Plankton: Scalable network configuration verification through model checking

Santhosh Prabhu[1], Kuan-Yen Chou[1], Ali Kheradmand[1], P. Brighten Godfrey[1], Matthew Caesar[1]

[1]University of Illinois at Urbana-Champaign

動機

- もともとこの論文が読みたかった（が、前回は挫折した）
- Network の分野における model check などの検証手法がどのように利用されているのか知りたい
- nsdi20 ということだしひとまず最新の論文で現状何ができていないのかざっくり掴みたい
- network/verification の両分野ともに明るくないのでこれからちょっとずつ取り込んでいきたい第一歩

Contents

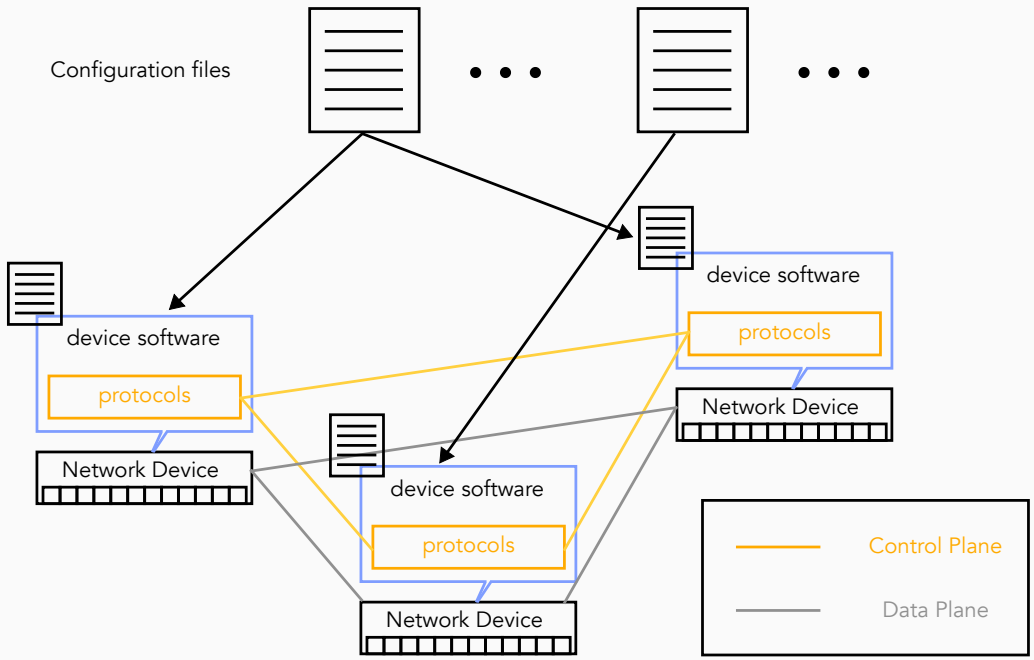
概要

概要

- 現在、ネットワーク設定検証 (Network Configuration Verification) において実用的なサイズのネットワークに耐えうる検証ツールは無い
- これを解決するものとして Plankton を提案
  - 複数 Protocol(OSFP, BGP) に対応する model を構築し SPIN(モデルチェッカ) で model check を行う
  - 様々な最適化により、既存の手法に比べ非常に高速、かつメモリ利用量も抑えることができる
- 様々な条件のベンチマークにおいて実用に耐えうることを確認した

背景

- Enterpise Network を管理・運用するのは大変。
  - 設定ファイルを記述し機器に投入することで想定した動作をさせる
  - 当然、設定にバグが混入すると事故る
- ネットワークの設定を投入する前に意図した振る舞いになるかどうかを予め確認できると嬉しい！！



問題

- Control Plane のモデル化の問題はおよそ以下に起因する状態の膨大さ
  1. ネットワークは動的に変化する (link/node failure, failure reaction)
  2. ネットワークの状態は非決定的である。
  3. ネットワークは多様である（同じネットワークで OSPF, BGP, IS-IS などが混在）
- 現実的なネットワークサイズを対象に実用的な速度の検証は困難
- 検証方法、最適化など多方面から様々な研究が行われている

関連 Tool

- 関連するようなツールはすでにいくつか存在している（次項）
- しかし、やはりネットワークのスケールに従って実用度が下がる
  - 多くのツールは速度を担保するために correctness/expressiveness を犠牲に。。。
- Minesweeper はこれを落とさないように頑張った
  - Scalability 以外については満たしている非常に強力なツール
  - 245 台のデバイスネットワークのループチェックに 4 時間以上かかる
- 現実問題として scalability までも満たすようなものって作れるの？

関連 Tool

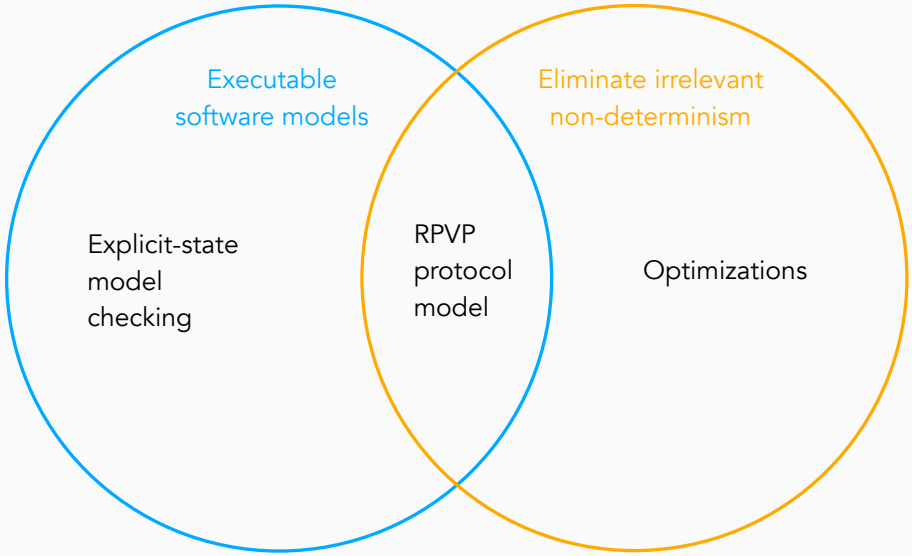
Challenge	**Batfish** Simulation	**ARC** Graph algorithm	**ERA** Symbolic execution	**Minesweeper** SMT	**Plankton**
Model failure	✘	✓	✘	✓	✓
Model protocol non-determinism	✘	✘	✓	✓	✓
Guaranteed soundness	✓	✓	✘	✓	✓
Support different protocols	✓	✘	✓	✓	✓
Scalable	✓	✓	✓	✘	✓

問題の分析

- Minesweeper を例に考えると scalability の問題は検証手法による
- Minesweeper では経路計算を完全に別ドメイン (SMT constants) に変換し、SMT Solver で解く
  - 決定的な経路計算を対象に取ると software model execution のほうが早い
  - 簡単なテストをしたら x1000-x10000 くらいの差が出た
- とはいえ基本的に現実世界は非決定的な事象の取り扱いになる
  - でもその非決定性、ほんとに考える必要あるの？
  - 非決定的な事象について考えるべきもの／考えなくてもいいものに分ける
  - もし必要な非決定的な事象のみに刈り込めればスケールしても実用的な速度で検証できるのでは?

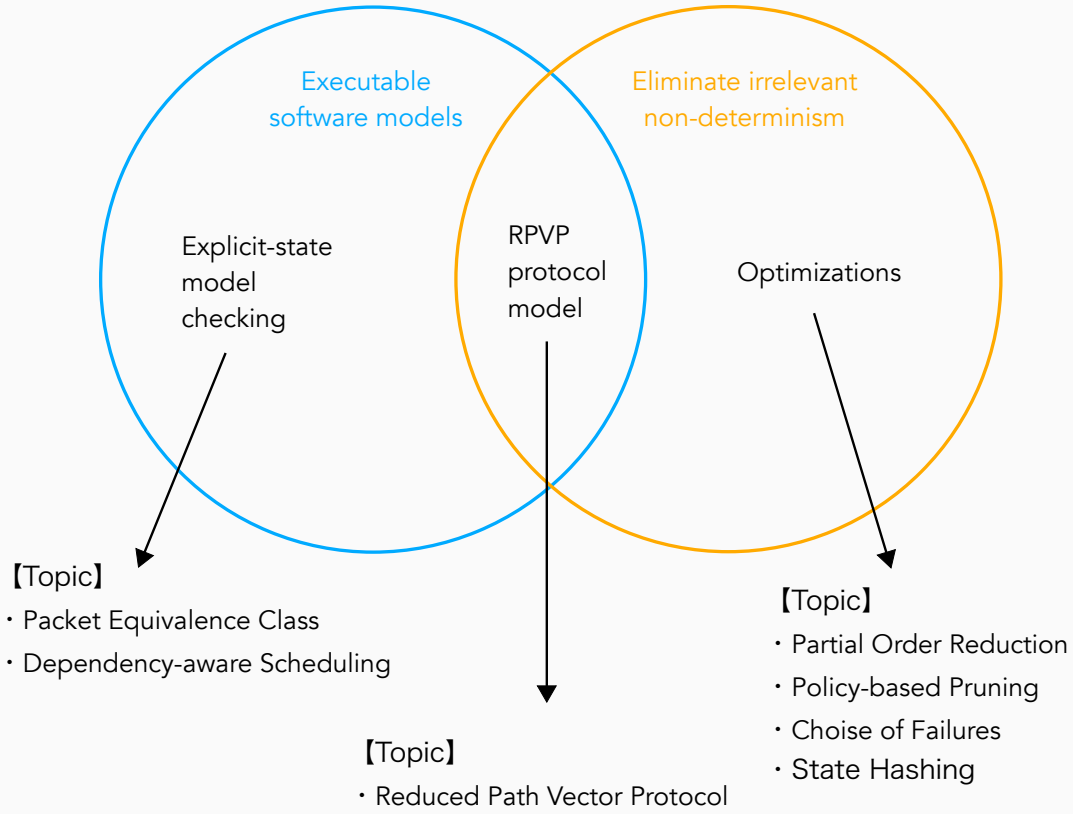
整理

- 前項の直感をまとめると下図
- 検証手法としては software model check を利用する。
- RPVP という protocol を提案し OSPF, BGP などを model 化する。
- 非決定的な事象について刈り込むために、各種最適化を行う



整理 - 詳細

- ここから話す topic との対応をつけたのが下図



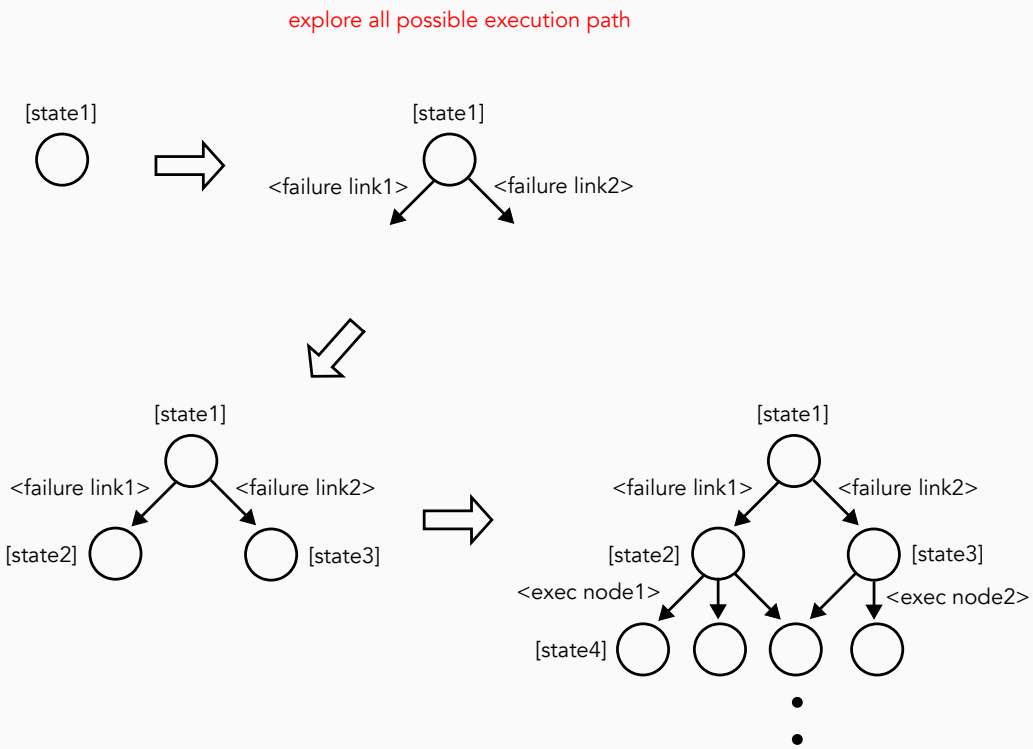
論文の目的

- packet equivalence computation と explicit-state model checking を組み合わせた configuration verification パラダイムの定義
- 実用的な規模のネットワークで実現するための各種最適化
- OSPF, BGP, Static Routing をサポートした Plankton prototype の実装と実用規模で検証可能であることを示し、最先端のものと比較し 4 桁以上の差をつけた

