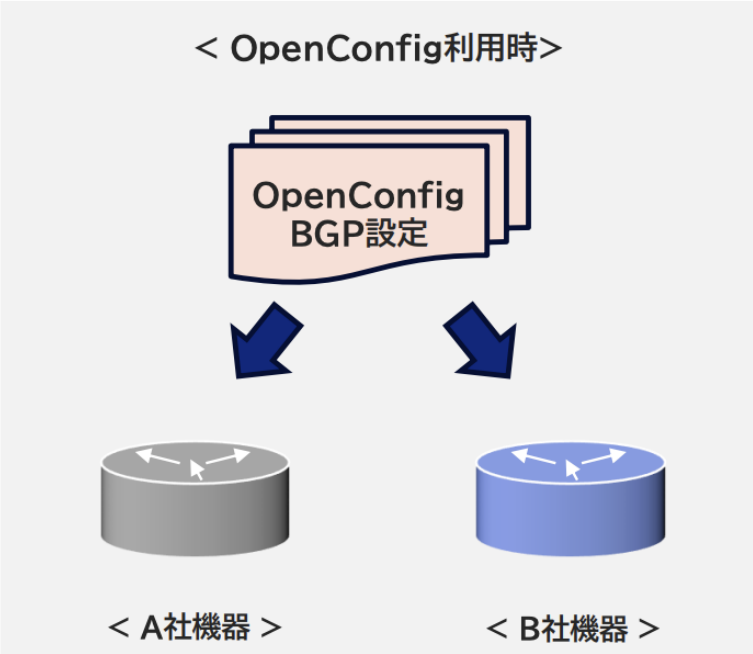
JANOG54資料

- 各ベンダ装置を一貫した手法で管理するための共通データモデル
- オペレーター主導で、WGにて共通データモデルを定義。既に各ベンダ機器で実装も進んでいる。

共通データモデル(OpenConfig)

従来は、機器/OSごとに異なるコマンドを利用したオペレーションが必要であったが、
OpenConfigにより、共通のデータモデルに基づいて**統一したオペレーションが可能**となる





ベンダ実装状況

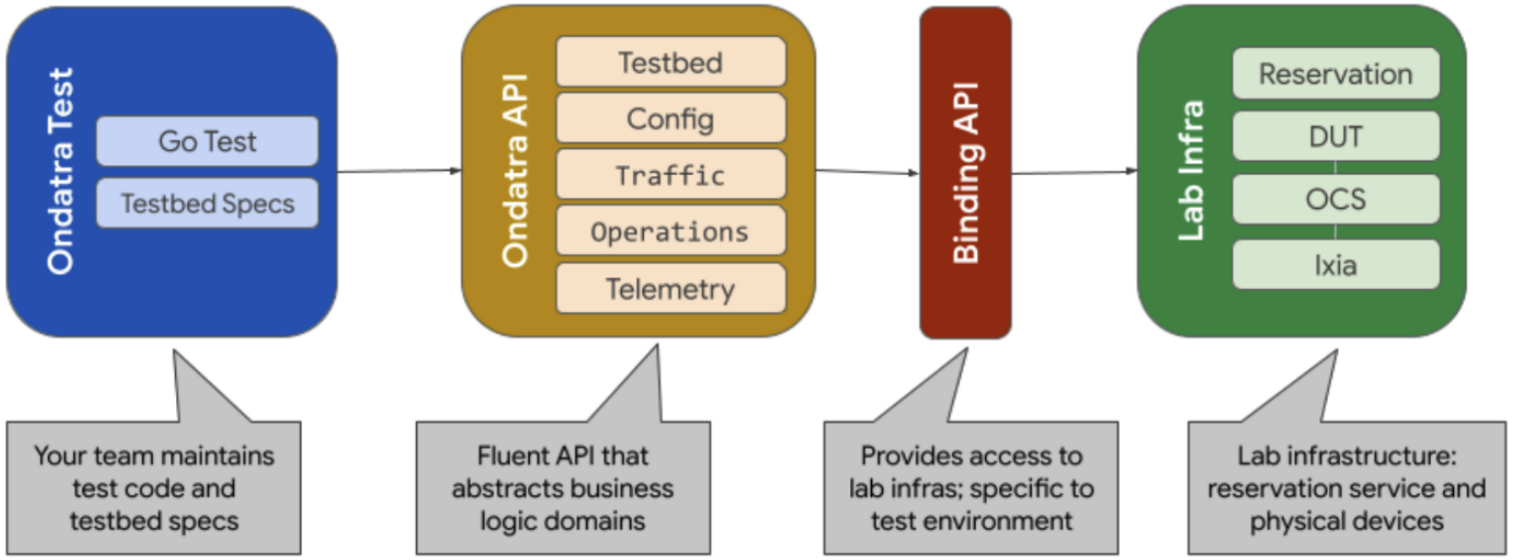
ルーティング／スイッチング/光トランスポート等の
領域において、**主要ベンダの多くのプラットフォームで実装**されている

