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

自己紹介

荻原　拓也
Ogihara Takuya

横山　直
Yokoyama Nao

所属	IPネットワーク部	IPネットワーク部
経歴	◘ 某通信事業者 DCBBNWの設計・開発、自動化 ◘ 某物流系スタートアップ 物流自動倉庫向けNW開発 ◘ NW自動化開発(2025～ KDDI入社) IPネットワークの自動化	◘ 仮想化基盤開発（～2024） プライベートクラウド(MShip3)、 オンプレ仮想化基盤等のインフラ開発 ◘ NW自動化開発（2025～） IPネットワークの自動化
JANOG参加歴	JANOG43(山梨) 若者支援プログラム JANOG57(大阪) 一般参加	なし

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😇

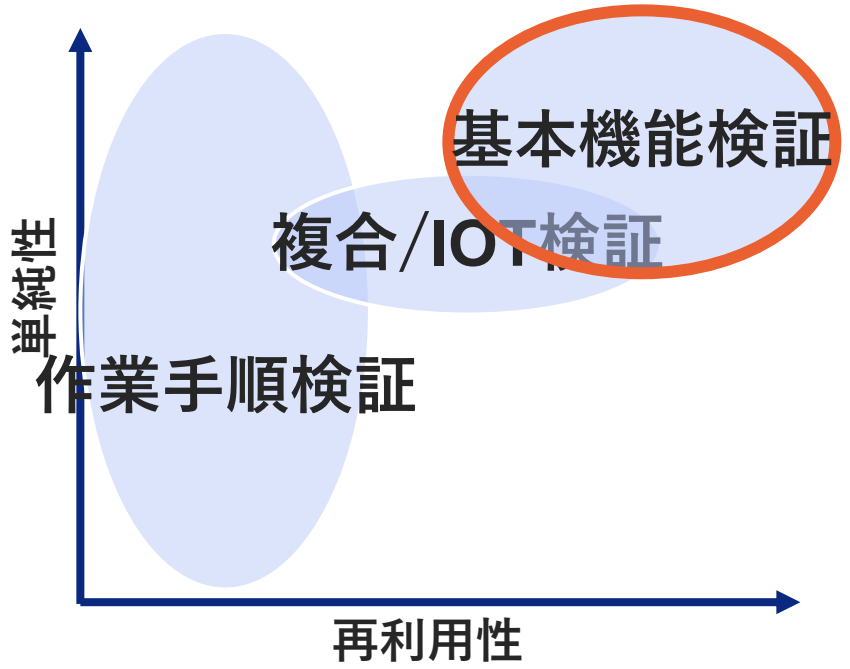
検証業務の分類と自動化対象

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

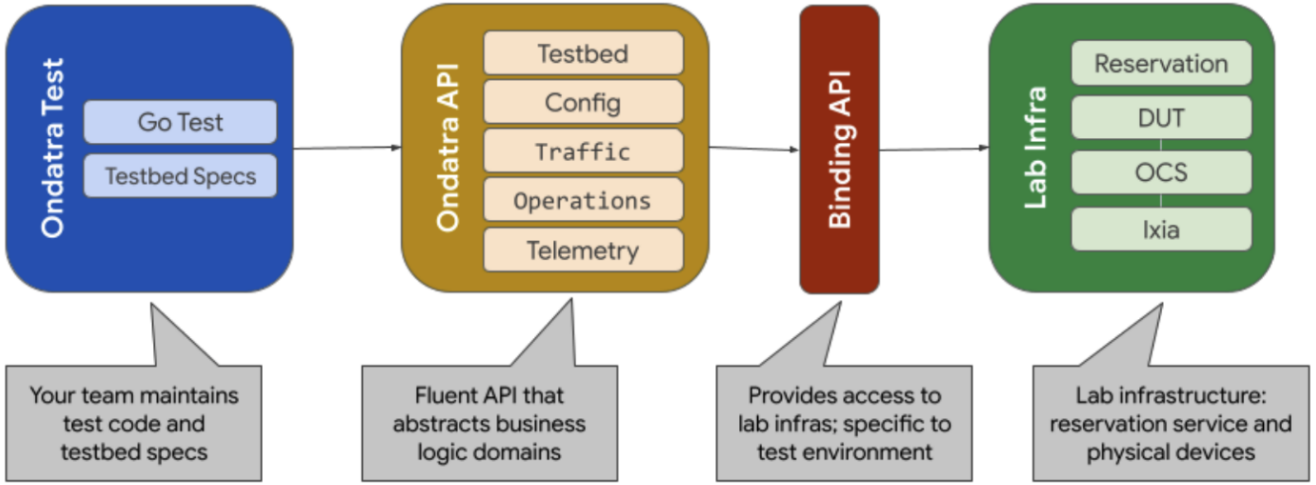
テスト環境の抽象化

- 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

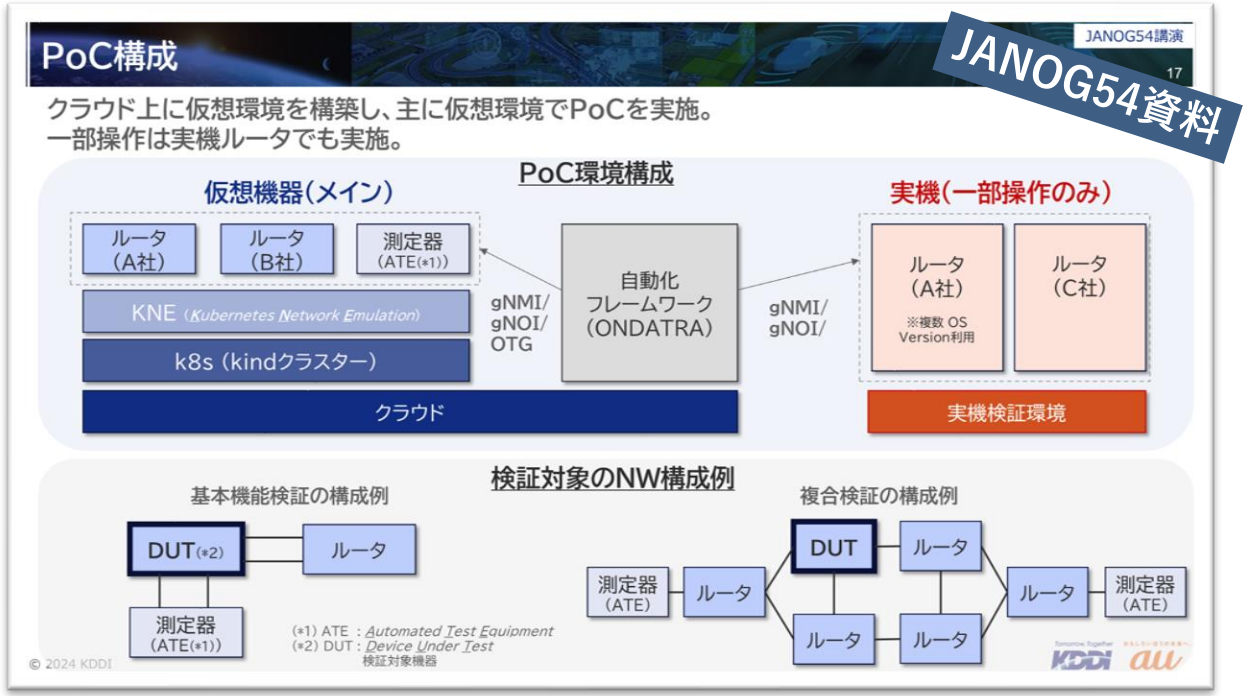
ONDATRAの概要

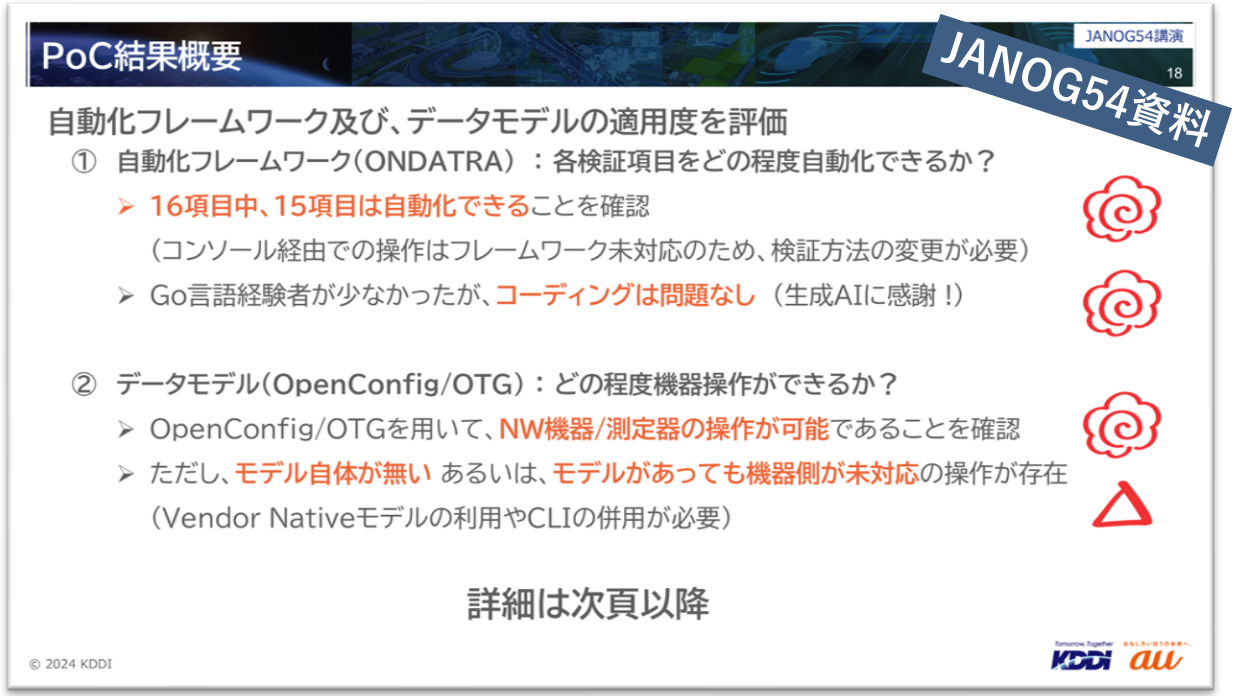


- Google製のネットワークテストフレームワーク
- 汎用モデルである**OpenConfig/OTG APIをネイティブサポート**
- bindingにより**実環境とテスト上の論理トポロジーを分離可能**
- **OSSとして公開**されている