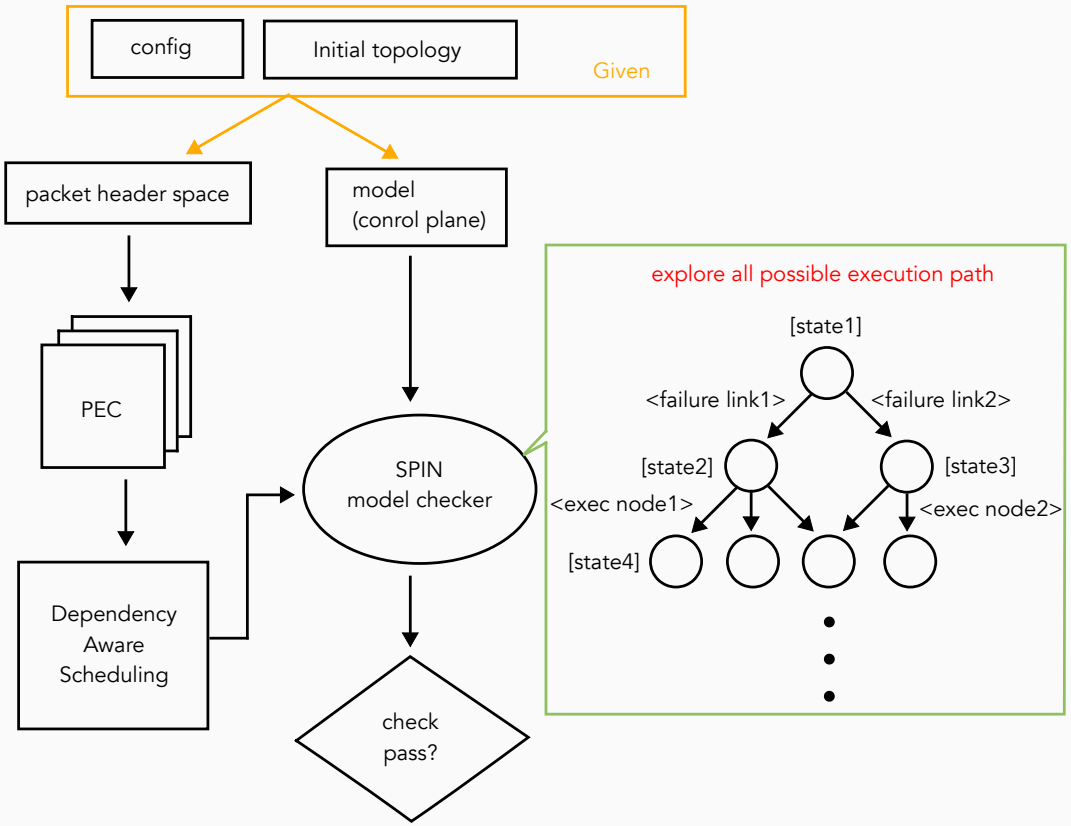
Explicit-state Model Checking

Explicit-state Model Checking

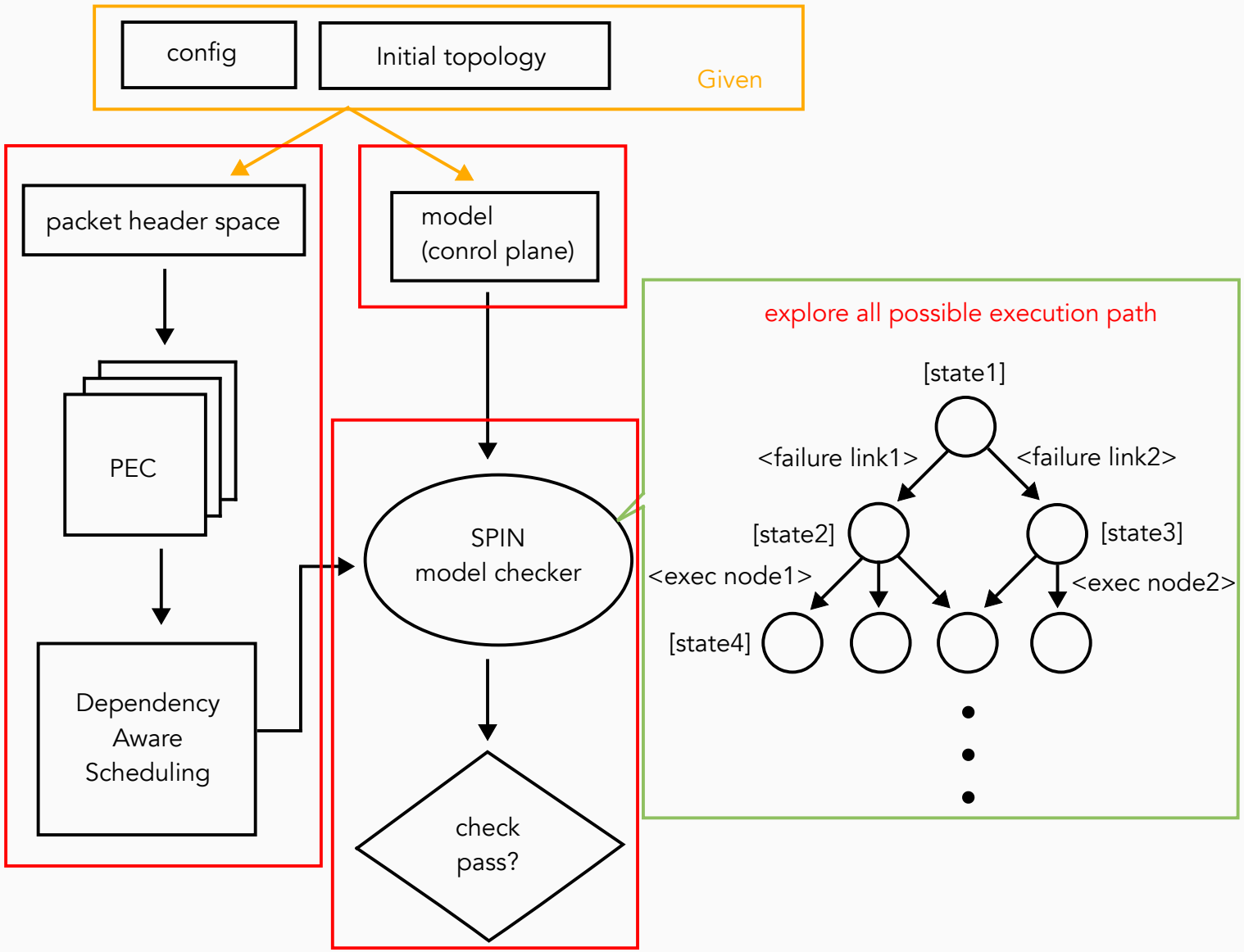
- SMT ではなく Explicit-state model checking (明示的状態モデル検査) により検証を行う
- program の model が与えられた時、その実行パスを網羅的に探索する
- 今回の例では、conrol plane を program として model 化する
- Network の state がどのように変更していくかは non-deterministic