< OpenConfig対応ベンダ(一部)>

Cisco　Juniper
Nokia　Arista
SONiC　DELL
Ciena

ONDATRA (Open Network Device Automated Test Runner & API)

・OpenConfigの利用を前提とした検証自動化フレームワーク
・主にGoogleが作成・公開しているOSS。Go言語ベースで、go testとして検証コードを実装
・NW機器、トラフィックジェネレータに対する操作を容易にするAPIを提供
・Testbedで定義したhost名やinterface名を利用するため、ベンダーや検証環境に依存しないコードが作成可能



https://docs.google.com/viewer?url=https://raw.githubusercontent.com/openconfig/ondatra/main/internal/tour/tour.pdf

ONDATRA の特徴

OpenConfigとTestbedによってベンダーと検証環境に依存しない共通コードが実装可能

```
dut := ondatra.DUT(t, "dut")
port := dut.Port(t, "port1")

i := &oc.Interface{}
i.Description = ygot.String("Sample")
i.Enabled = ygot.Bool(true)

gnmi.Update(t, d.dut, gnmi.OC().Interface(port.Name()).Config(), i)
```

コード例：I/F Description, Shutdownの設定

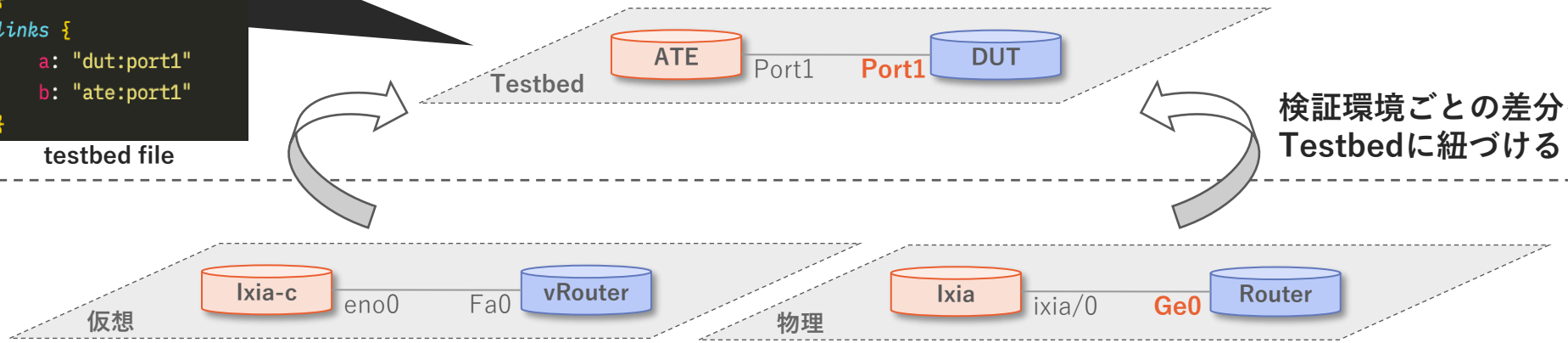
```
duts {
    id: "dut"
    ports {
        id: "port1"
    }
}
ates {
    id: "ate"
    ports {
        id: "port1"
    }
}
links {
    a: "dut:port1"
    b: "ate:port1"
}
```