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

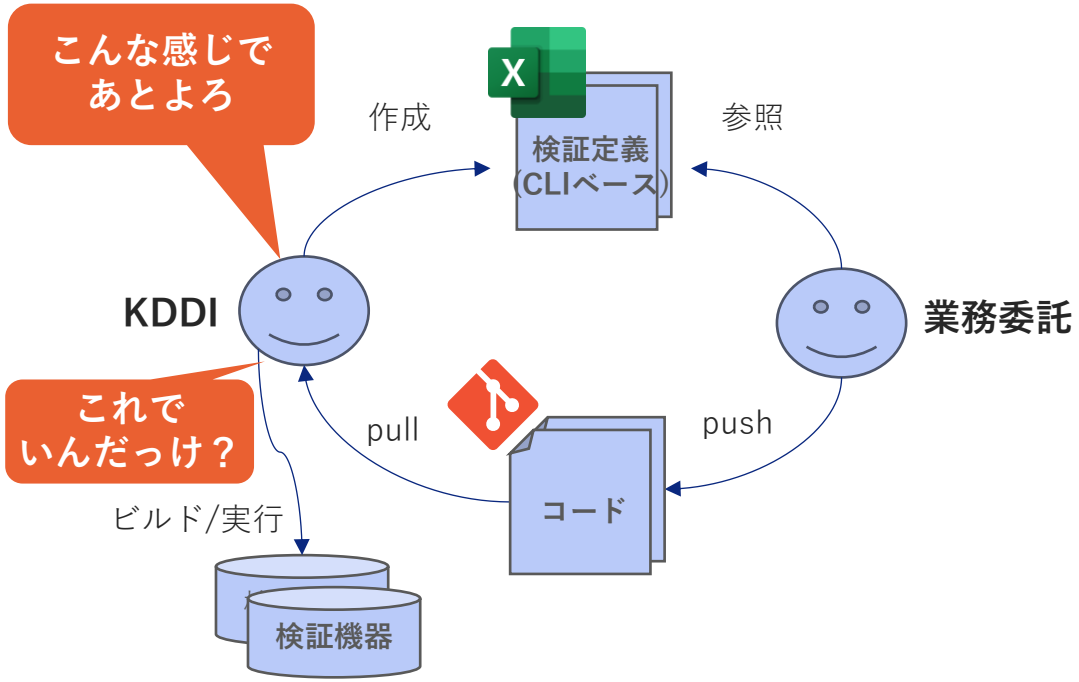
カバレッジを優先してアーキテクチャや保守性などを考慮せず開発を進めた結果、技術的負債が蓄積し、テストコードの有用性が損なわれていきました

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

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

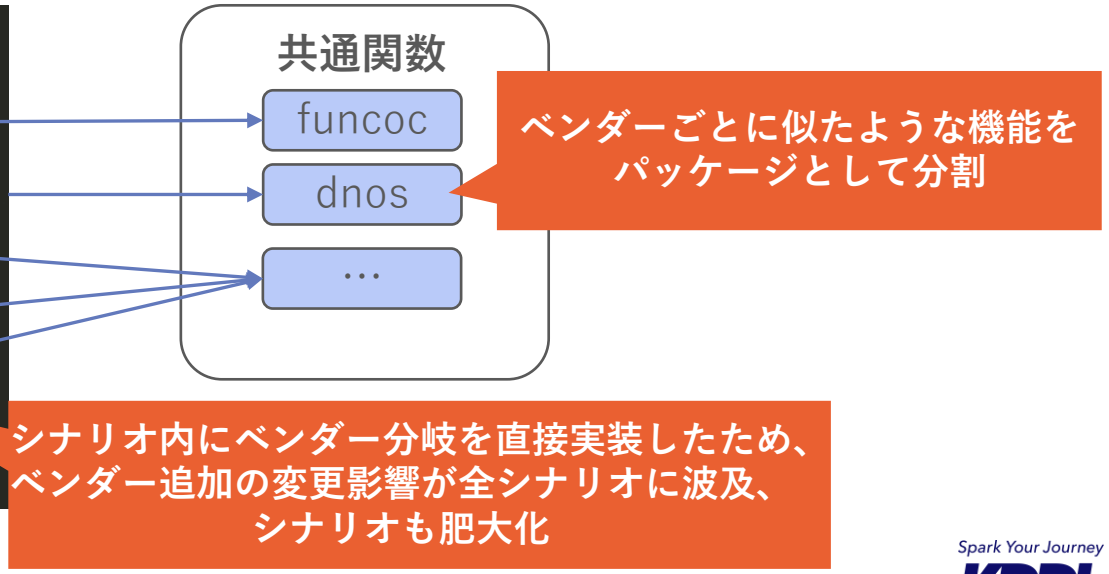
**OpenConfigでカバーしきれない操作に対し、
シナリオコード内に直接デバイスごとのif/switch分岐を記述。**