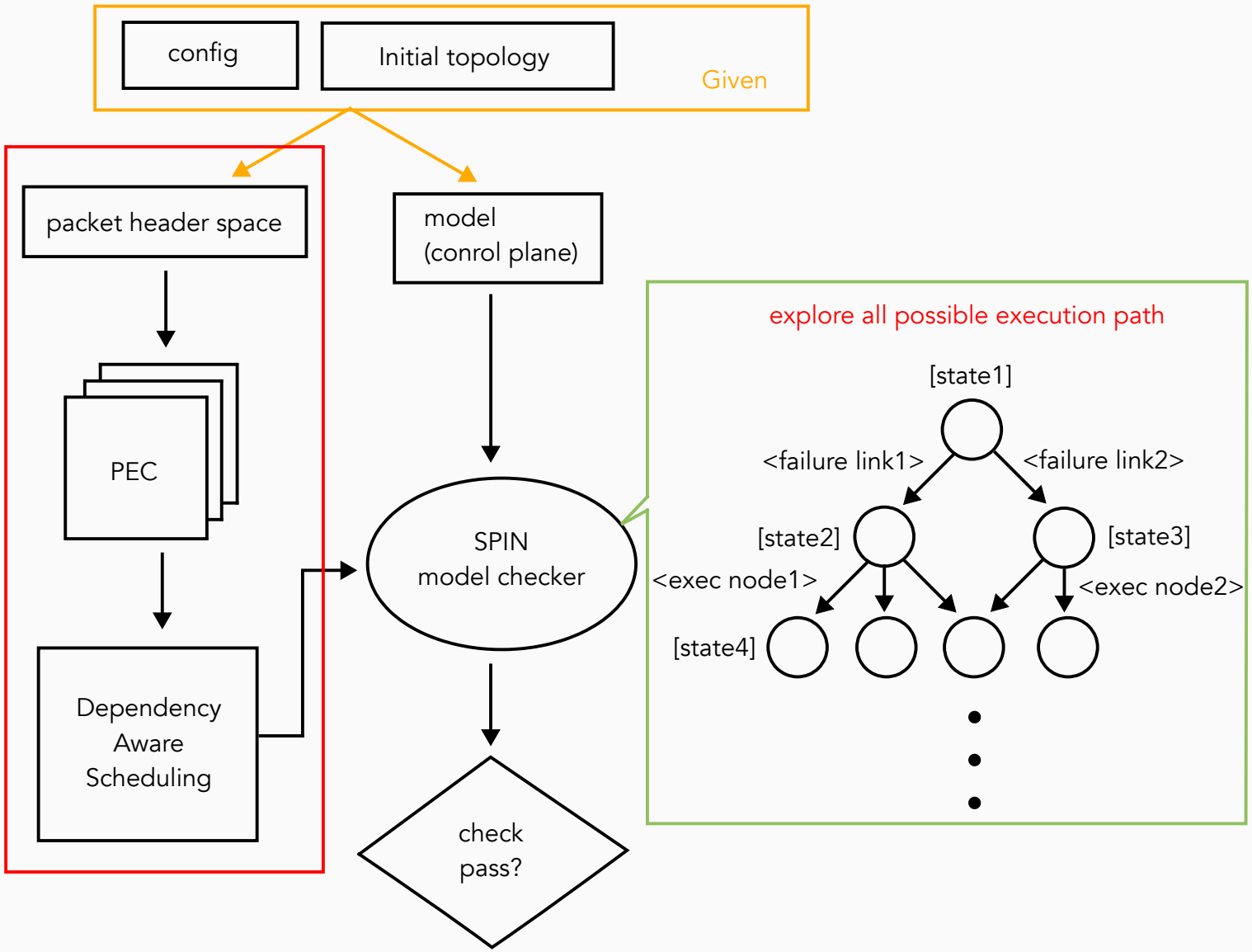
Plankton Design Overview



Plankton Design Overview



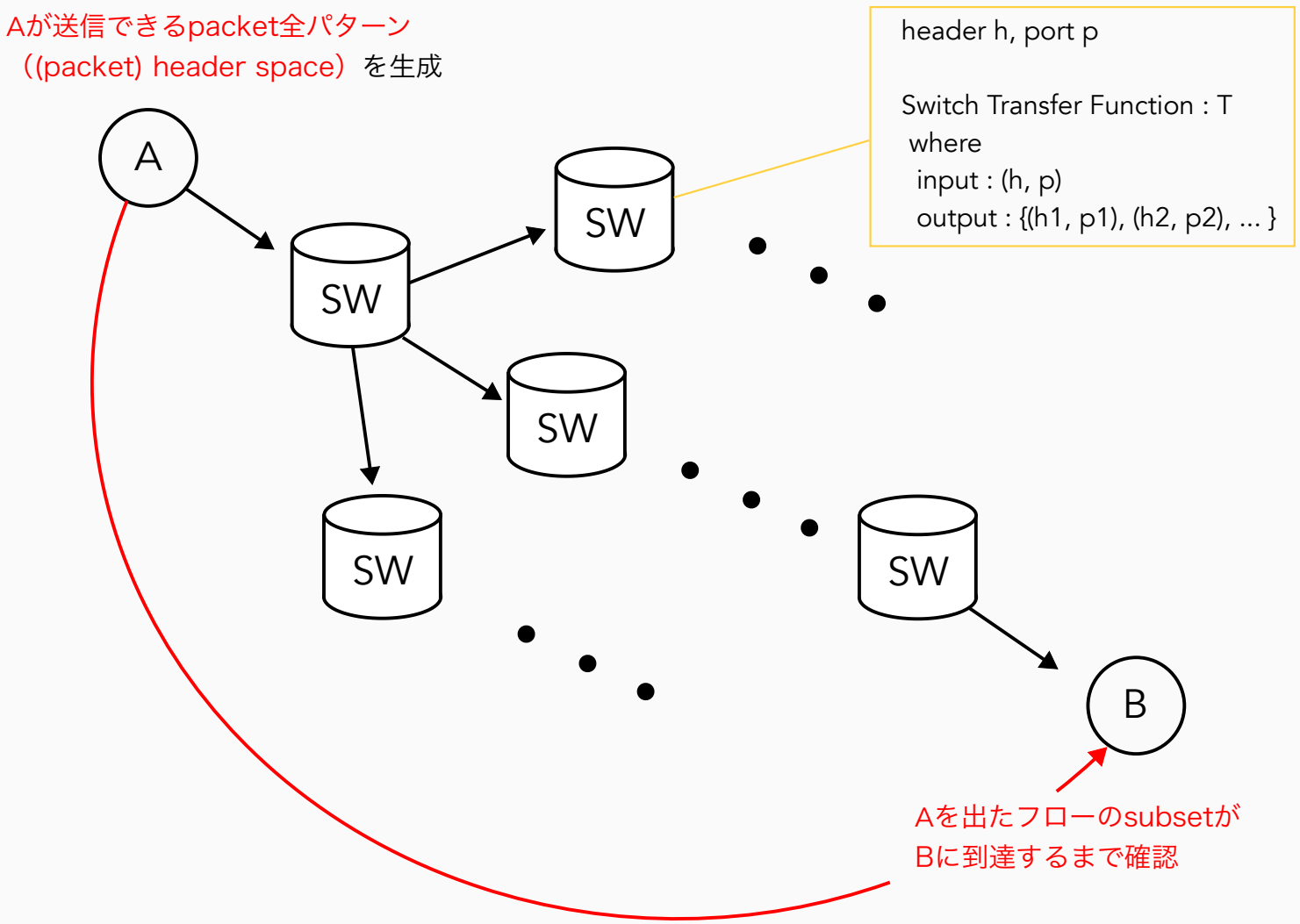
Plankton Design Overview



PEC & Dependency-aware Scheduling

- 探索する Packet レベルの施行をいかに削るか、という話。
- しかし、依存関係とか考える必要があったりするのでそれは考える
- 以下の流れで説明をしていきます
  1. Header Space Analysis
  2. Packet Equivalence Class(PEC)
  3. Stringly Connected Components (SCC)
  4. Dependency-aware Scheduling

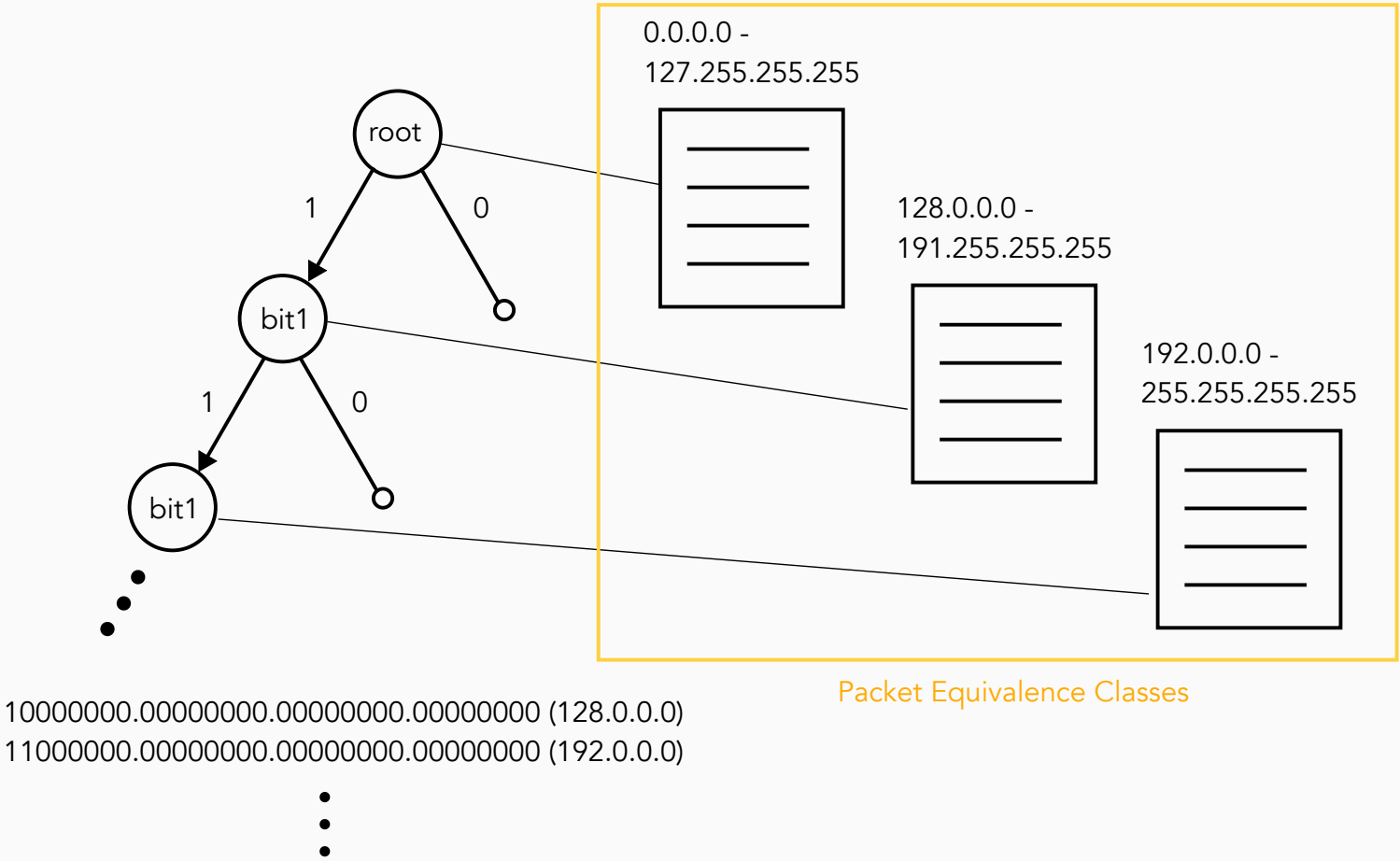
Header Space Analysis(HSA)



Packet Equivalence Class

- Packet の Equivalence を考え packet をまとめることで計算量を削減する
- Equivalence はどうするの？
  → 同一設定内容の適用範囲で分ける (Prefix ベースの設定適用が前提になっていそう)
- トライ木で Prefix 単位でノードを伸ばしていき、各 node(Prefix) に対して configuration が紐付いている感じ。

Packet Equivalence Class



10000000.00000000.00000000.00000000 (128.0.0.0)
11000000.00000000.00000000.00000000 (192.0.0.0)

Strongly Connected Components (SCC)

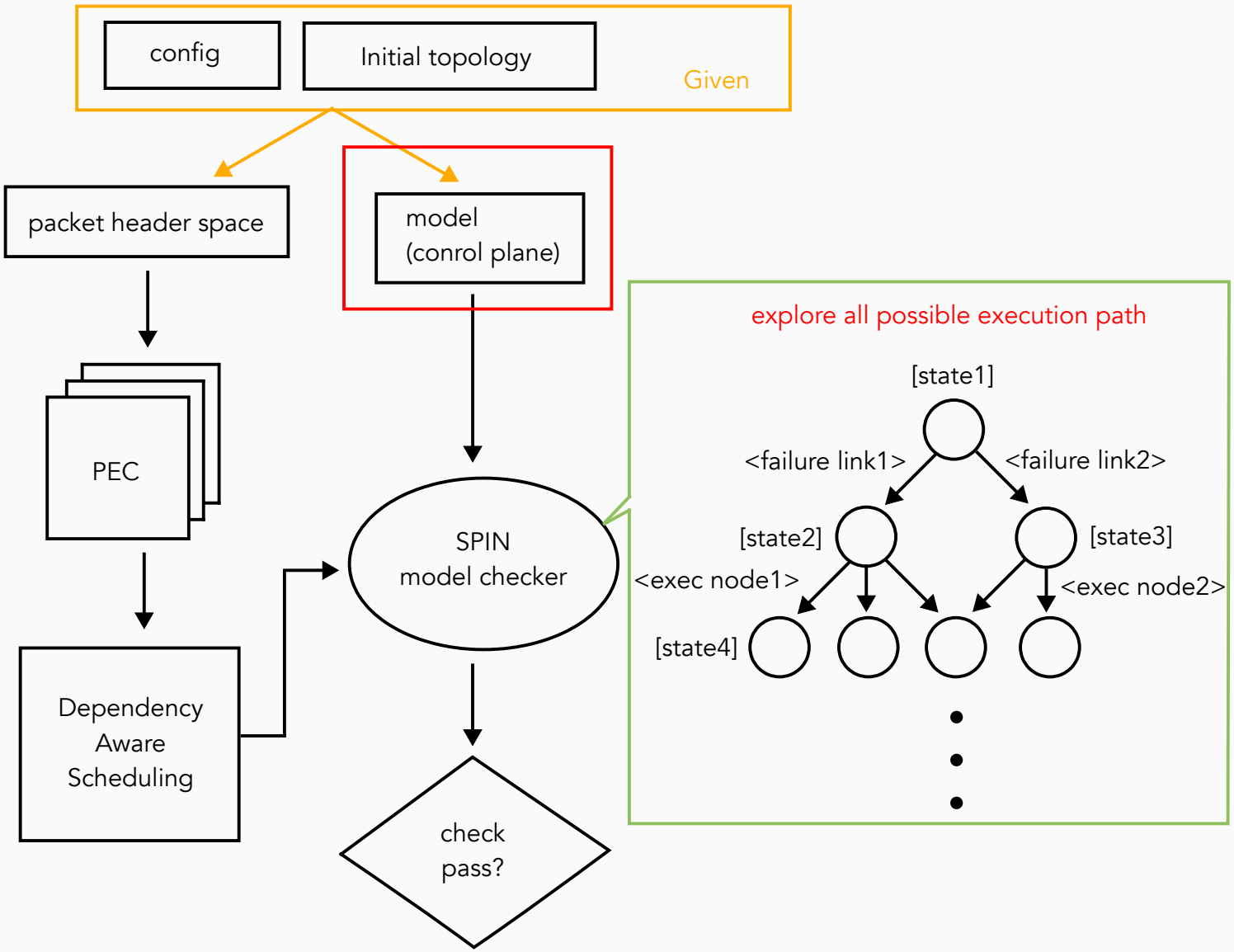
- PEC は基本独立していて、並列計算の観点でもその方が嬉しい
- ただ、例えば iBGP のことを考えてみると、前提として OSPF で dataplane を接続していないといけなかったりする。
- この場合は、iBGP の PEC と OSPF の PEC の間に依存が発生することになる。(Strongly Connected Components)
- SCC 同士の依存、みたいなことも考えられるし考えているらしいが、経験的に SCC はあまり依存しあわないらしい。

Dependency-aware Scheduling

- 実際に model check するときには、極力並列化して実行したい。
- 最初に SCC を確認して、並列化できるところは並列化し、SCC があるところは依存性を保つため同時に検証を行うなどのスケジューリング機構があるらしい
- 動作周りの細かいことが書いていななったため、検証プロセス自体の効率化もされている、くらいの認識。

RPVP protocol model

Plankton Design Overview



RPVP: Reduced Path Vector Protocol

- 前回話した Simple Path Vector Protocol(SPVP) の拡張
- 今回のネットワークが収束状態の時について、モデル検査が正しく実行できるようなモデル
- 実行 path の一部／全部で発散することがある。
- 我々の目的は収束状態でのフォワーディングをチェックするものであるため、ネットワークが収束状態の時について、モデル検査が正しく実行できるようなモデルを定義する
- ちょっとこれは時間的に咀嚼しきれなかった。。。

