# 论文阅读笔记

## 异构网络配置分析与翻译方法的研究与实现

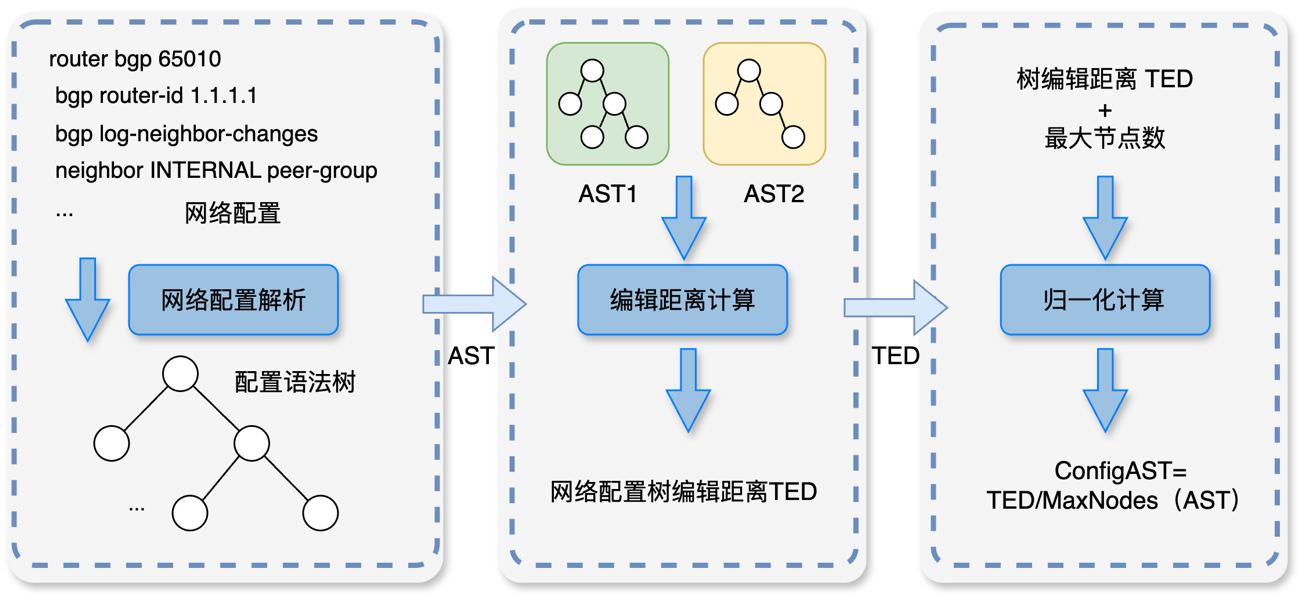
### 1.1 评价指标分类

配置正确性维度：和参考标准答案完全匹配来计算

效率和成本维度：杠杆率以及成本估计等指标，衡量消耗成本

模型效果维度：自然语言处理领域的传统机器学习指标，BLEU、ROUGE等。

### 1.2 基于抽象语法树的网络配置相似度评测指标

**流程：**

1. 解析不同厂商的配置，转化为统一的抽象语法树
2. 基于树编辑距离算法，计算配置语法树之间的差异
3. 通过归一化处理树编辑距离 —— [0, 1]

**实验：**

1. Cisco和Juniper真实网络配置实例，专注BGP和OSPF协议配置的场景
2. Campion作为评估功能一致性的标准
3. 使用Pearson 相关系数评估各个指标与实际网络配置功能一致性间的线性关系

**翻译测评任务定义：**

1. Cisco IOS CLI配置与Juniper Junos结构化配置间的双向自动化翻译
2. 检验生成的翻译是否满足以下要求：

- 语法正确性

- 功能一致性

**评测集的设计与实现：**

1. <https://www.cisco.com/> <https://www.juniper.net/cn/zh/products/routers.html.>
2. 两个维度构建评测集

- 协议维度

- 长度维度

**进一步研究计划：**

1. 扩展网络配置翻译评测集

- 扩展协议：BGP、OSPF、ACL、NAT等

- 扩展厂商：Cisco、Juniper、Huawei以及诺基亚等

1. 结合SFT与RAG的网络配置翻译
2. 单个配置文件（配置片段）翻译 ——> 全网配置迁移场景(结合SMT求解器进行全网配置意图的挖掘与参数求解)
3. 结合 <https://arxiv.org/abs/2504.10046> （CodeRAG）论文方法提升翻译效率（晨阳学姐推荐）