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、ベンダー実装の呼び出しと分岐処理に埋もれてしまい、コードの可読性が崩壊。
- 呼び出しているベンダー実装が本当に同じ操作をしているのかが把握できない。

```
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

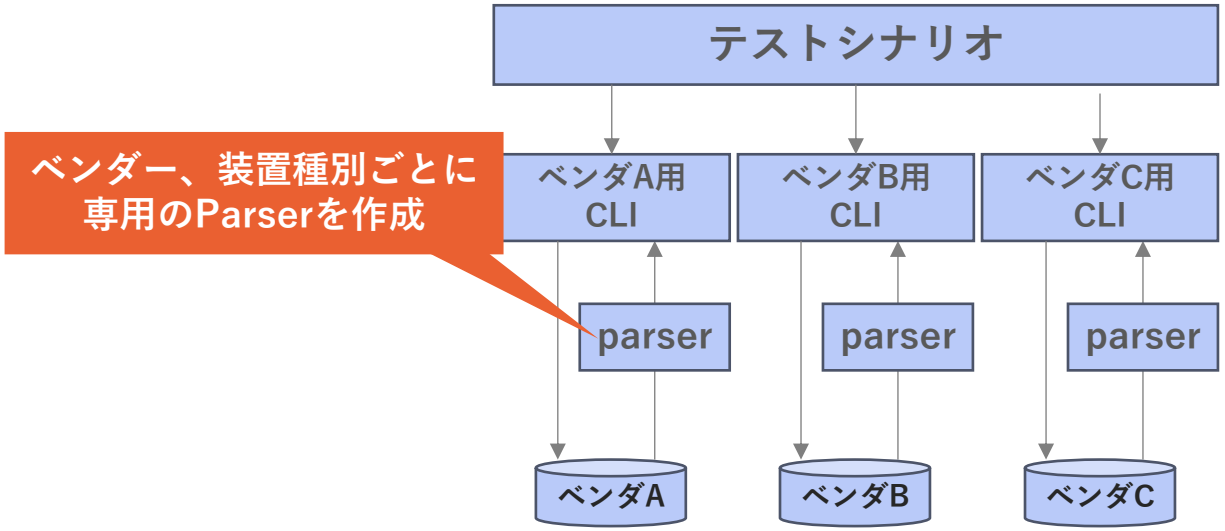


課題2：CLIパーサーのメンテナンス問題

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？

- プラットフォームや状態によって出力が変わるため、環境によって意図しない合格/不合格が発生。
- 構造化されていないテキストデータのパース処理→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

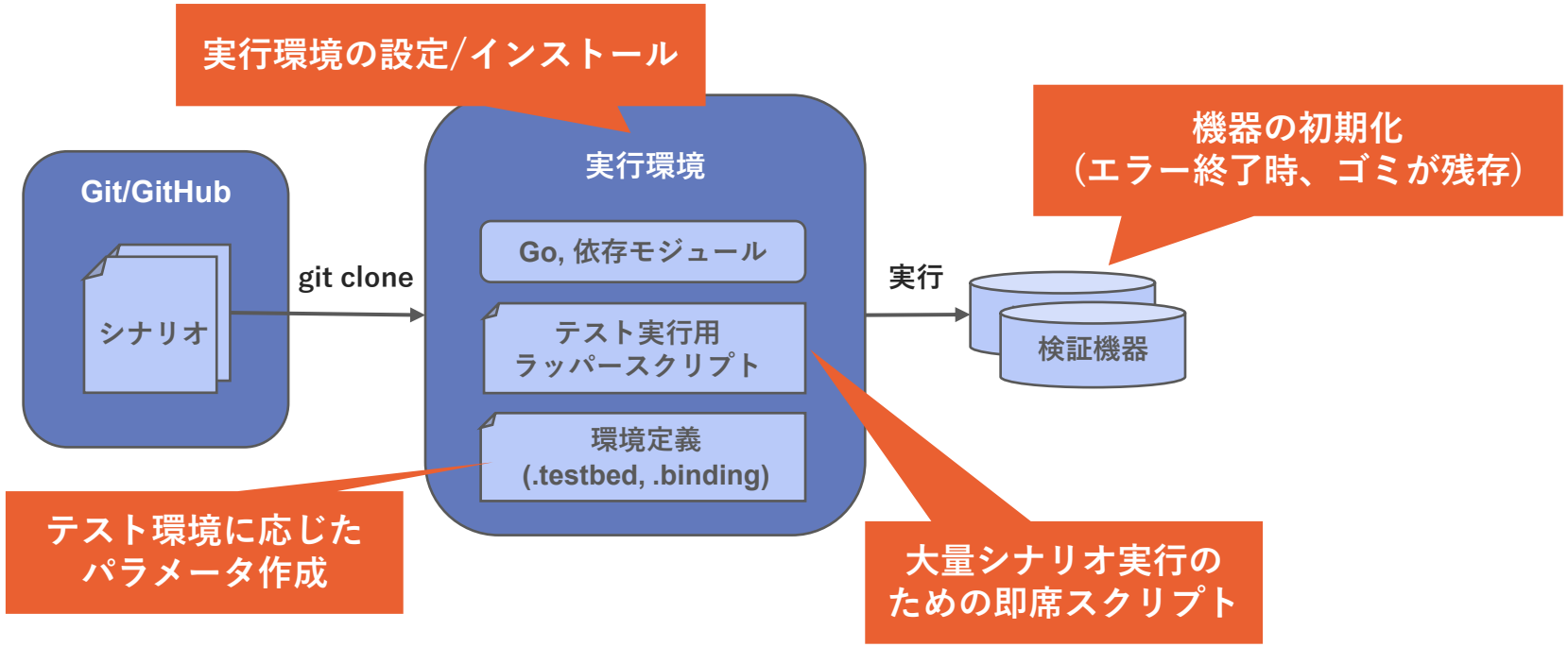
CLI+パーサによる自動化はスケール、品質、保守全てで限界があった💀
ソフトウェア制御に適したモデル/APIベースの自動化に移行すべき🤔

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？

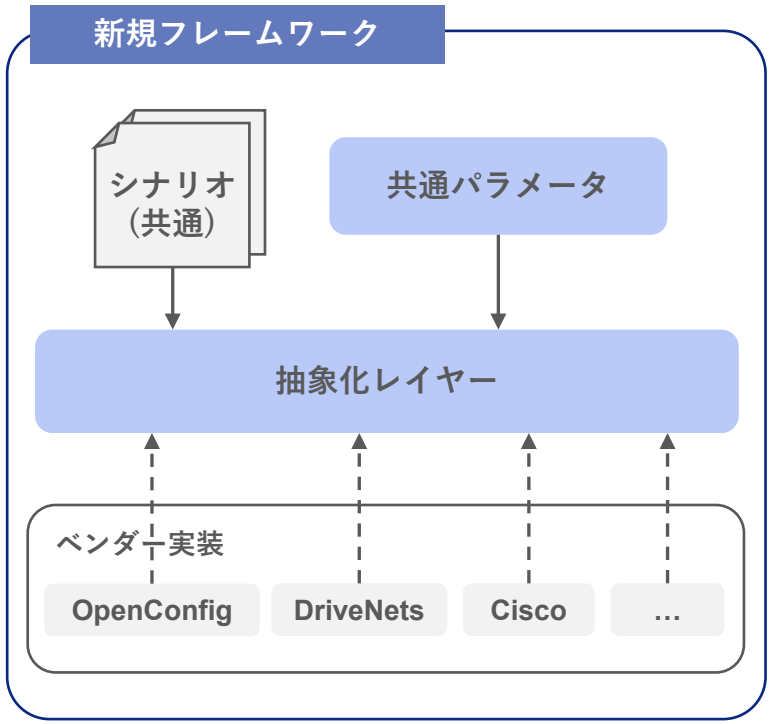
- 実行環境の構築や実行パラメータの整備に複雑な操作や設定が必要となり、属人化が進行。
- 初期化処理がシナリオ内にあるため、途中でエラーが起きると機器に不要な設定が残り、後続テストが失敗する問題が発生（冪等性の欠如）。