- RPVP Algorithm

Algorithm 1 RPVP

procedure RPVP(:)
  Init : $\forall n \in N - Origins.\,\text{best-path}(n) \leftarrow \perp$
  Init : $\forall o \in Origins.\,\text{best-path}(o) = \varepsilon$
  while true **do**:
    $E \leftarrow \{n \in N \mid \text{invalid}(n) \vee \exists n' \in \text{peers}(n).\,\text{can-update}_{\text{n}}(n')\}$
    if $E = \emptyset$ **then**:
      break
    end if
    $n \leftarrow \text{nondet-pick}(E)$
    if $\text{invalid}(n)$ **then**
      $\text{best-path}(n) \leftarrow \perp$
    end if
    $U \leftarrow \text{best}(\{n' \in \text{peers}(n) \mid \text{can-update}_{\text{n}}(n')\})$
    $n' \leftarrow \text{nondet-pick}(U)$
    $p \leftarrow \text{import}_{n,n'}(\text{export}_{n',n}(\text{best-path}(n')))$
    $\text{best-path}(n) \leftarrow p$
  end while
end procedure

RPVP - Code Base

- 初期化: Origin 以外の node の best-path を $\perp$ で初期化、Origin は ϵ で初期化
- 収束状態になるまでループ
- 全 node のうち, update するべき対象 node の集合を取り出す。これが空集合の場合に収束状態となりプロトコル停止
- 上記 node 集合のうち無造作に一つ取り出して、もし invalid なら best-path として $\perp$ を入れておく。
- 取り出した node の隣接 node のうち、node をアップデートできるような隣接 node の集合を取り出し、best な node の集合を取り出す
- best な node 集合から無造作に一つ取り出して、その best-path を export -> import して、対象 node の best-path とする
- これを収束状態になるまで繰り返す

RPVP(補足)

invalid

$$invalid(n) \triangleq best\text{-}path(best\text{-}path(n).head) \neq best\text{-}path(n).rest$$

- node n の best-path と、その best-path 上の次の node の best-path を見て、同一経路かどうかを確認している
- 同一経路でない場合、$invalid(n)$ は $true$ になる感じ

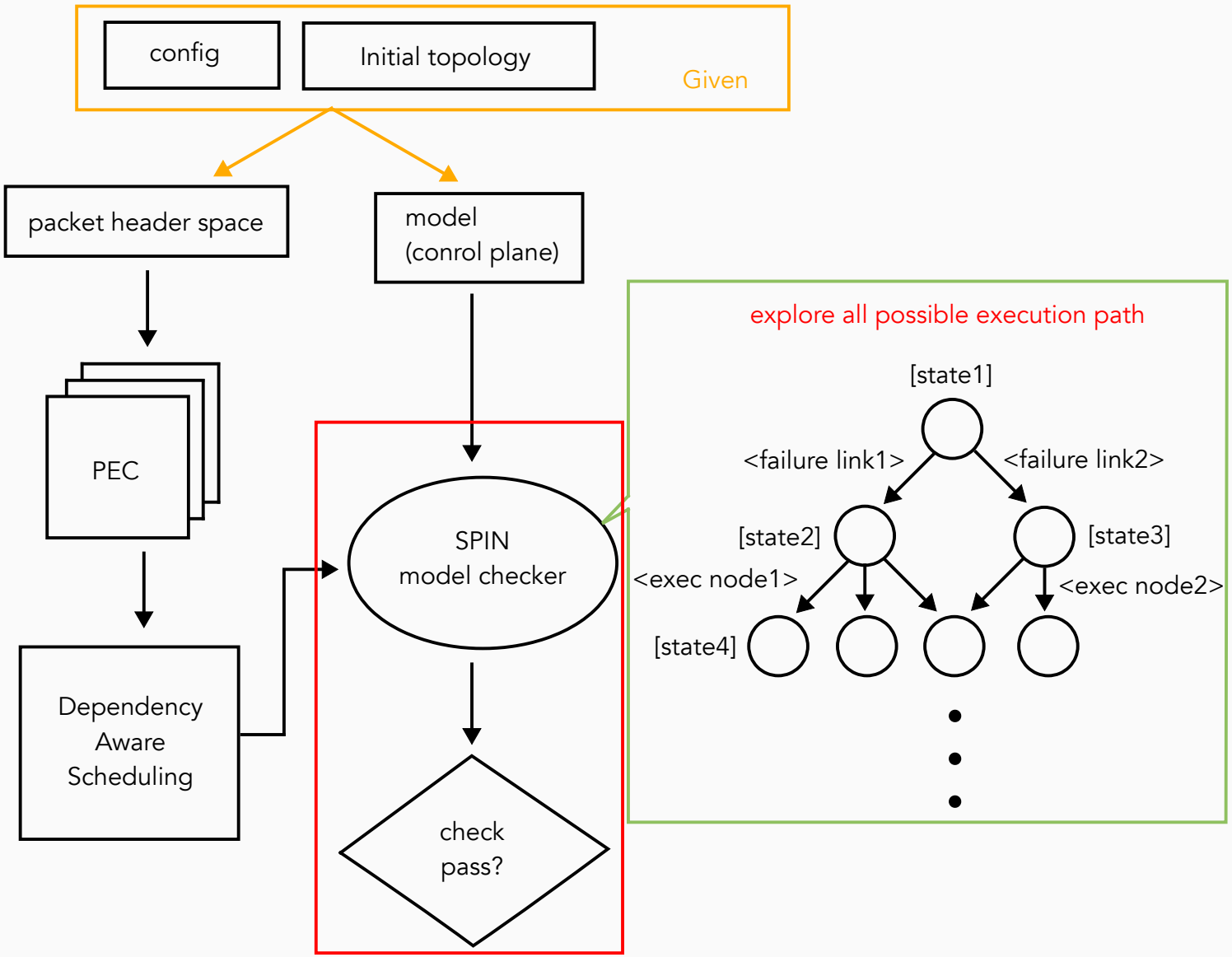
can-update

$$can\text{-}update_n(n') \triangleq better(import_{n,n'}(export_{n',n}(best(n'))), best(n))$$

- node n の隣接 node n' の best(SPVP でやったやつ) を n' から送信し、n が受信したものと、その時点での node n の best を比較し, 前者の方が良ければ $true$ となる感じ

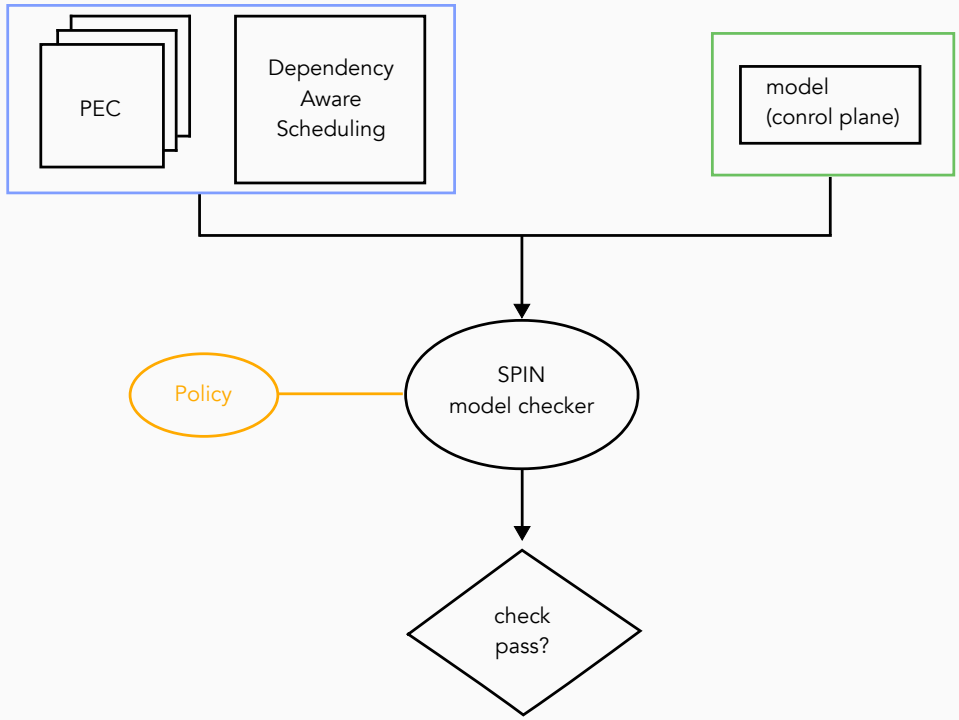
Policies

Plankton Design Overview



Policy

- 今までの話ではしていなかったが、実際は Policy ベースで性質を満たすかどうか検証する
- 以降ではこれについて話をしていく



Policy / Policy API

- ネットワークの収束した状態に対するデータプレーンポリシーの検証をターゲットに考える
- Plankton は Policy API を実装している
- API によって callback を登録する感じ？（多分）で、model checker が収束状態を生成するたびに呼び出される感じ

Policy API

- Plankton API では、Plankton の探索を最適化するための追加情報として、source node / interasting node を与えることができる。
  → 最適化に利用できる (Policy-based Pruning / Choise of Failures)
  → 2 種類のデータプレーンの収束状態の等価 (equivalent) を考えられる

Def: equivalence of two converged data plane states for a PEC

1. source node からの path 長さが同じ
2. 同じ interesting node が path 上の同じ位置にある

Optimization

Optimizations

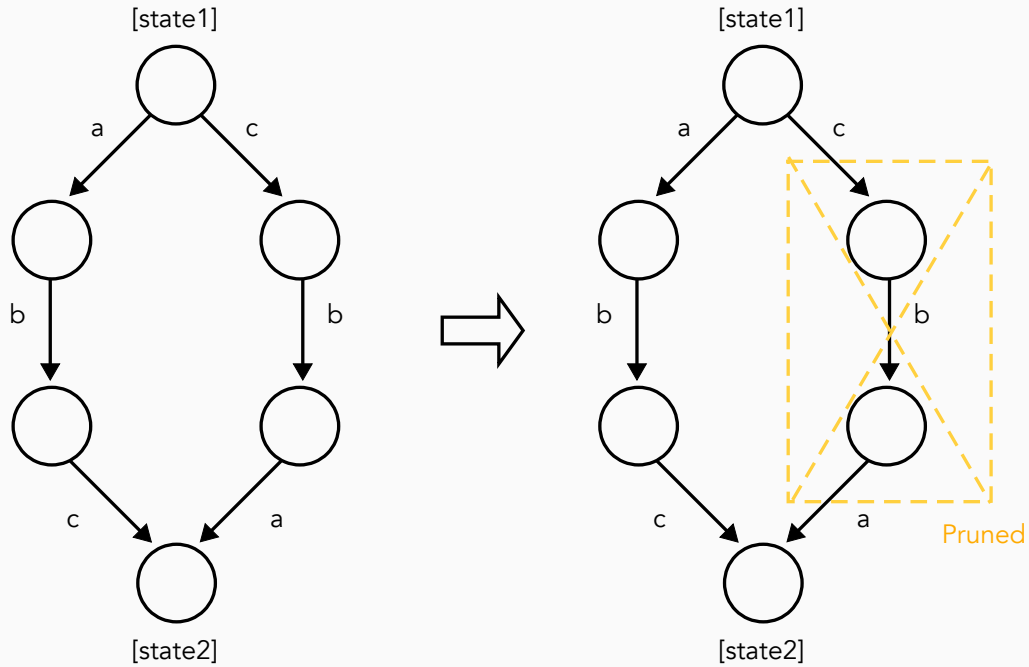
- いくつかの最適化を施している
- ちょっと多いのでいくつかピックアップして話したい
- スライドでは特に Partial Order Reduction の話をしていきたい
  - Partial Order Reduction
  - Policy-based Pruning
  - Choise of Failure
  - State Hashing

Partial Order Reduction

- 状態空間の爆発への対策として重要な技術の一つ
- ある状態を起点にした経路集合から代表経路を決定することで状態数を削減する
- 今回は 4 種類の Partial Order Reduction のインスタンスがある
  - Explore consistent executions only
  - Deterministic nodes
  - Decision independence
  - Failure ordering
- 今回は特に、上２つが丁寧に書かれていたのでそれを見ていく

Partial Order Reduction

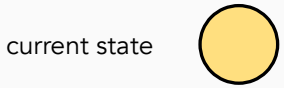
- 同じ結果になるような、複数のアクションが異なるオーダーで実行される場合、一つのオーダのみ考える感じ