testbed file

共通トポロジーを定義

Testbedをベースに
コードを実装

検証環境ごとの差分を
Testbedに紐づける



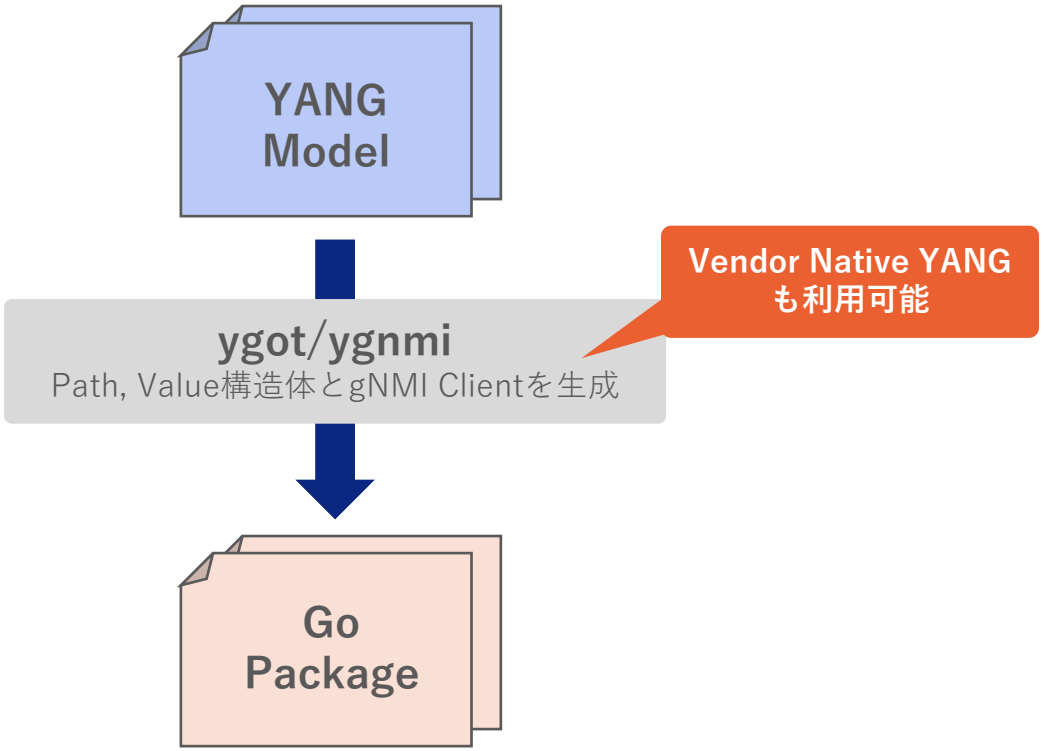

```
duts {
  id: "dut"
  name: "dut"
  vendor: DRIVENETS
  options {
    username: "dnroot"
    password: "dnroot"
    skip_verify: true
  }
  gnmi {
    target: "172.18.0.54:50051"
  }
  ssh {
    target: "172.18.0.54:22"
  }
  gnoi {
    target: "172.18.0.54:50051"
  }
  ports {
    id: "port1"
    name: "ge100-0/0/0"
  }
}
ates {
  id: "ate1"
  name: "ate1"
  vendor: IXIA
  otg {
    target: "172.18.0.51:40051"
    insecure: true
    timeout: 120
  }
  gnmi {
    target: "172.18.0.50:50051"
    skip_verify: true
    timeout: 30
  }
  ports {
    id: "port1"
    name: "eno0"
  }
}
```

binding
file

ygot/ygnmi

ネットワークの状態をGoのコードで直接操作することが可能。

YANGから自動生成されたGoのコードを利用するため、 ntc-templates等のCLIパーサーが不要。




```
dut := ondatra.DUT(t, "dut")
port := dut.Port(t, "port1")

i := &oc.Interface{}
i.Description = ygot.String("Sample")
i.Enabled = ygot.Bool(true)

gnmi.Update(t, d.dut, gnmi.OC().Interface(port.Name()).Config(), i)
```

ygotで生成した
Value構造体でパラメータを定義

コード例：I/F Description, Shutdownの設定

ygnmiで生成した
ClientとPath構造体を利用し、
設定投入と状態取得を実施

```
portName := d.dut.Port(t, portID).Name()

wantStatusOC := oc.Interface_OperStatus_UP
// 期待値になるまで待機
got := gnmi.Await(t, d.dut, gnmi.OC().Interface(portName).OperStatus().State(), 30*time.Second, wantStatusOC)

if status, _ := got.Val(); status == wantStatusOC {
    t.Logf("インターフェースの動作状態が期待通りです: %v (port:%s, dut:%s)", wantStatusOC, portID, d.dut.Name())
} else {
    t.Errorf("インターフェースの動作状態が期待と異なります: %v (port:%s, dut:%s)", wantStatusOC, portID, d.dut.Name())
}
```

コード例：I/F Statusの取得