- テストの準備や実行ができる人間が限定され、検証のボトルネックに。



課題へのアプローチ: 再設計とプラットフォーム化

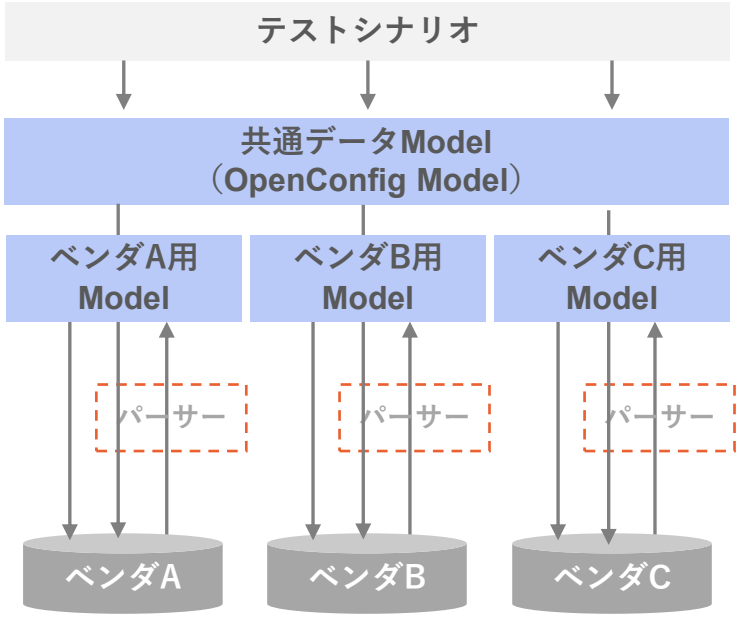
課題へのアプローチの全体像

課題1 シナリオの抽象化



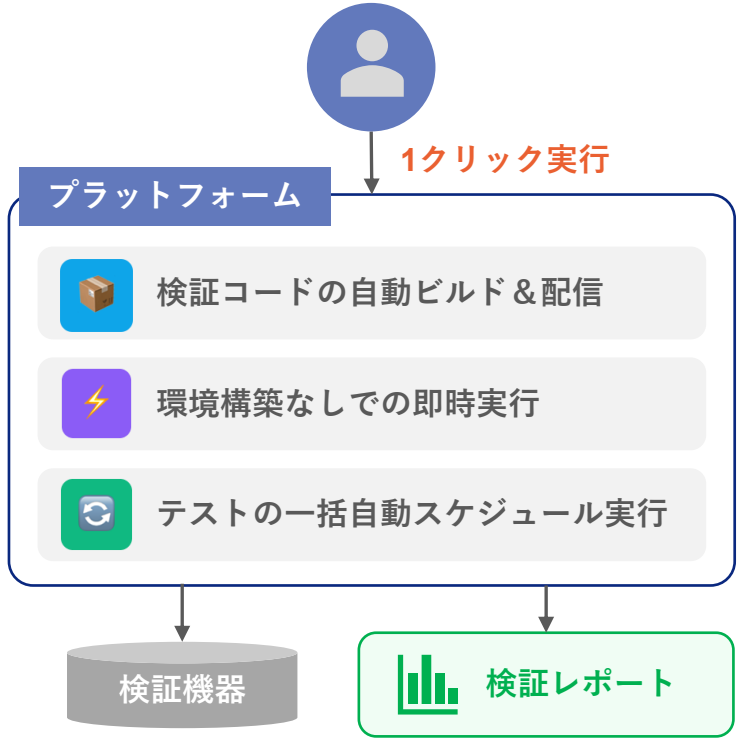
- ベンダ依存処理をシナリオから外部化
- パラメータの構造化と共通化
- 実装ルールをFWでパターン化

課題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

課題3 ユーザビリティの向上



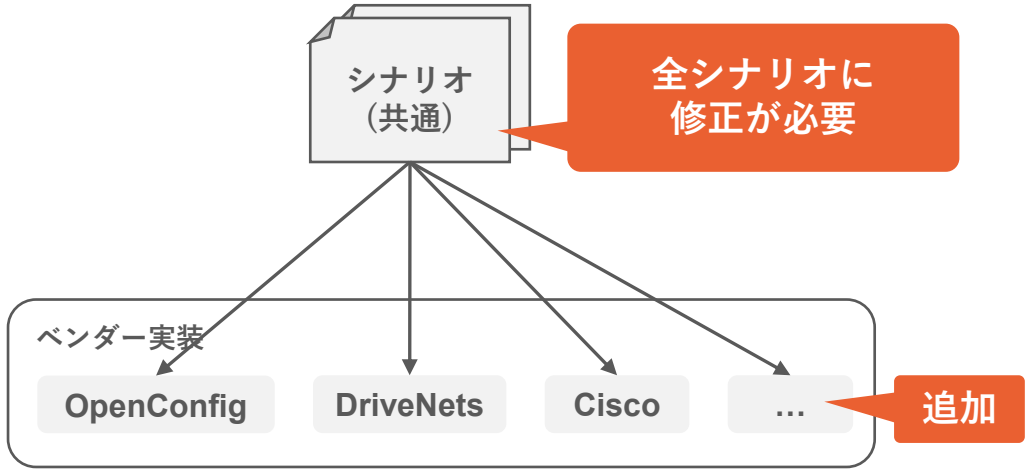
- 実行環境のプラットフォーム化
- CI連携による検証のワンクリック化
- 検証結果、レポートの自動生成

シナリオの抽象化

ONDATRAをベースとした
新しいフレームワーク 「Ondatra Catalog (OndaCat)」を内製開発
ベンダー追加時のコード変更量を最小化

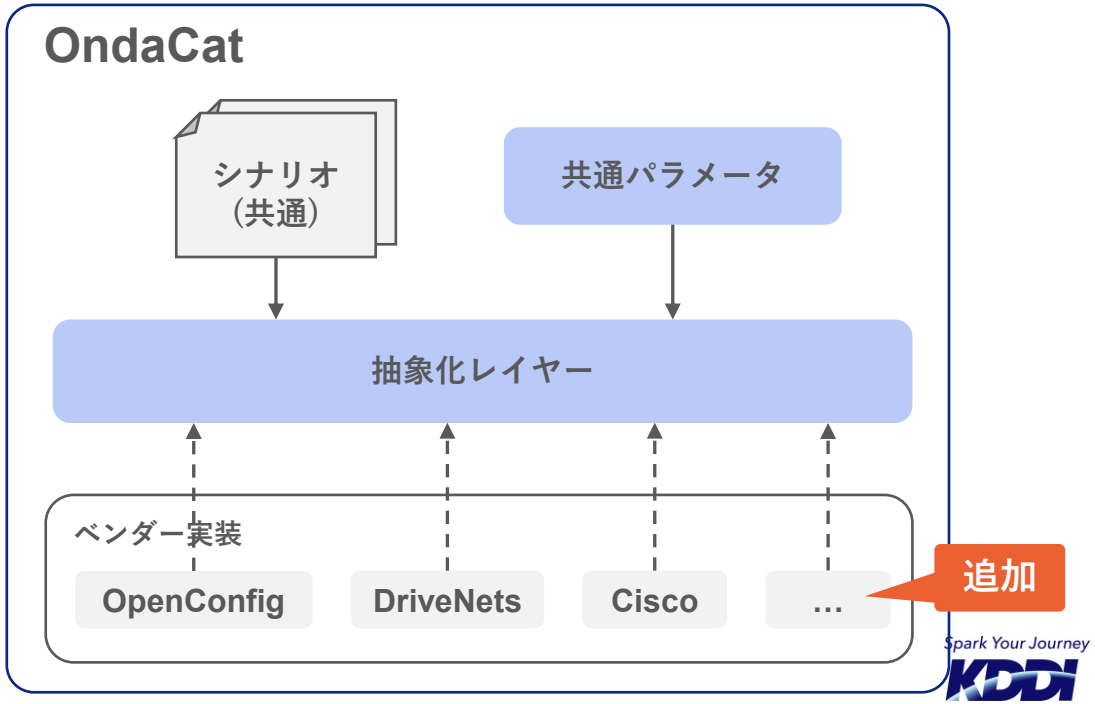
before

シナリオが直接ベンダー実装に依存するため、
「シナリオ」と「ベンダー実装」の
両方に修正・追加が必要



after

「依存関係逆転の原理（DIP）」を採用し、
依存関係を逆転させたことにより、
「ベンダー実装」のみの追加だけでOK



シナリオの抽象化

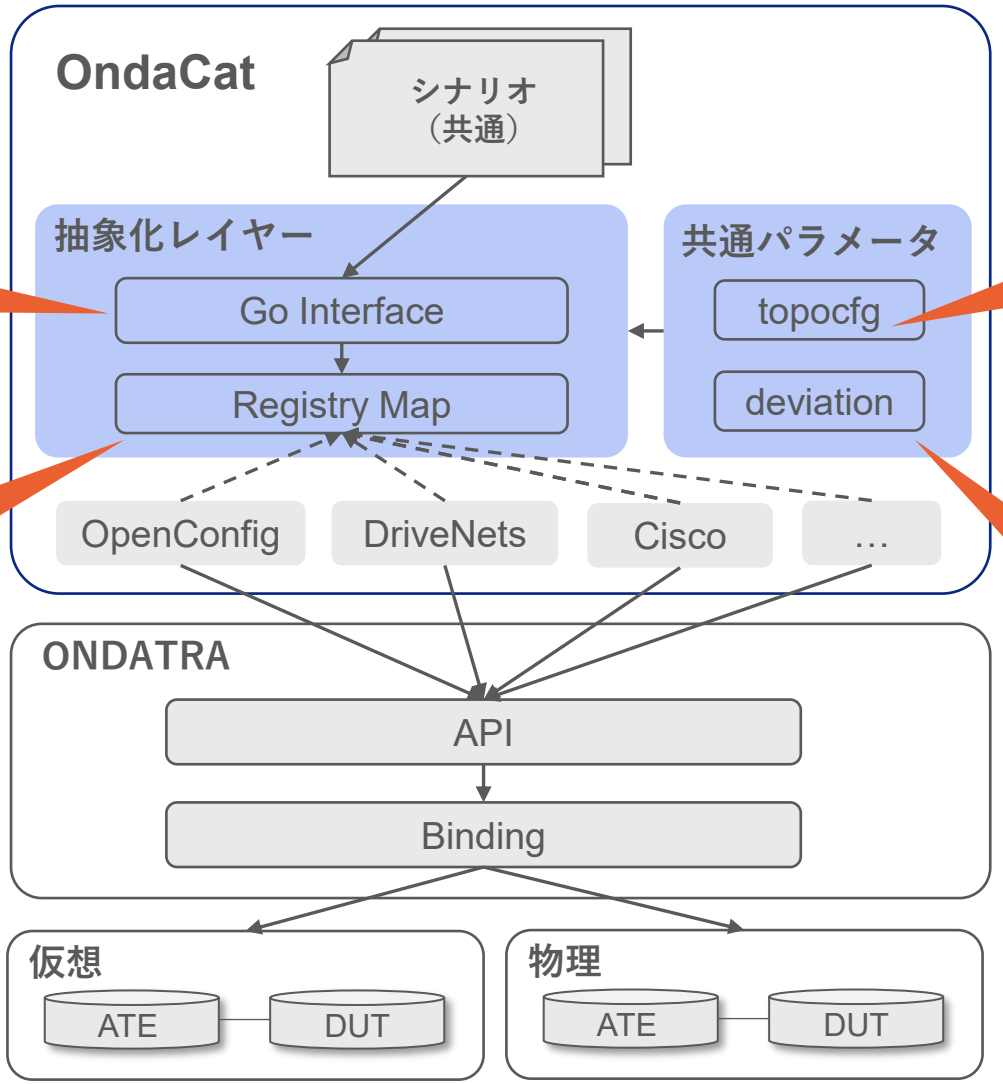
実装ルールをパターン化し、誰もが均質で疎結合なコードを記述できるように設計

ベンダー実装の呼び出しを
Go Interfaceによりパターン化