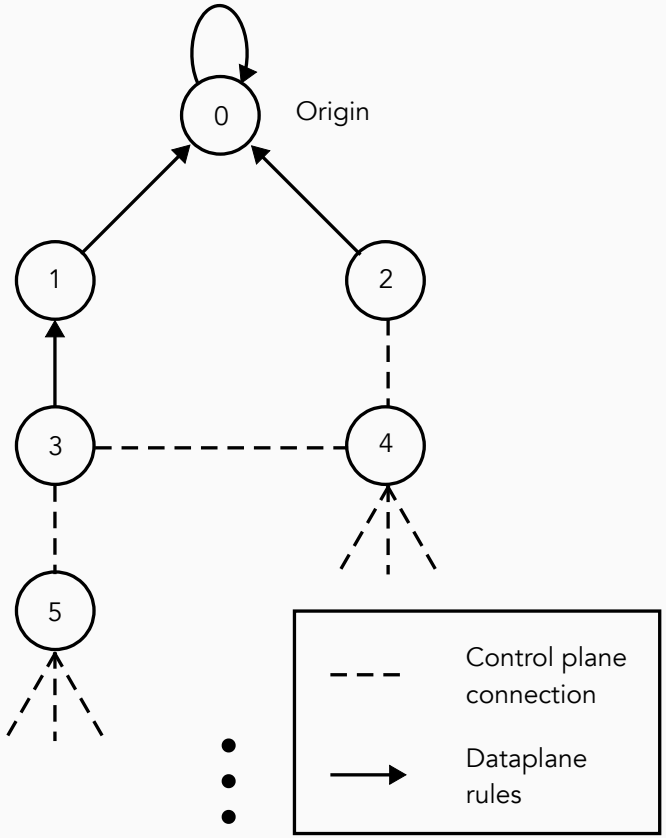
Explore consistent executions only

- consistent execution のみ実行する
- consistent execution とは？
- 収束状態 S, partial execution of RPVP π の時
  実行の各 step において、ある node が S の best-path を pick して、その後それが変更されない場合
- もちろん、探索を開始するときにある収束状態につながる consistent execution を正確に把握することはできないため、チェックの必要がある?
- そこで、我々が探索するすべての実行は relevant であると仮定し、もし反対の証拠を得た時（例えばデバイスが選択されたベストパスを変更する時）その実行の探索を止める

consistent, inconsistent (1)

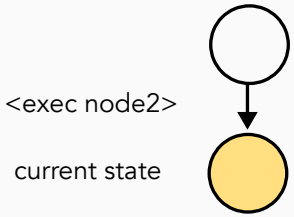


(state diagram)

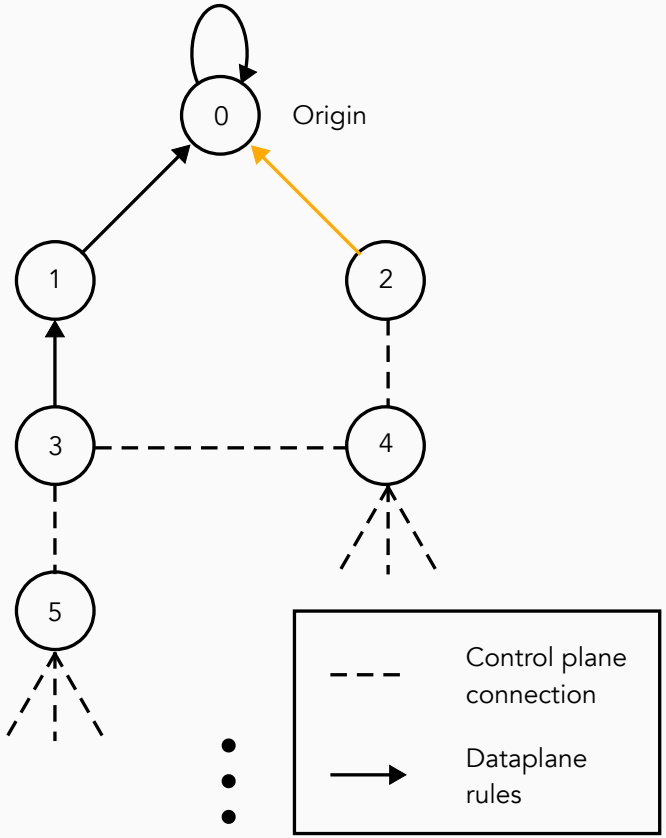


(routing topology)

consistent, inconsistent (2)

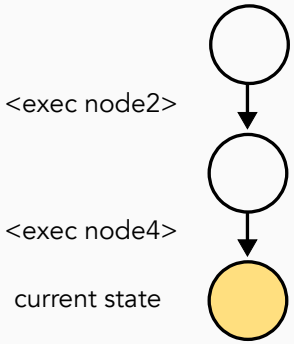


(state diagram)

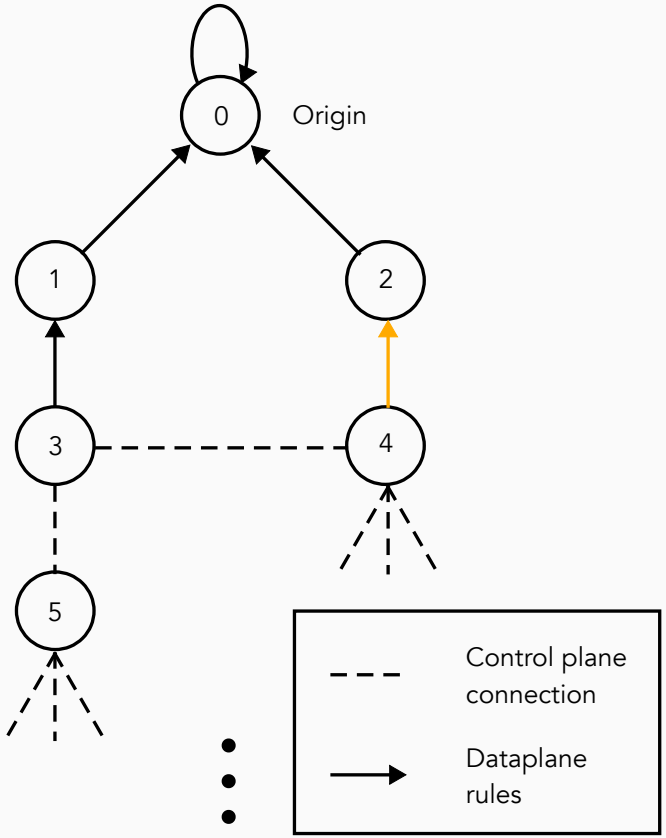


(routing topology)

consistent, inconsistent (3)

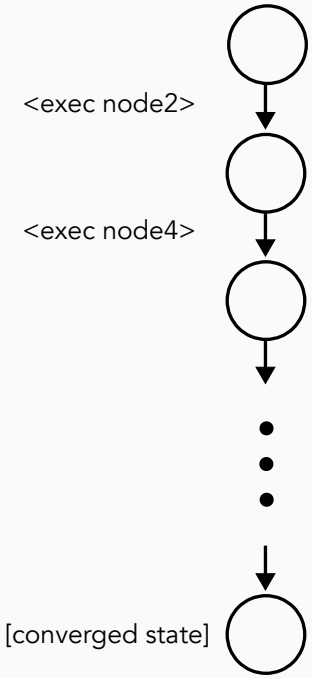


(state diagram)

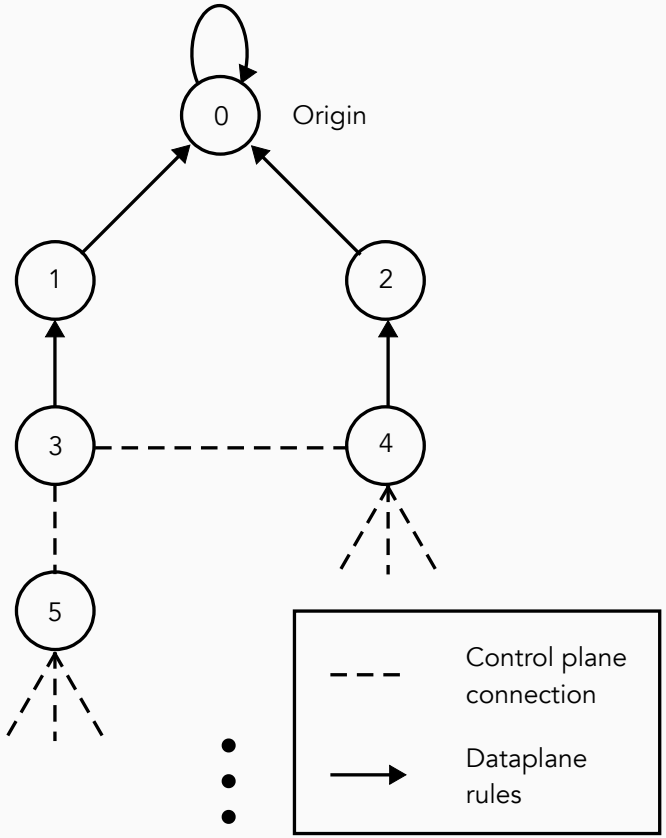


(routing topology)

consistent, inconsistent (4)

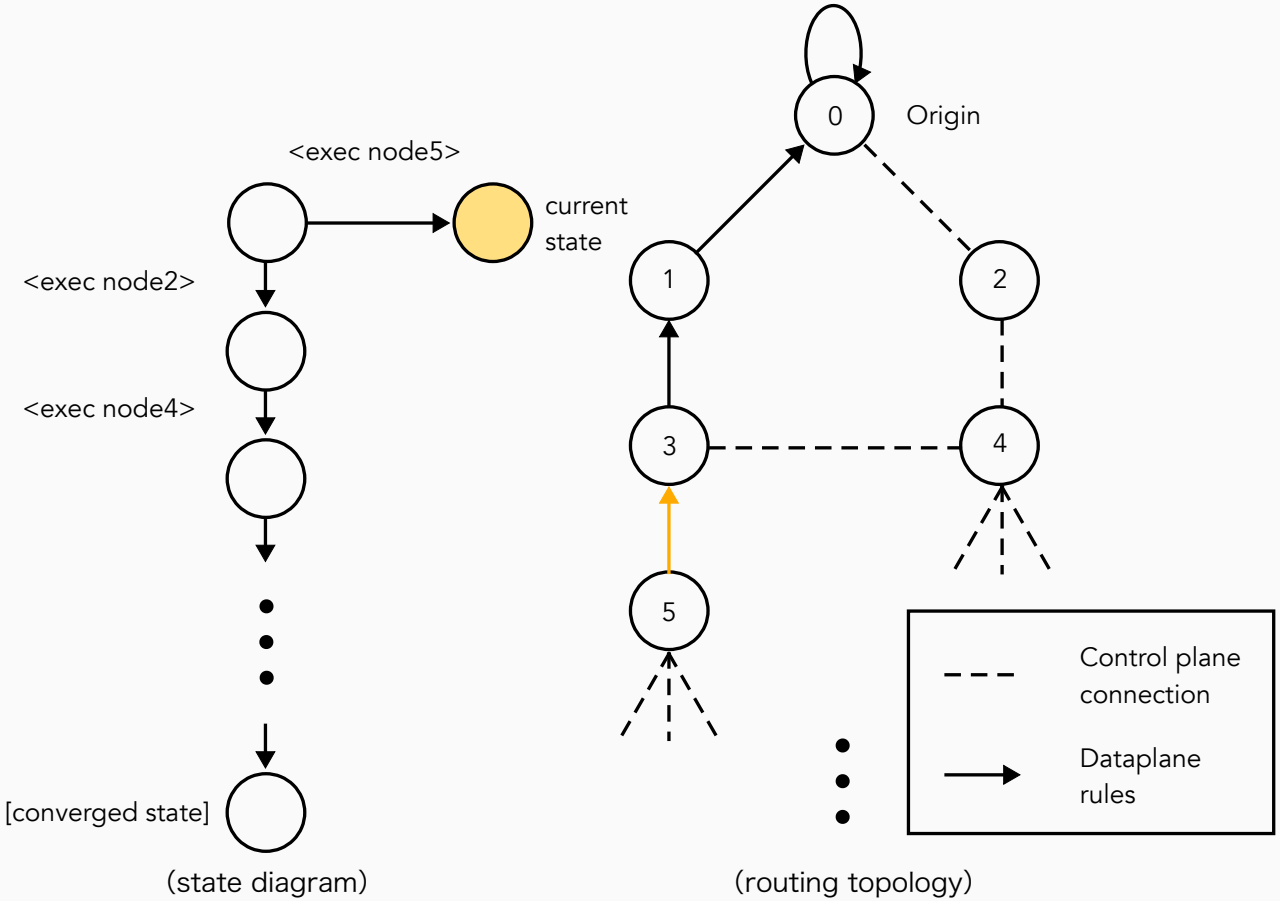


(state diagram)