### 1.3 思考-扩展

在大模型实现程序翻译中，比如 Java ——> Python ，是如何评估翻译效果的呢？使用了什么指标？

**指标对比表：**

|  |  |  |  |
| --- | --- | --- | --- |
| 指标名称 | 解释 | 缺点 | 优点 |
| BLEU  （Bilingual Evaluation Understudy） | 衡量生成代码与参考代码在词序列 n-gram 匹配度上的相似性 | 简单易用 | 结构上不同但语义等价的代码可能会被误判为“差” |
| CodeBLEU | 在BLEU基础上引入语法树匹配、  数据流匹配、  关键字重叠等 |  |  |
| Edit Distance | 计算生成代码与参考代码之间的编辑距离（如 Levenshtein 距离），衡量修改成本 | 编辑距离仅关注字符差异，无法捕捉语法树或逻辑等价性。 |  |
| EM  （Exact Match Accuracy） | 判断翻译结果是否与参考答案完全一致 | 明确、可度量 | 对变量名差异、顺序变换不够鲁棒 |
| Execution Accuracy / Functional Correctness | 将生成的目标语言代码 运行并测试，看其是否能通过与源代码相同语义的测试用例 | 操作复杂度高 | 是目前最有效也最贴近实际应用的评估方法 |
| Tree Edit Distance / AST 相似度 | 将代码解析为抽象语法树（AST），计算结构差异 | 能反映结构层面的等价性 | 实现复杂，需良好语法解析器支持 |
| Syntax Error Rate | 统计生成代码中无法通过编译器/解释器解析的比例。 |  |  |
| Human Evaluation  （一般作为辅助手段，用于对自动指标效果进行验证） | 由有经验的程序员评估代码翻译质量：可读性、功能性、风格一致性等 |  |  |