```
type OSPFService interface {
    ConfigureOSPF(t *testing.T, config *OSPFConfig)
    ConfigureOSPFArea(t *testing.T, ospf *OSPFConfig)
}
```

＊コードイメージ

各ベンダー実装を
レジストリ(Map)に登録し、
実行時にファクトリを介して
依存オブジェクトを注入しながら、
対象のインスタンスを
動的に生成・解決する



トポロジーに紐づくパラメータを構造化し、その呼び出しをパターン化

```
interfaces: {
  id: "port1"
  description: "DUT-ATE Link 1"
  mtu: 1500
  subinterfaces: {
    index: 0
    description: "DUT-ATE Link 1"
    ipv4_address: "192.0.2.1"
    ipv4_subnet_prefix: 24
```

Protobuf

デバイスの差分やOpenConfig対応
可否を構造化し、シナリオから分離

```
deviations: {
  interface_enabled_required: true
  ospf_config_requires_native: true
}
```

Protobuf

CLIパーサの削減

CLI実装の大部分をYANG/gNMIによるモデルベースの実装に置き換え ygot/ygnmiによりYANGからGoコードを自動生成し、型安全と開発効率性を両立

実装の優先度

優先度	実装手法	選択理由
高	OpenConfig	一度の実装で全ベンダーに適用可能。最も保守性が最も高いため
中	Vendor Native	OpenConfig未サポート時の次案
低	CLI	YANG未対応機能や、showコマンドのパースが必要な場合のみ限定的に使用

「型安全」と「Pathの自動補完」

goplsによる型チェック、コード補完

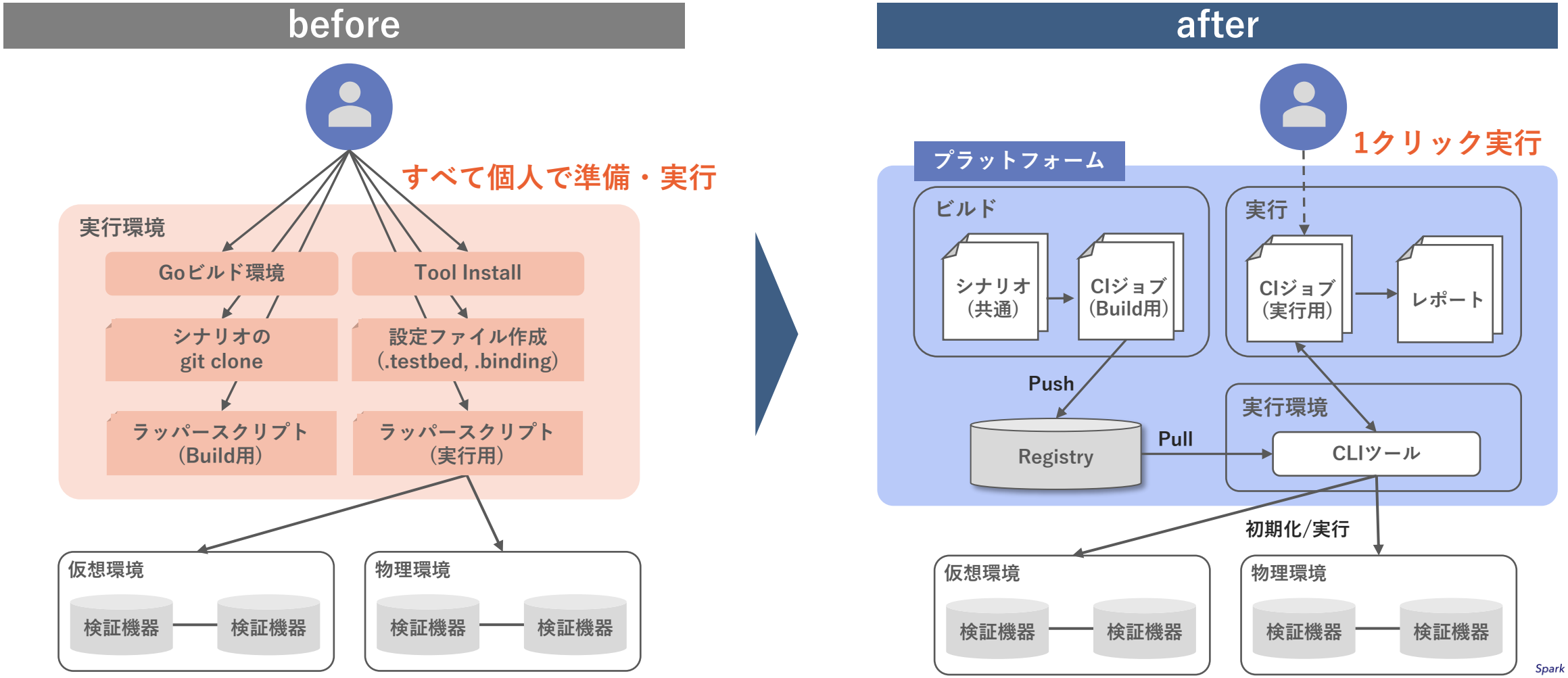
```
field Description *string `path:"state/description" m
path:"config/description" shadow-module:"openconfig-i
(oc.Interface).Description on pkg.go.dev
i.Description = ygot.String("Sample")
i.Enabled = ygot.Bool(true)
```

例：interface descriptionの型定義

例：/interfaces/interface[name]/

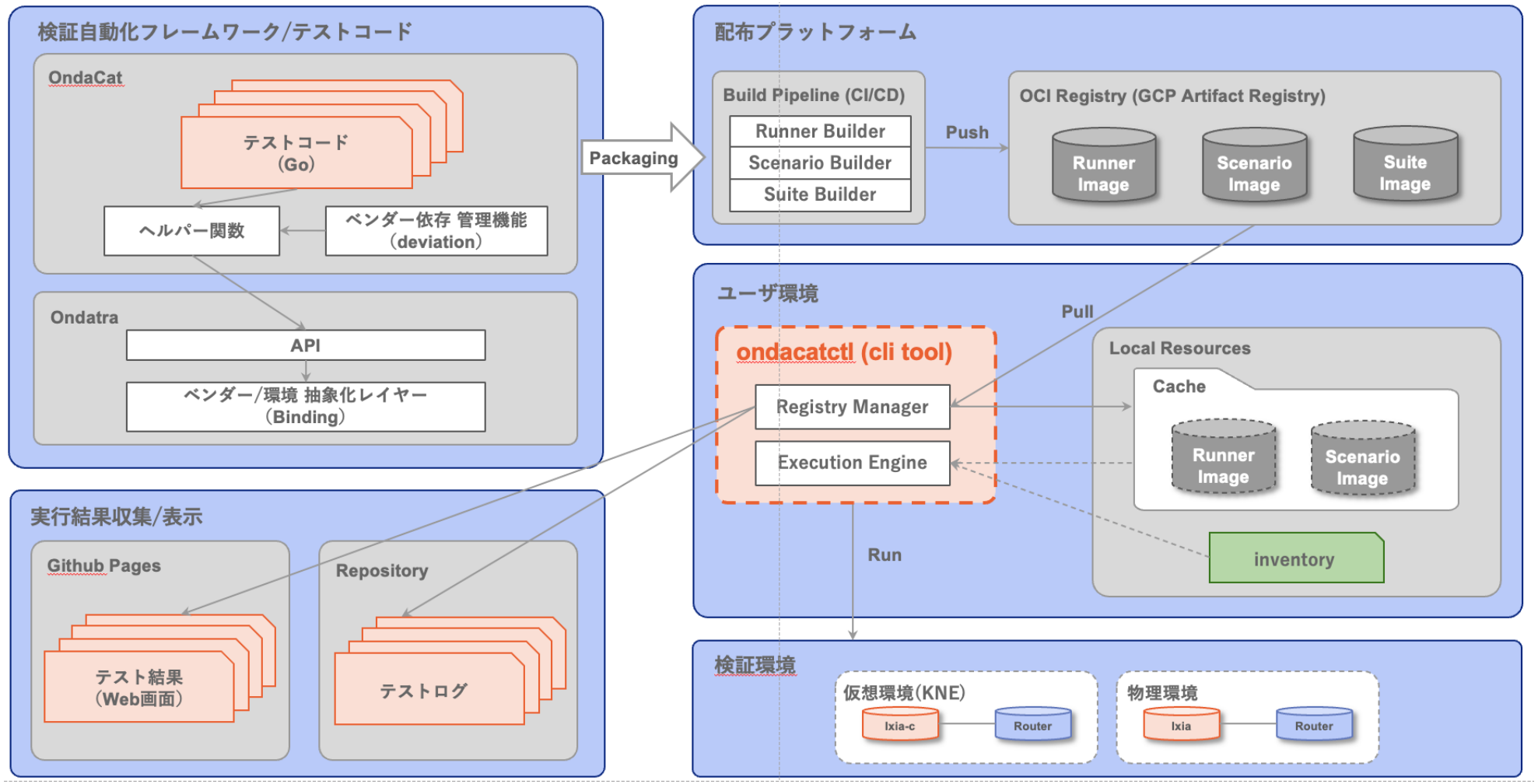
ユーザビリティの向上

複雑な実行環境をプラットフォーム化
ユーザはCIツールからワンクリックでテスト実行が可能に



ユーザビリティの向上

ビルド、バイナリ管理、実行やレポート作成等、テストの付帯機能をプラットフォームに集約



ユーザビリティの向上

検証をパッケージ化し、物理・仮想問わず「どこでも・即座に」再現が可能

環境構築の簡素化

ondacatctl をインストールするだけでシナリオを実行。プログラミング言語の環境構築が不必要。

1. シナリオのインストール（Pull）

```
$ ondacat install unit3-1-1:v0.0.1
```

2. シナリオの実行（Run）

```
$ ondacat run unit3-1-1:v0.0.1 --lab kne
```

検証環境の抽象化

実行環境の柔軟な切り替えを実現

```yaml
environments:
  kne:
    description: "KNE"
    architectures:
      "architecture1-1": "./binding/kne_archi1-1.binding"
      "architecture1-2": "./binding/kne_archi1-2.binding"
  tama5-ark:
    description: "TAMA5_ARK"
    architectures:
      "architecture1-1": "./binding/ark_architecture1-1.binding"
      "architecture1-2": "./binding/ark_architecture1-2.binding"
      "architecture1-7": "./binding/ark_architecture1-7.binding"
      "architecture1-17": "./binding/ark_architecture1-17.binding"
      "architecture1-22": "./binding/ark_architecture1-22.binding"
```

inventory.yaml

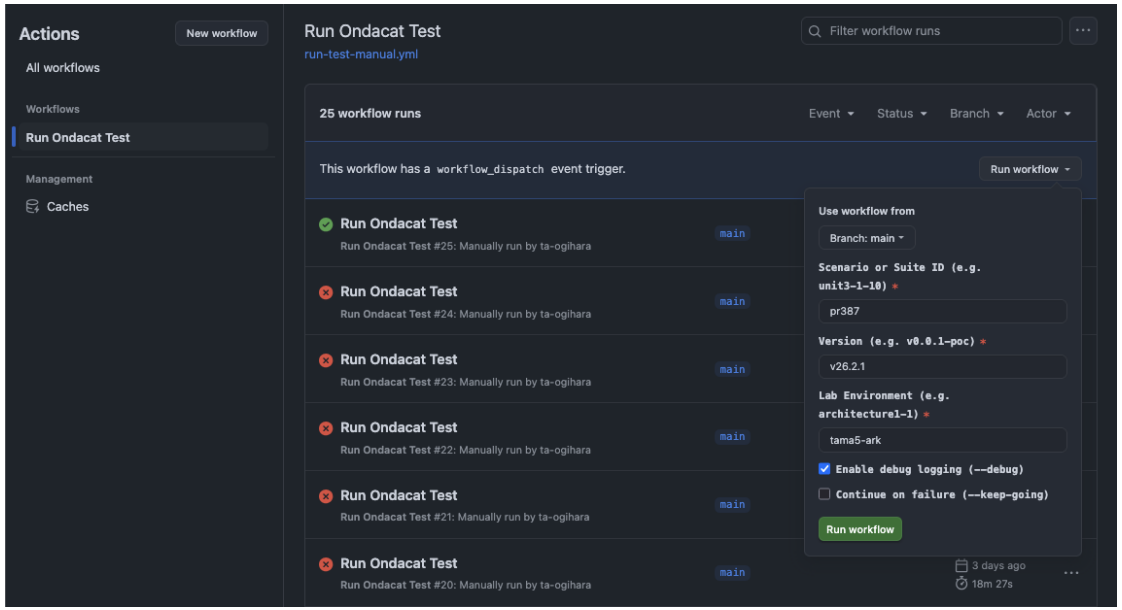
シナリオの一括実行

シナリオを機能単位(Suite)で管理

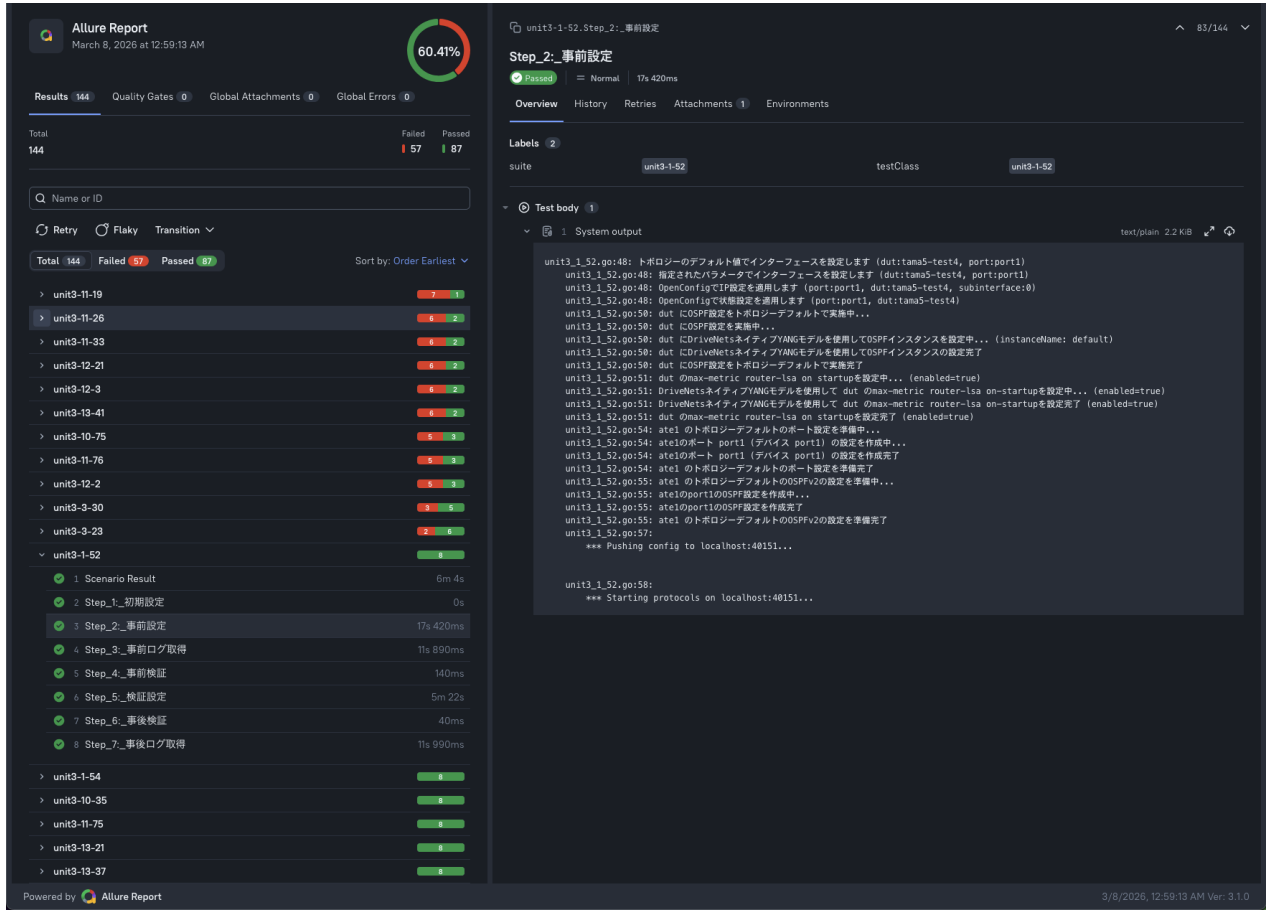
```
runner@nxnet-vm-kne-ogihara:~$ ondacat suite describe pr387:v26.2.0
Suite:       pr387
Version:     v26.2.0
Description: pr387 scenarios collection