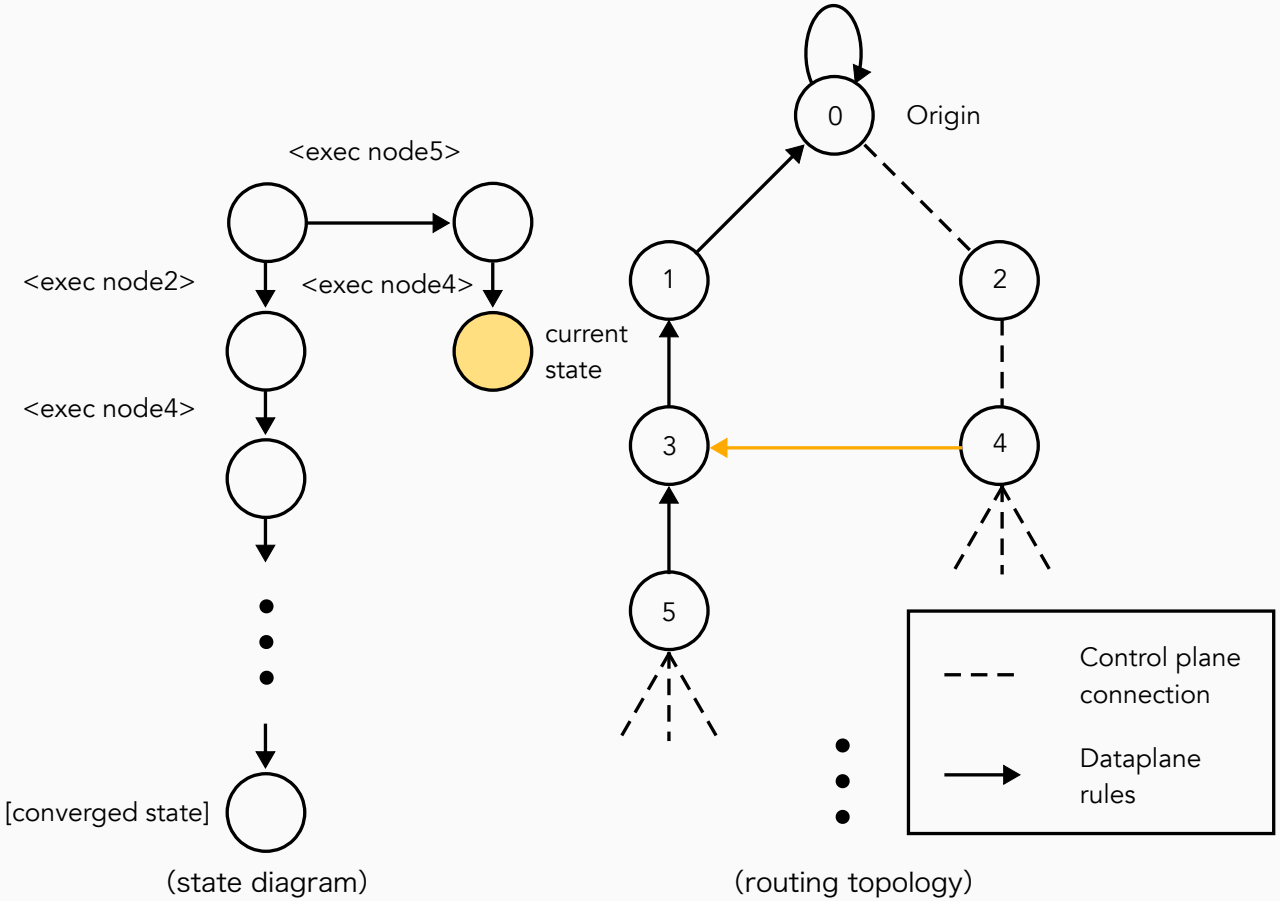


(routing topology)

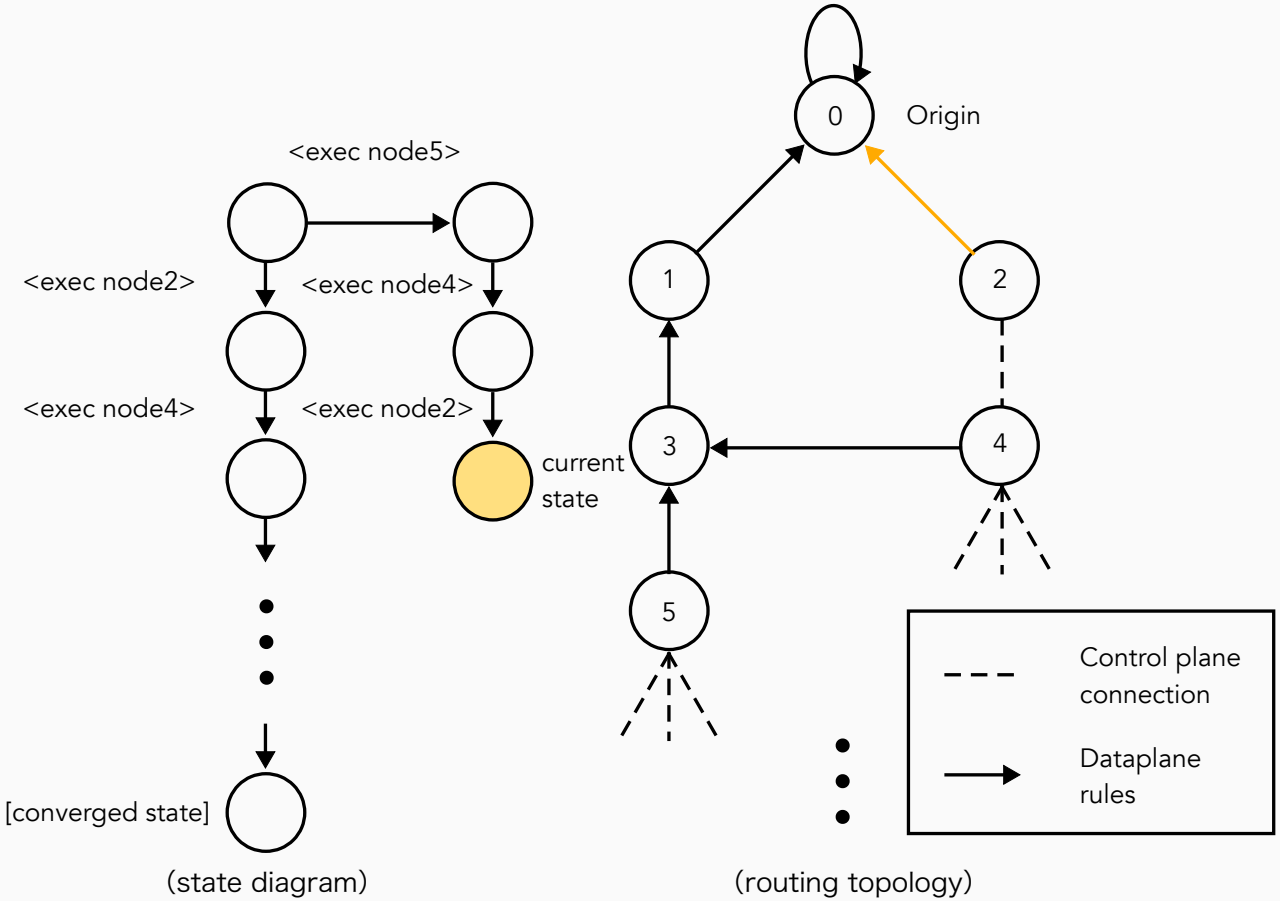
consistent, inconsistent (5)



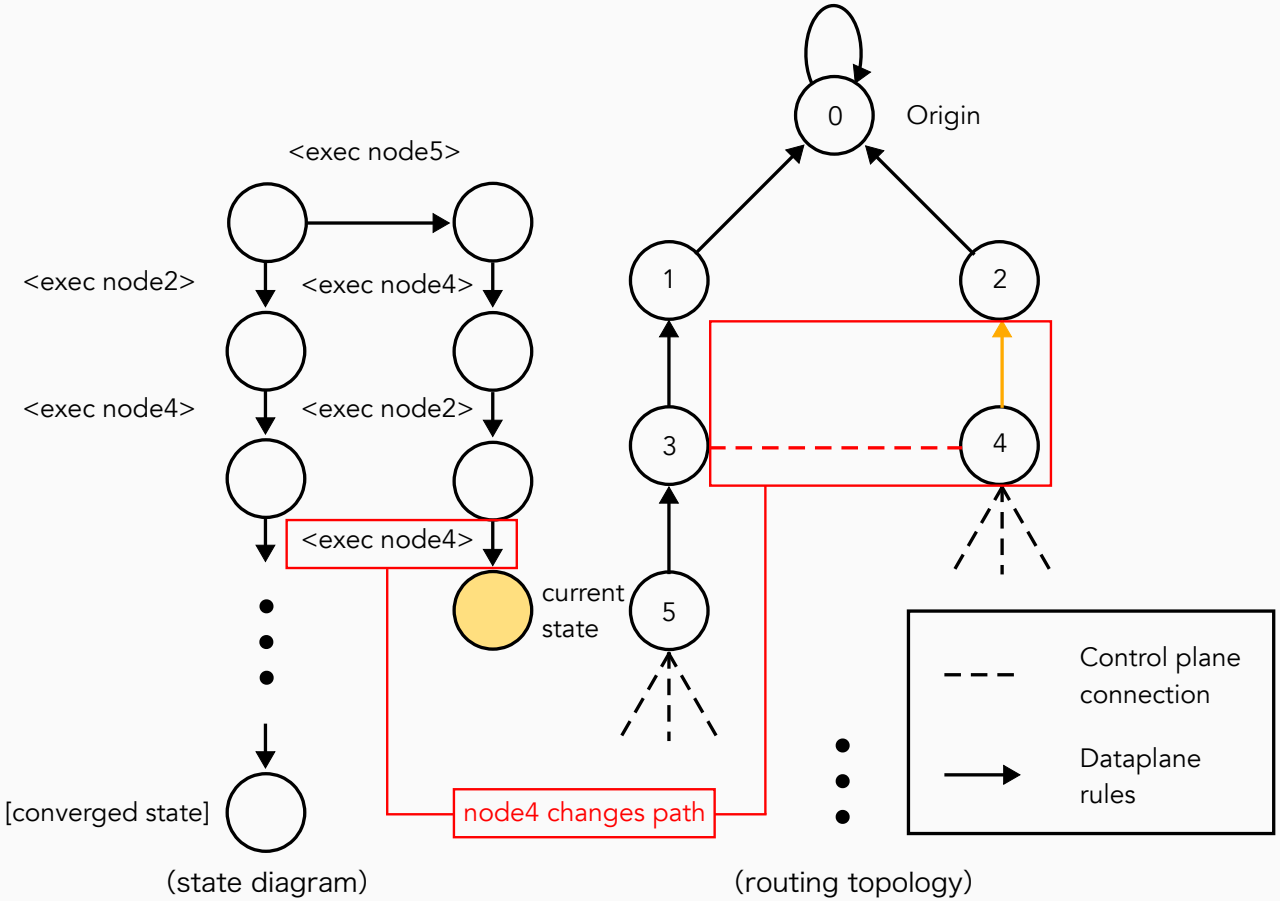
consistent, inconsistent (6)



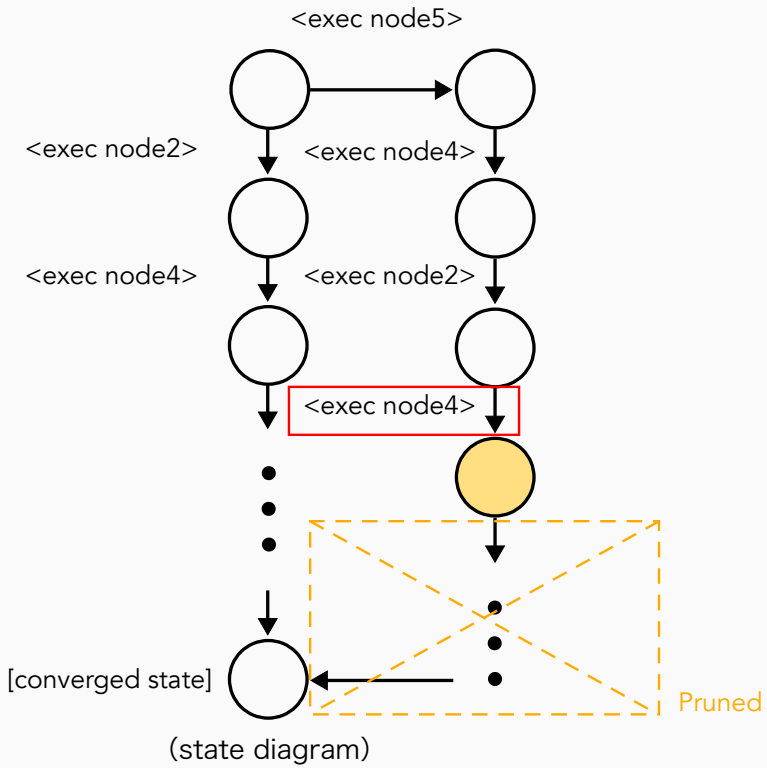
consistent, inconsistent (7)



consistent, inconsistent (8)



consistent, inconsistent (9)