Scenarios:
ID            ARCHITECTURE
unit3-1-52    architecture1-2
unit3-1-54    architecture1-2
unit3-3-9     architecture1-2
unit3-3-23    architecture1-1
unit3-3-30    architecture1-22
unit3-11-75   architecture1-2
unit3-13-21   architecture1-17
unit3-13-37   architecture1-1
unit3-13-41   architecture1-17
unit3-10-35   architecture1-1
unit3-10-75   architecture1-7
unit3-11-19   architecture1-17
unit3-11-26   architecture1-17
unit3-11-33   architecture1-17
unit3-11-76   architecture1-2
unit3-12-2    architecture1-2
unit3-12-3    architecture1-2
unit3-12-21   architecture1-17
runner@nxnet-vm-kne-ogihara:~$
```

ユーザビリティの向上

GitHub ActionでSuiteを自動実行。
検証結果をレポートとして生成し、GitHub Pagesに自動アップロード。



GitHub ActionからSuiteを実行



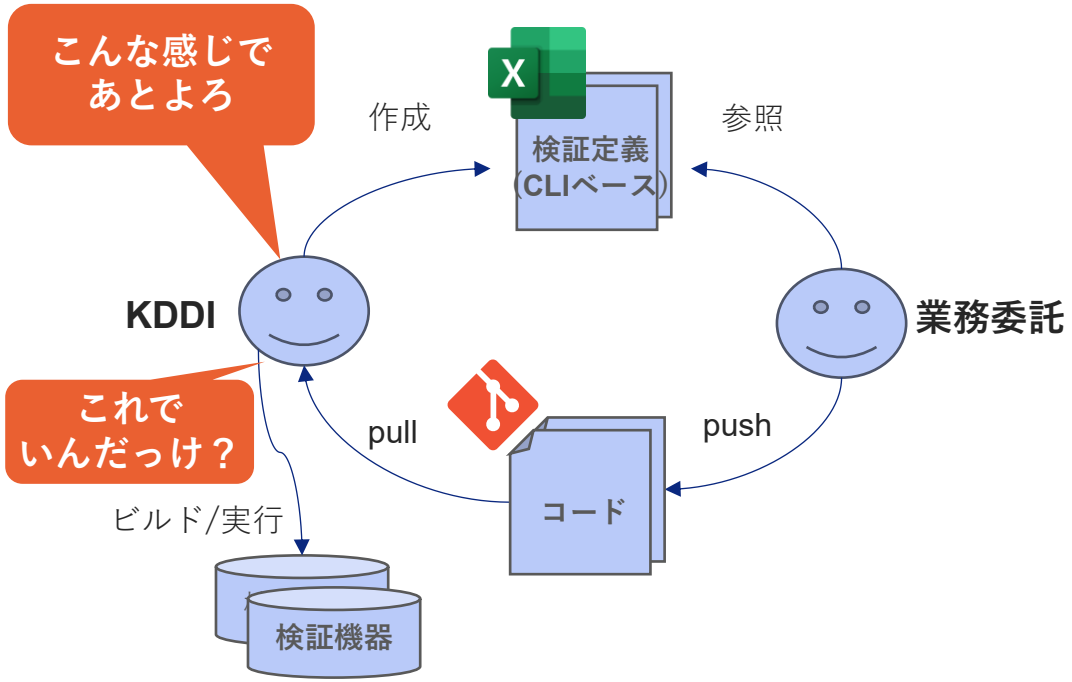
GitHub Pagesにアップロードされる検証結果

デモ

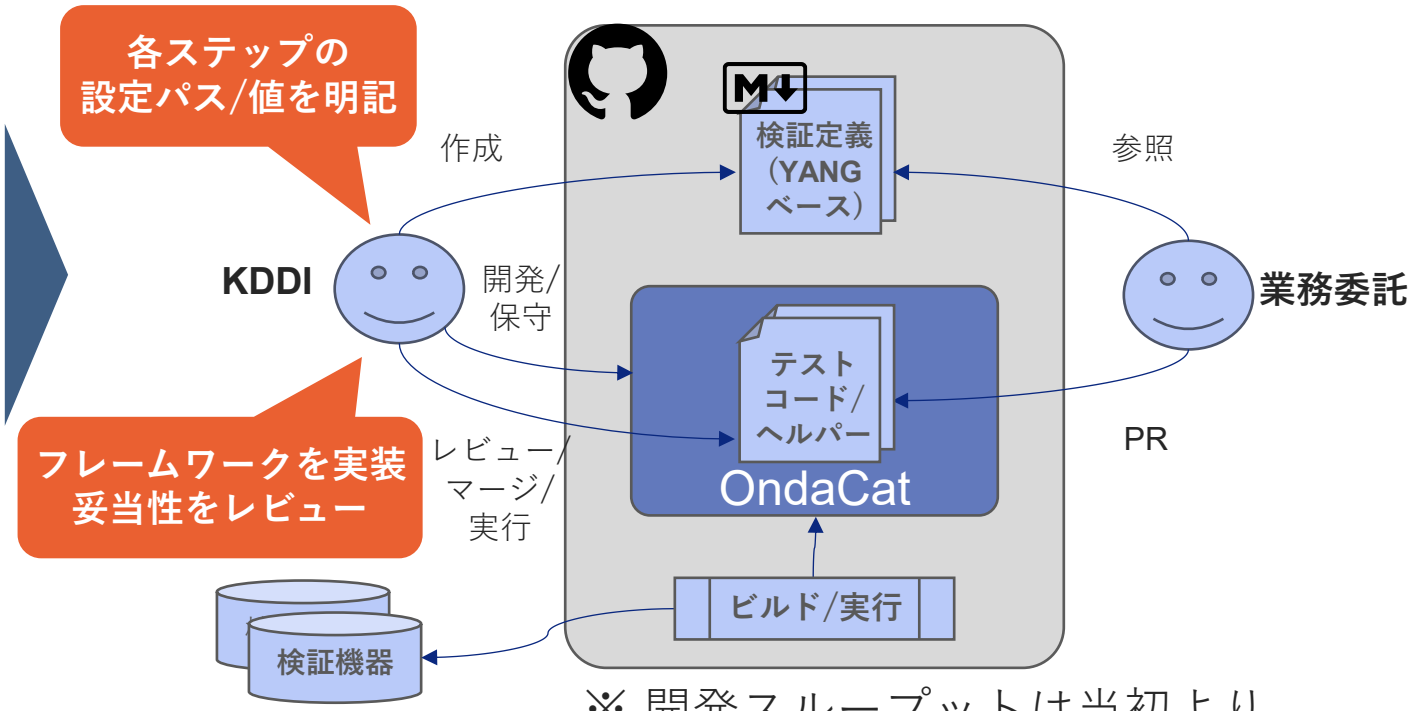
開発スキームの見直し

フレームワークやサンプル、実装ガイドラインをKDDIで作成
全てのコードをKDDIでレビューすることで品質や妥当性を担保する形にしました

当初の体制



変更後の体制



※ 開発スループットは当初より落ちました🤑

中身が分かる、自分達で改善できるコードベースに⚒️

得られた成果と知見

成果: OndaCat と プラットフォームの導入効果

1. 検証業務

・新規筐体の導入に関わる基本機能検証において、テストコードを利用した検証の内製化を実現

・検証時間が **3ヶ月 -＞ 3日 に大幅短縮**

2. 検証実施の容易性

・どの環境でもCLIツールをインストールするだけで検証が実現可能

・**ワンボタンで実行、Web画面上で結果とログ確認が可能に**

3. コードの拡張性、保守性

・シナリオと振る舞いを分け、インターフェースで実装を抽象化

・**ベンダー/デバイスの新規追加でシナリオコードの変更が不要に**

知見: テスト自動化におけるソフトウェアデザイン

要点1

初期設計の重要性

後からの仕様やアーキテクチャの変更は極めて困難を伴うため、
「まずは小さく始めて安定させ、その後に全体展開（大規模化）する」
アプローチを徹底する。

要点2

フレームワークによる規約の実装

自由度の高いONDATRAに対し、**OndaCatでシナリオやパラメータの実装ルールを「規約」として制御。**
これにより、実装バリエーションの拡散を防ぎ、開発とレビュー効率が向上。

知見: モデルベースの現状と展望

現状ではOpenConfigのカバレッジは低く、ベンダネイティブモデルが主
他方で、CLIベースの限界を改めて痛感

OpenConfig/OTGのカバレッジ

Interface関連

50%~: 基本的な設定は概ね対応済。Carrier Delay等細かい設定で未対応/非互換が出てくる

プロトコル関連

数%~数十%: ベンダ次第で振れ幅大。BGP, IS-IS等OTT/海外オペレータ利用プロトコルは高い傾向あり

測定器/OTG関連

100%: IXIAでは利用する全操作をOTG APIで実装できた

→ OpenConfigは途上、IXIAのOTGは十分使える

CLIが残っている場所

ルーティングテーブル関連

gRIBI(※)の範疇だがKDDIではまだ未導入

例: RIBの参照, 経路の注入,,,

※ gRPC Routing Information Base Interface

システムオペレーション関連

gNOI(※)の範疇だがKDDIではまだ未導入

例: プロセス再起動, BGPセッションクリアー,,,

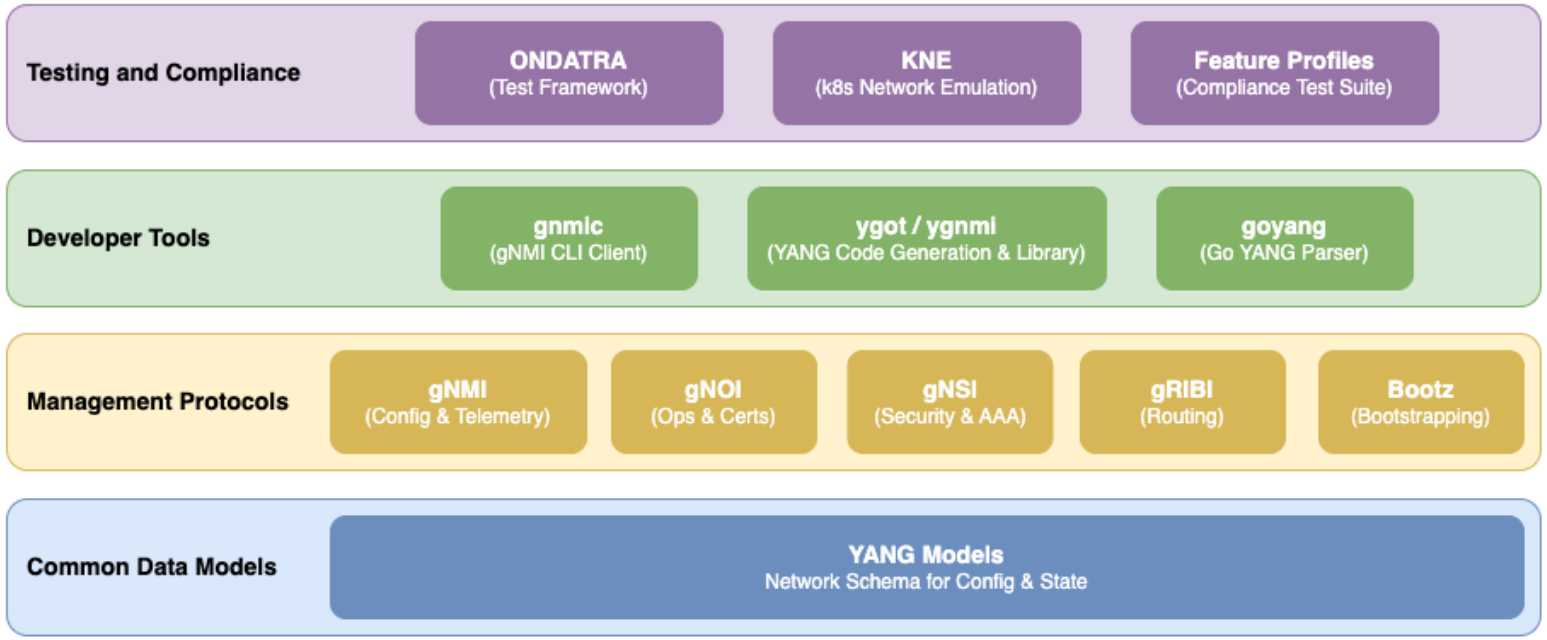
※ gRPC Network Operations Interface

→ CLI実装は20%程まで削減

知見: OpenConfigエコシステムの有効活用

OpenConfigプロジェクトはモデル関連のライブラリやツールも提供している これらを用いればより効率的に開発ができる

OpenConfigエコシステム



＊主要なプロジェクトのみ記述

https://github.com/openconfig

gnmic： YANGモデルの解析, CLIベースのgNMI
ygot/ygmni： YANGに対応する型/操作メソッドの
featureprofiles： ONDATRAで実装されたテストコード（Googleが利用）
KNE： k8sベースの仮想環境

関連するツール群

YANGModels/yang

いろんなベンダーのYANGモデルのカタログ

https://github.com/YangModels/yang

open-traffic-generator/models

OTGのモデル定義

https://github.com/open-traffic-generator/models

OTG API

OTGのAPIリファレンス

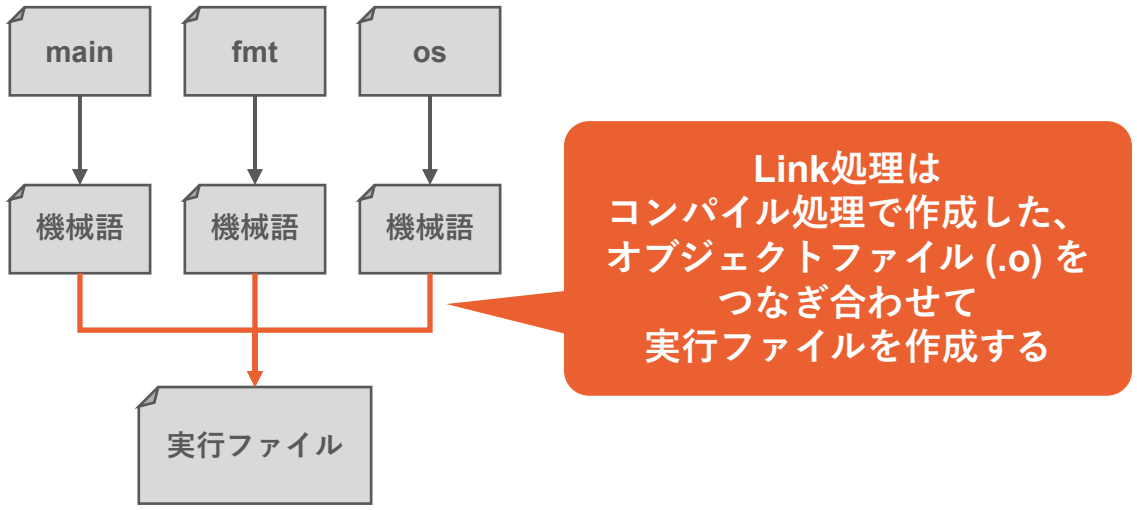
https://redocly.github.io/redoc/?url=https://raw.githubusercontent.com/open-traffic-generator/models/master/artifacts/openapi.yaml&nocors

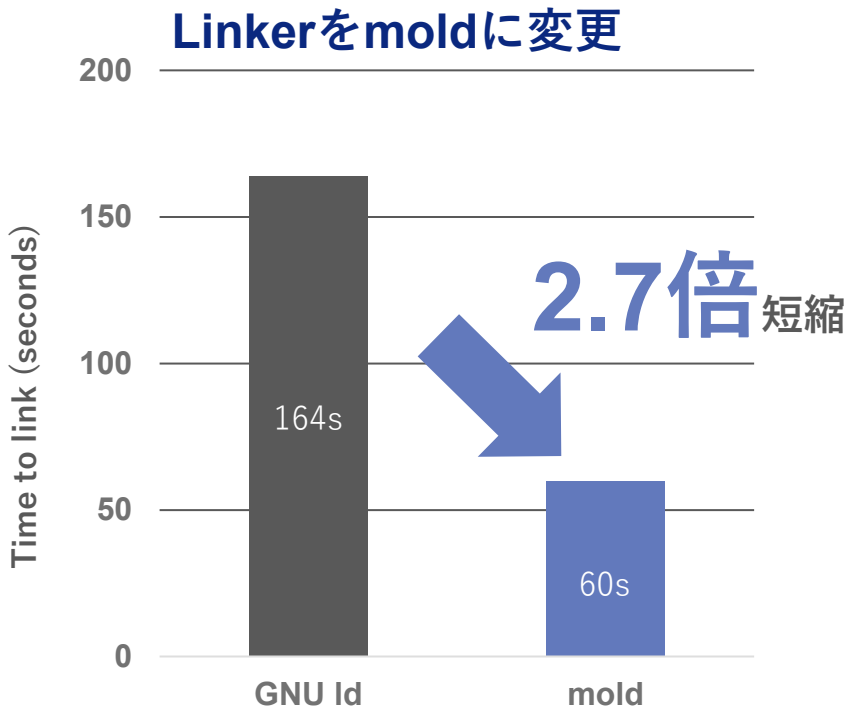
知見: 高速リンカーによるビルド時間の短縮

モデルによってはYANGから生成したコードが巨大化しビルドが長時間化するが、高速なリンカーを導入すればビルド時間を大幅に短縮できる