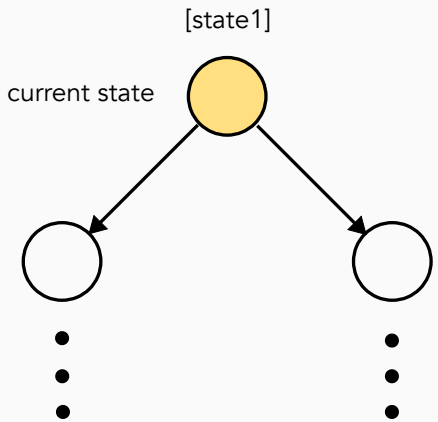


(state diagram)

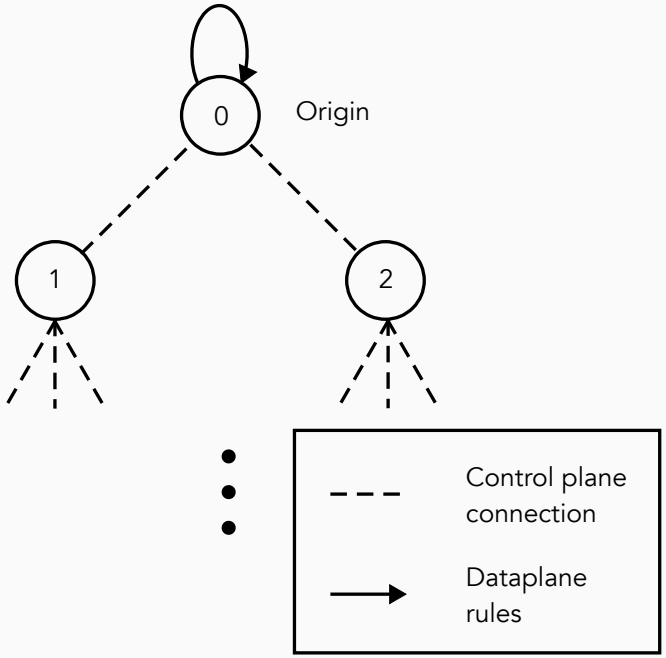
Prioritize deterministic nodes

- ある node が deerministic とは
  そのノードのパス、以降のプロトコル実行の流れの中で絶対に変更されない場合を指す
- configuration から必ず優先されるような node がある場合（その後絶対に path が変更されない場合?）それ以外の選択は考えられない
- Plankton の model checker の探索空間を減らすために、無関係な non-determinism を刈り込む

deterministic nodes

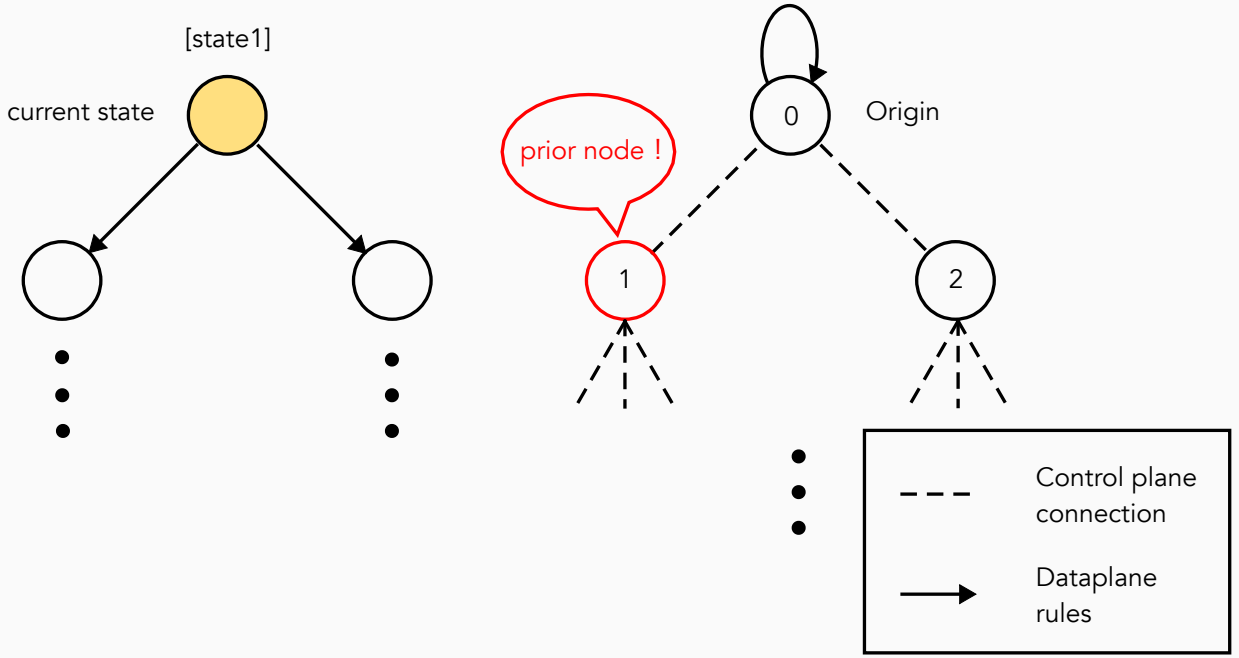


(state diagram)



(routing topology)

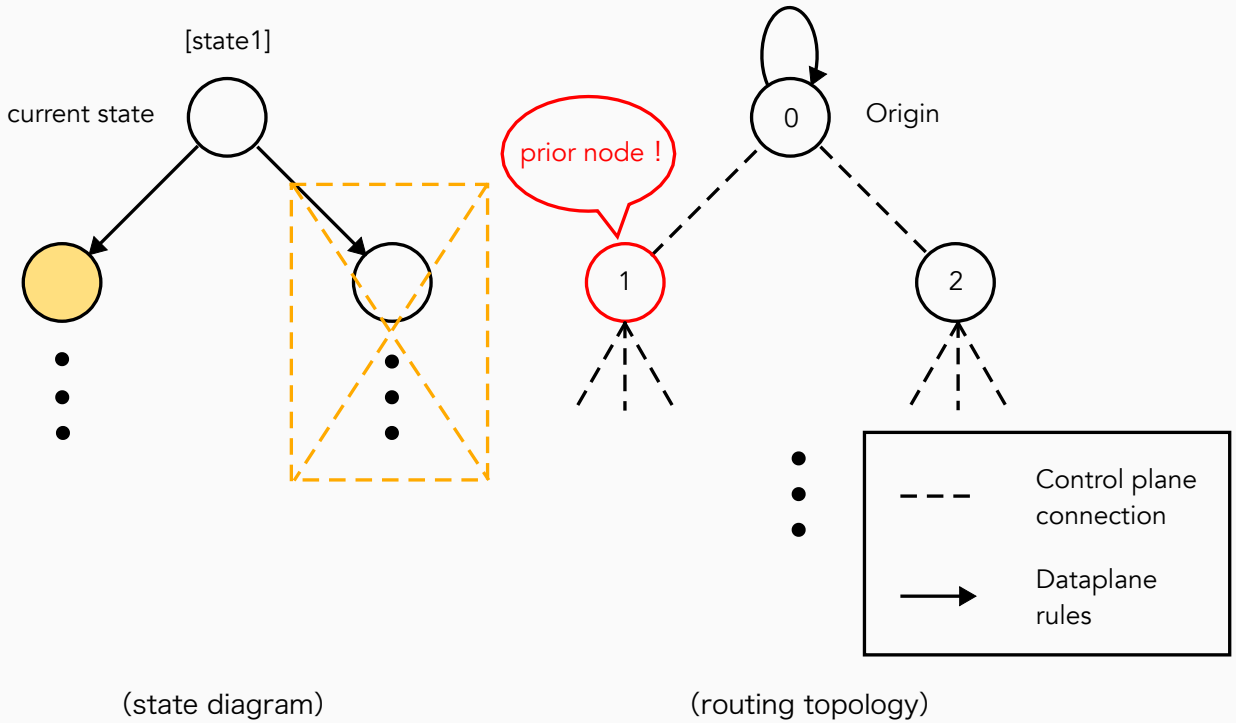
deterministic nodes



(state diagram)

(routing topology)

deterministic nodes



(state diagram)

(routing topology)

Prioritize deterministic nodes

- そもそも partially に protocol が実行されていくのに、どうやって node が deterministic であるかわかるの？
- これは各 Routing Protocol で固有の heuristics を考えるらしい
- ちょっとこれもいまいち理解しきれなかった。。。

Policy-based Pruning

- Policy-based Pruning 概要
- policy-based pruning は、protocol execution に limit をかける
- API で source node の set が定義されている場合、policy はその source node からの forwarding のみを分析することで、policy をチェックする
- 例えばある node への到達性チェックをする場合、source node を与えていると source node からのみ到達性をチェックし問題なければ policy holds

Choise of Failure

- Bonsai が提案した Equivalence Partitioning of Device とか言う話があるらしい。
- 結構ガッツリした論文 + 2018 で最近の物なのでそのうち読みます。

State Hashing

- state 空間を徹底的に探索する間、model checker は同時に膨大な state を追跡する必要がある
- 単一の network state はすべてのデバイスのすべてのプロトコル固有の状態変数のコピーからなる
- そのため、これらの膨大な状態変数のコピーを維持するコストが膨大になる
- 特性として、あるデバイスでルーティングを決定しても、他のデバイスの変数にすぐに影響を与えない
- 具体的な手法がちゃんと書いてなかったが上記の特性からメモリフットプリントを削減する

Evaluation

諸元

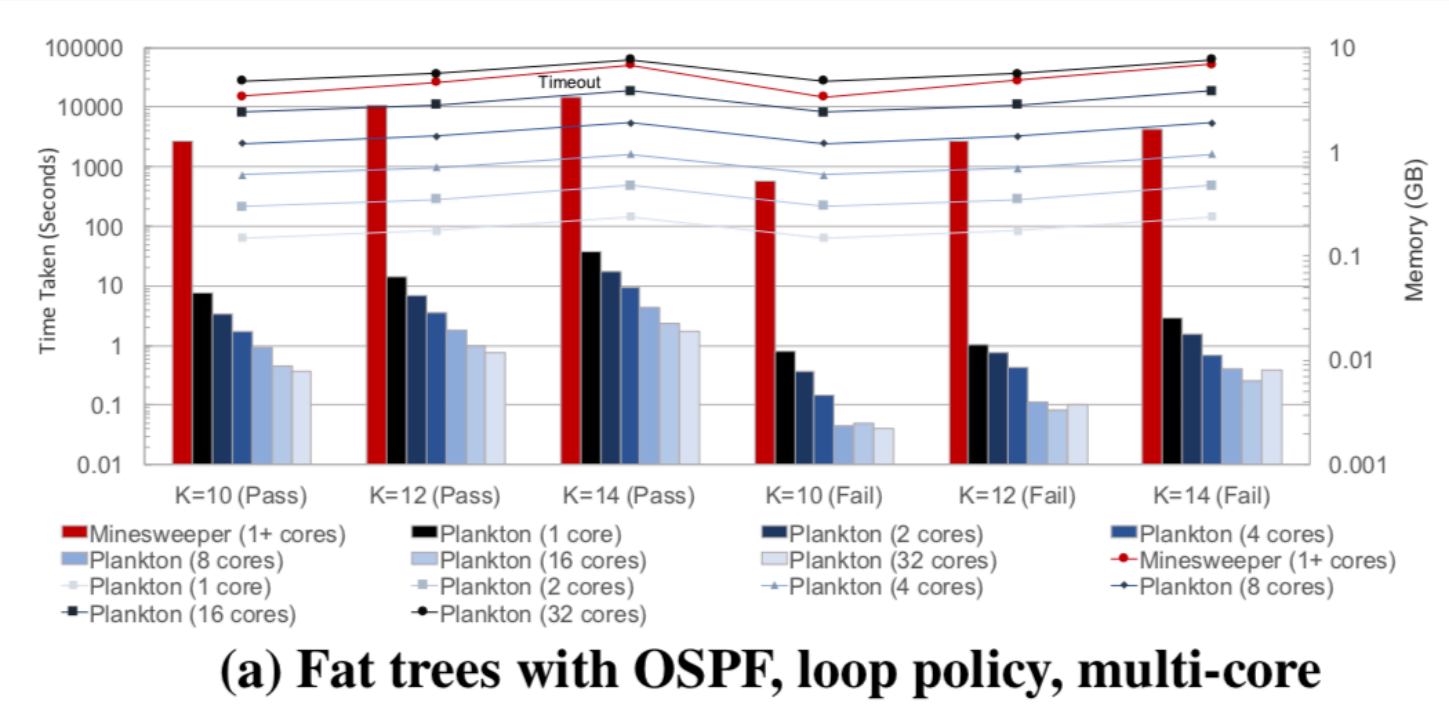
- Plankton の prototype 実装を行った
  (373 Promera Code / 4870 C++ Code)
  - equivalence class computation
  - control plane model
  - policy API
  - optimization
- Ubuntu 16.04
- 3.4 GHz Intel Xeon processor with 32 Hardware thread
- 188G RAM
- 結果が多かったのでいくつか絞って紹介
- 基本的には Minesweeper と比較する。ポリシー検証結果は Minesweeper と Plankton は検証を通じて同じ結果になった。