課題と対処

ygot生成による巨大パッケージ（数十万シンボル）の影響で、従来のLinkerではビルドに約3分要し、コード開発に支障をきたしていた。
これに対し、高速リンカ**「mold」を導入したことでビルド時間を短縮**し、開発の高速化を実現。





今後の取り組み

今後の取り組み

1. テストのカバレッジ拡大

・対応ベンダー/機器の拡充
・対応シナリオの拡充

2. AI活用の高度化

・コーディング/検証定義におけるYANGモデルパース/解析精度の向上(Skills/Tool開発)
・各種作業の精度向上に向けたチューニング/適切なハーネスの見極めと設置

3. 機能拡張

・テスト環境の構成に基づく動的なbinding生成
・OpenConfigエコシステム（ygot/ygnmi, featureprofiles）へのコントリビュート
・仮想環境の利便性向上(containerlab/clabernetes)
・gRIBI, gNOIへの対応

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

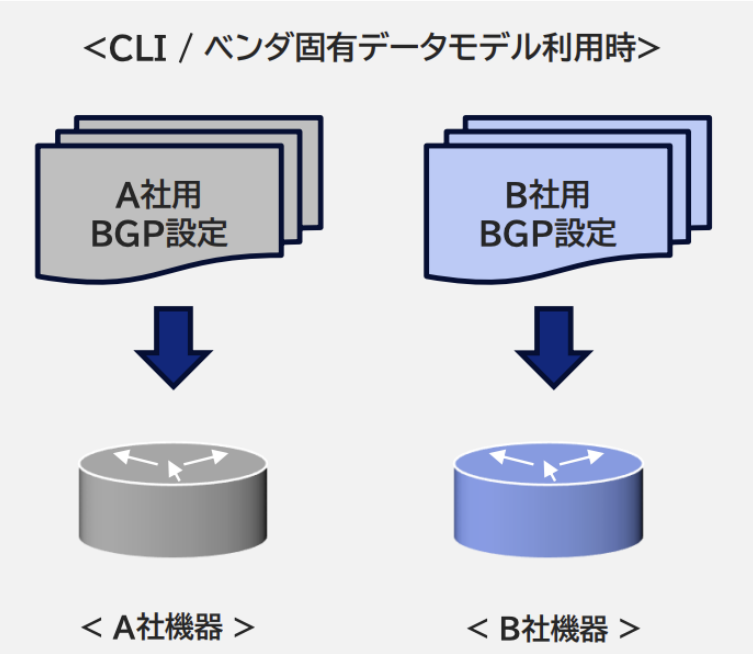
OpenConfigとは？

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

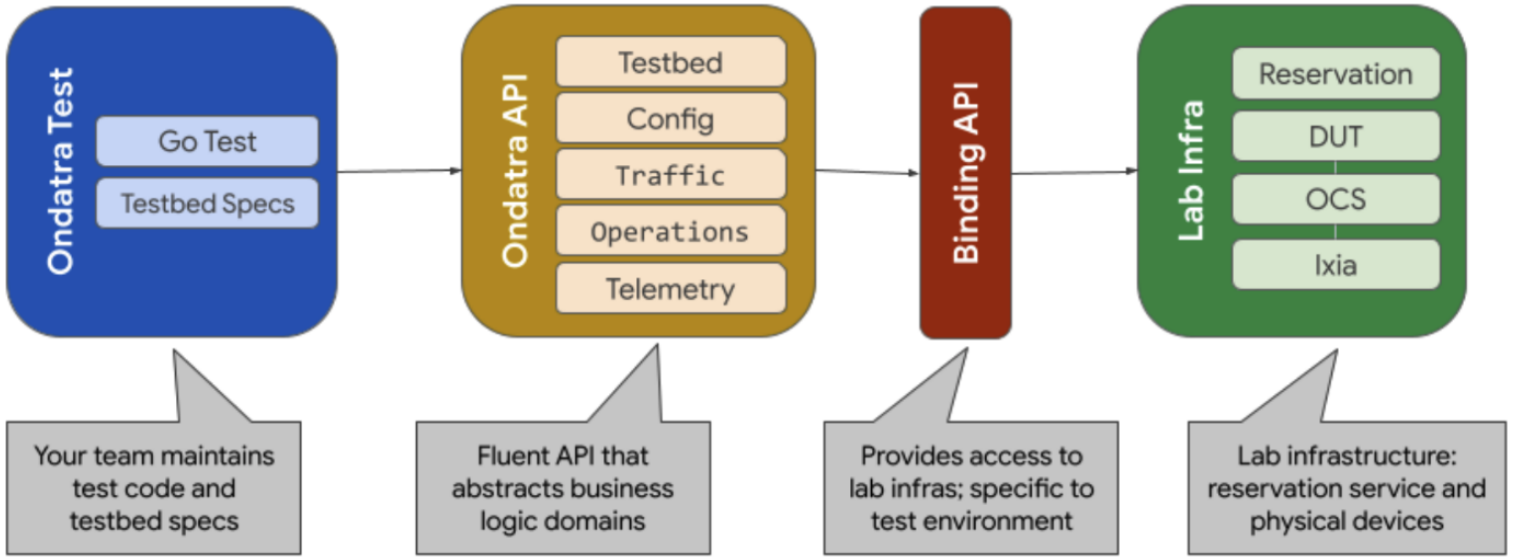
ルーティング／スイッチング/光トランスポート等の
領域において、**主要ベンダの多くのプラットフォームで実装**されている

< 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

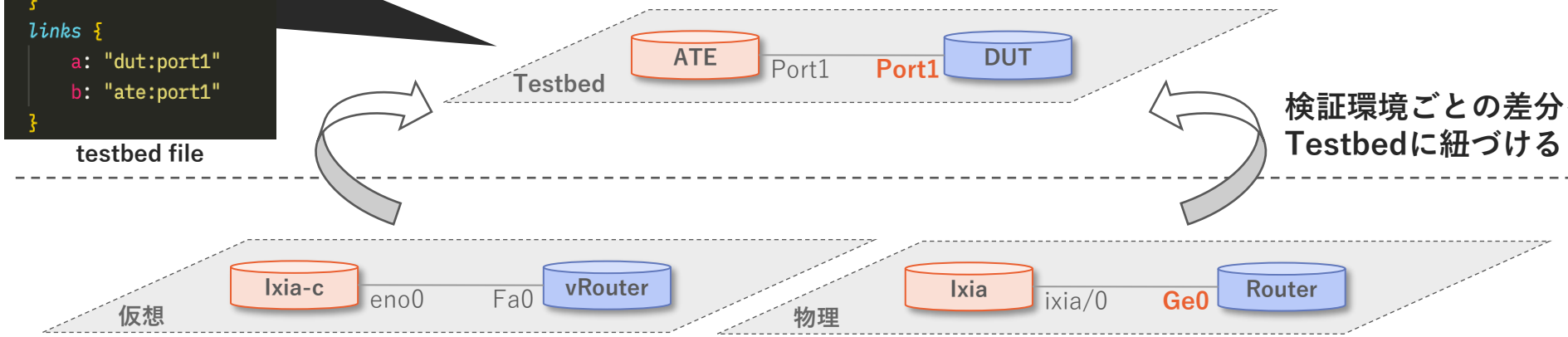
```
duts {
    id: "dut"
    ports {
        id: "port1"
    }
}
ates {
    id: "ate"
    ports {
        id: "port1"
    }
}
links {
    a: "dut:port1"
    b: "ate:port1"
}
```

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



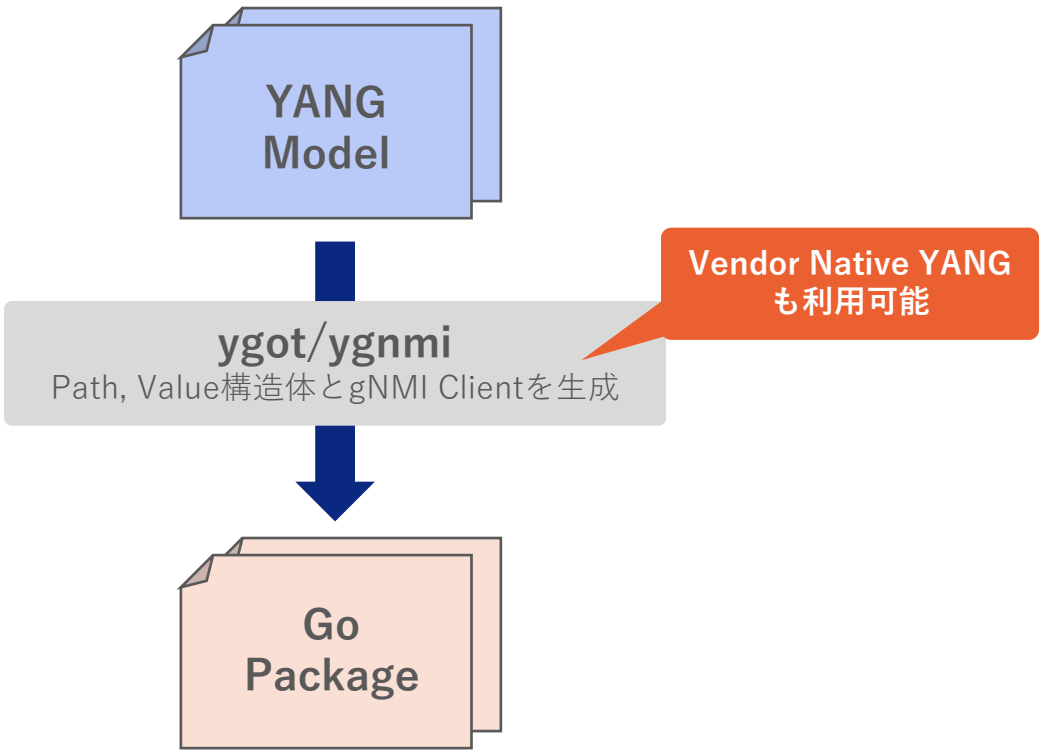

```
duts {
  id: "dut"
  name: "dut"
  vendor: DRIVENETS
  options {
    username: "dnroot"
    password: "dnroot"
    skip_verify: true
  }
  gnmi {
    target: "172.18.0.54:50051"
  }
  ssh {
    target: "172.18.0.54:22"
  }
  gnoi {
    target: "172.18.0.54:50051"
  }
  ports {
    id: "port1"
    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