Experiments with synthetic configurations

- まず、fat tree を利用した performance test をいくつか試した
- 各エッジスイッチが OSPF をしゃべる。リンクの重みは同じ。
- ルーティングループ検出を行ってみる。
- static route を追加し意図的に一部のルーティングでループをおこす。

Experiments with synthetic configurations

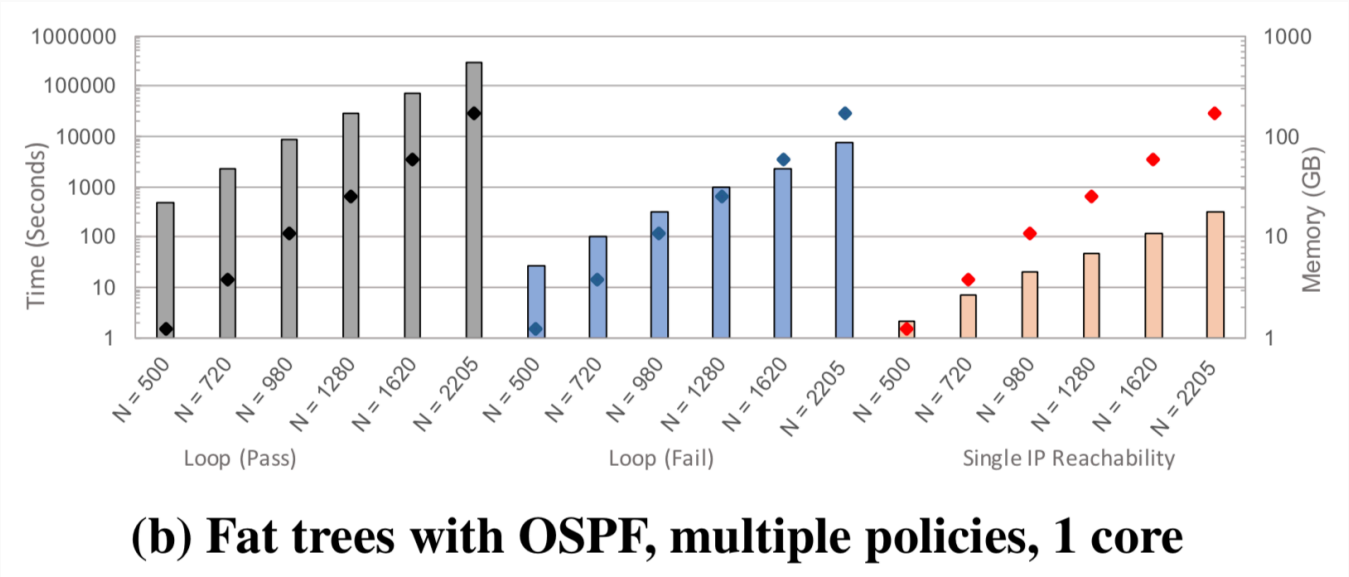
- Plankton の様々なコア数での実行、Minesweeper を使用した場合のグラフ
- 横軸は fat tree のサイズ + ループあり (Pass) とループなし (Fail)
- 時間 (棒グラフ) とメモリ消費量 (折れ線グラフ) を示している
- Plankton は入力ネットワークサイズに応じて現実的な速度で抑えれている
- minesweeper と同一量のメモリを利用する 16core の場合、K=10 では 2000s が 0.5s 程度に



(a) Fat trees with OSPF, loop policy, multi-core

Experiments with synthetic configurations

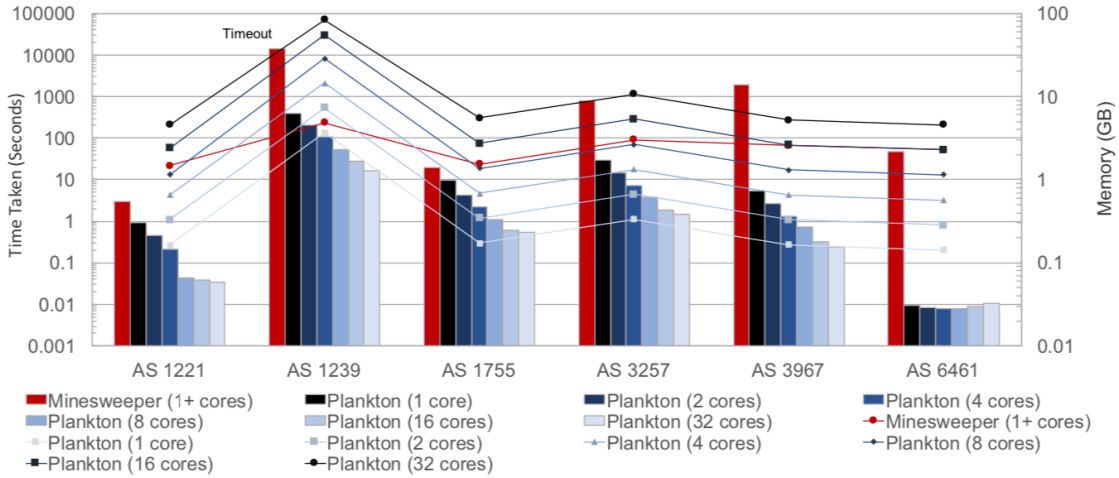
- 更に大きな fat tree までスケールアップしてみたものが下図
  (横軸が Device 数になっているが、fat tree のサイズと見て良さそう)
- 棒グラフは時間、マークは memory
- 最大サイズ（2205-devices）の場合でも、Single IP の到達性は数分程度



(b) Fat trees with OSPF, multiple policies, 1 core

Experiments with semi-synthetic configurations

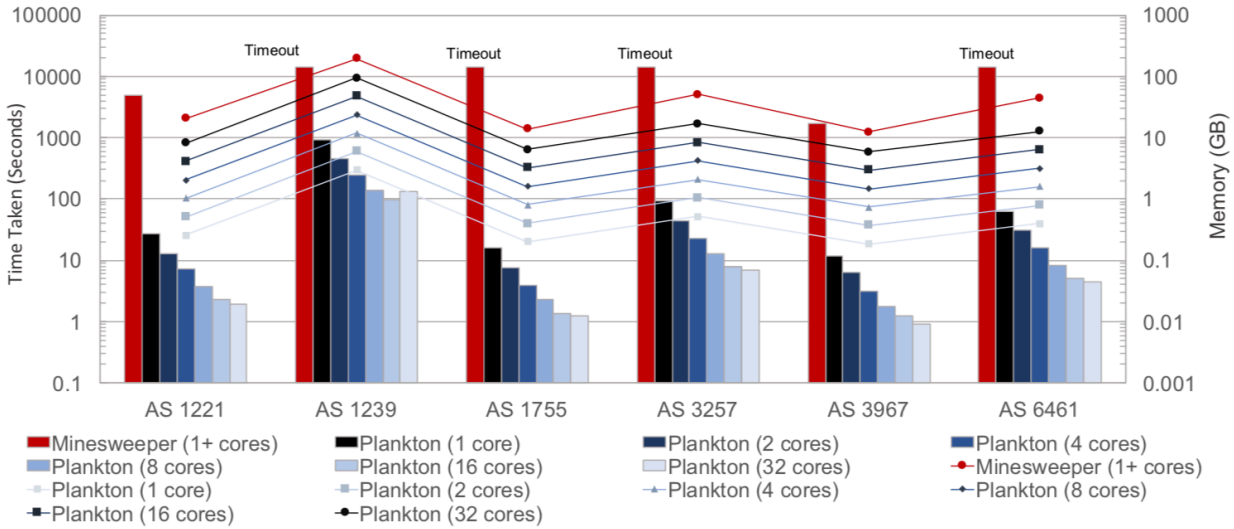
- OSPF によってつながった Real-world AS topology を Rocketfuel から取得
- あるリンク障害が到達可能性に影響を与えるかどうかの測定をした
- 時間とメモリ両方の面で Plankton の方が優れたパフォーマンス



(d) AS topologies with OSPF and failures, reachability policy, multi-core

Experiments with semi-synthetic configurations

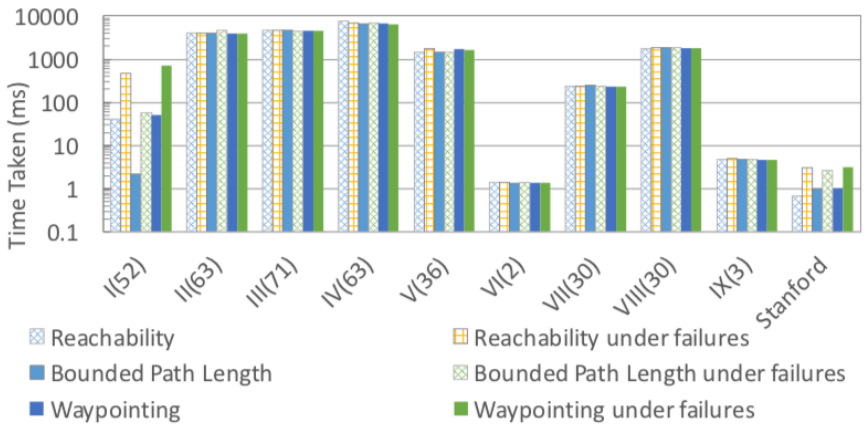
- PEC 間の依存の取り扱いを評価するために、AS トポロジ上で OSPF 上に iBGP を設定した
- iBGP でアナウンスされたプレフィックスに向けられたパケットが正しく配信されているかどうかチェックする
- Dependency-aware scheduler のおかげで、Plankton の方が何桁も優れている



(e) AS topologies with iBGP over OSPF, reachability policy, multi-core

Testing with real configuration

- 3 つの異なる組織の 10 種類の real-world configuration を検証した
- 障害の有無 + 複数の policy で確認した
- 実世界の構成検証の複雑さにも現実的な時間で対応できていることを示している



(h) Real-world configs (number of devices in parentheses), multiple policies, 1 core

Testing with real configuration

- ネットワーク構成としては 3 種類 + Policy3 種類 (link failure ありなし)
- 幅広いポリシーが実世界のネットワーク上で検証できている
- 結果としては Plankton のものしか載せていないが、Minesweeper と比較して優位に良かったようである。

Network	Policy	Links Failed	Memory	Time
II	Loop	0	2.37 GB	8.79 s
		≤ 1	2.47 GB	13.62 s
	Multipath Consistency	0	2.37 GB	16.28 s
		≤ 1	2.37 GB	22.01 s
	Path Consistency	0	2.37 GB	15.43 s
		≤ 1	2.37 GB	23.55 s
III	Loop	0	2.36 GB	11.49 s
		≤ 1	2.88 GB	29.81 s
	Multipath Consistency	0	2.37 GB	16.33 s
		≤ 1	2.67 GB	24.86 s
	Path Consistency	0	2.36 GB	15.53 s
		≤ 1	2.88 GB	21.01 s
IV	Loop	0	2.31 GB	12.37 s
		≤ 1	2.40 GB	13.14 s
	Multipath Consistency	0	2.34 GB	16.36 s
		≤ 1	2.37 GB	17.04 s
	Path Consistency	0	2.33 GB	16.33 s
		≤ 1	2.37 GB	17.00 s

(i) Real-world configs, multiple policies, 32 cores

結論

結論

- Plankton は formal network configuration verification tool
  - equivalence partitioning of the header space
  - explicit state model checking of protocol execution
- 既存の configuration verification tool よりもかなり早い
  - partial order reduction
  - state hashing
- 今後の興味／課題
  - transient state のチェック改善
  - 実ソフトウェアへの組み込み
  - partial order reduction の改善

自戒

- わかっていない点が多い
  - RPVP の議論ともう少し細かいところ
    - RPVP の詳細の議論
    - SPVP の関係やそれを取り巻く証明など
    - これだけで一回話す分位、トピックとしては重そう。。。
  - Optimization
    - Deterministic Node の話（heuristic の話）
    - Choise of Failure (Equivalence Partitioning of Device) の話
    - State Hashing の話（これはどちらかというと SPIN 自体の話に近いかも）
- また時間をおいて色々納得できるところまで読み込めるようになったらまた取り上げたい