**CodeBLEU**:

计算方法如下：

取以下四者的加权平均：N-gram 匹配（BLEU）、抽象语法树（AST）匹配、数据流（Data-flow）匹配、关键字（Keyword）匹配

CodeXGLUE

论文及开源代码：

<https://arxiv.org/abs/2102.04664>

<https://github.com/microsoft/CodeXGLUE>

## 基于大模型和图神经网络的配置示例泛化合成系统

### 1.1 背景

现有配置合成系统仍高度依赖人工参与

- NetComplete：要求运维人员手动起草并提供配置模板

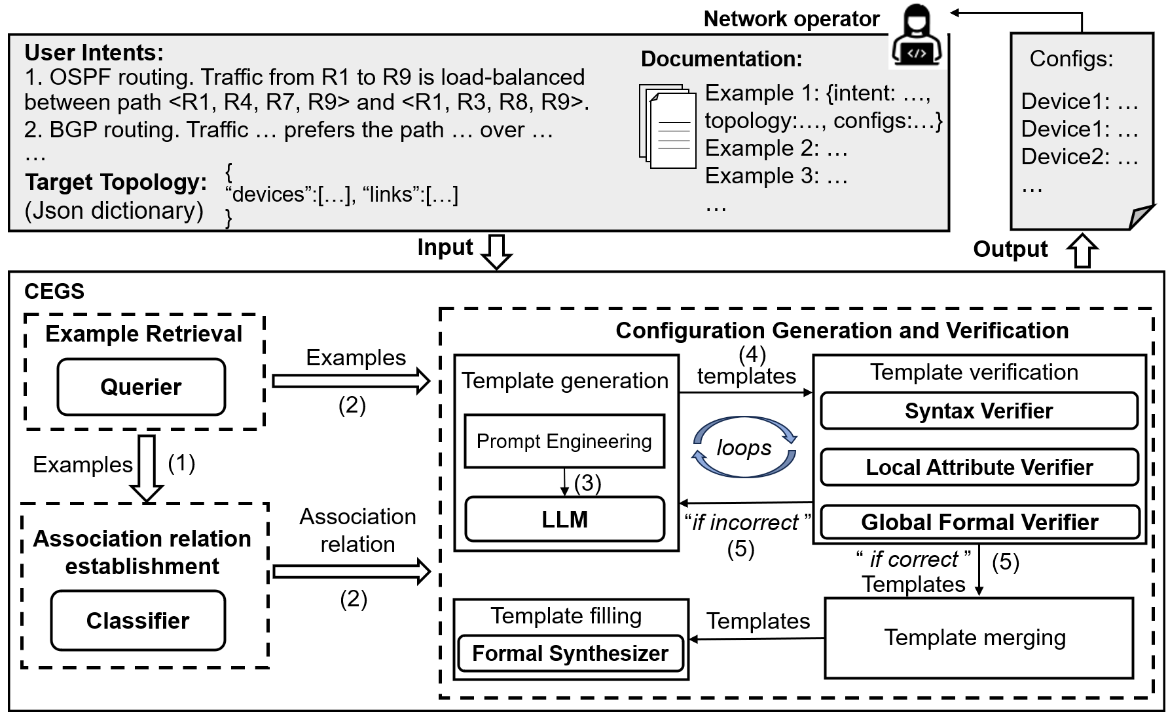
- SyNet、Aura：要求运维人员编写这些系统自定义的DSL

原因：缺乏配置示例遵循和泛化（EFG）

实现EFG需要：

1. 示例拓扑和目标拓扑的映射和理解能力
2. 自然语言意图的解析能力
3. 设备配置代码的生成能力

### 1.2 本文工作

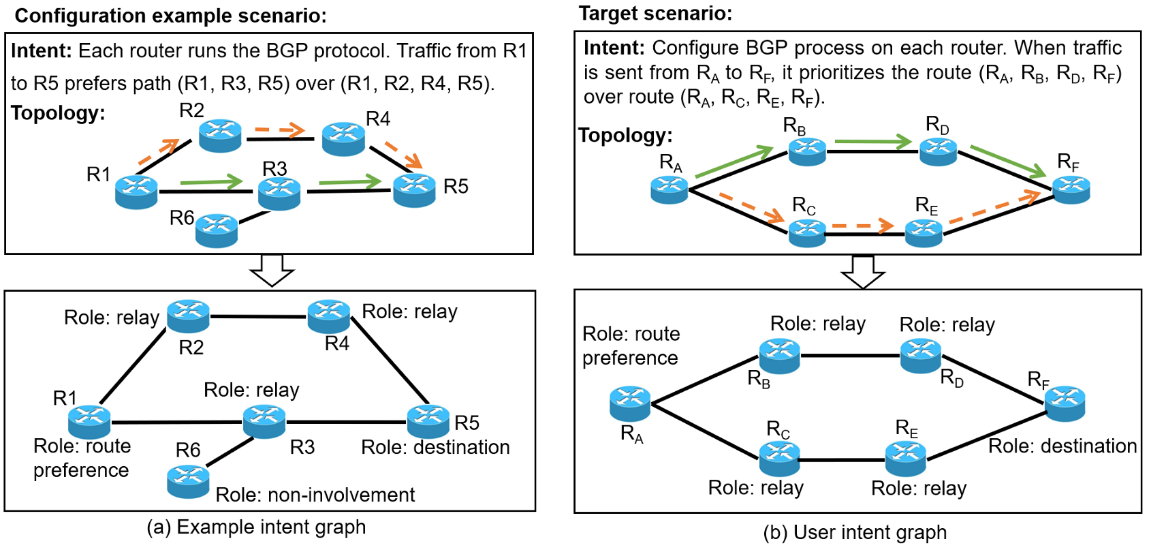
CEGS三阶段：

1. 示例检索
2. 设备关联
3. 配置生成和验证

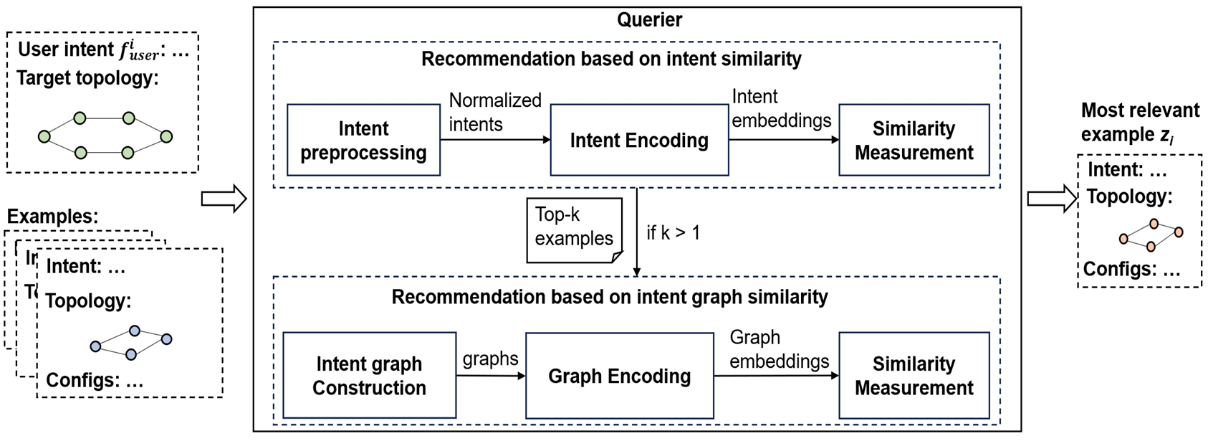
**1.2.1 示例检索**

1. 示例意图与用户意图的语义相似
2. 示例拓扑与目标拓扑的结构相似

意图图（intent graph）：在拓扑节点中引入"Role"属性，将意图嵌入网络拓扑

加速配置示例检索过程：两阶段推荐策略

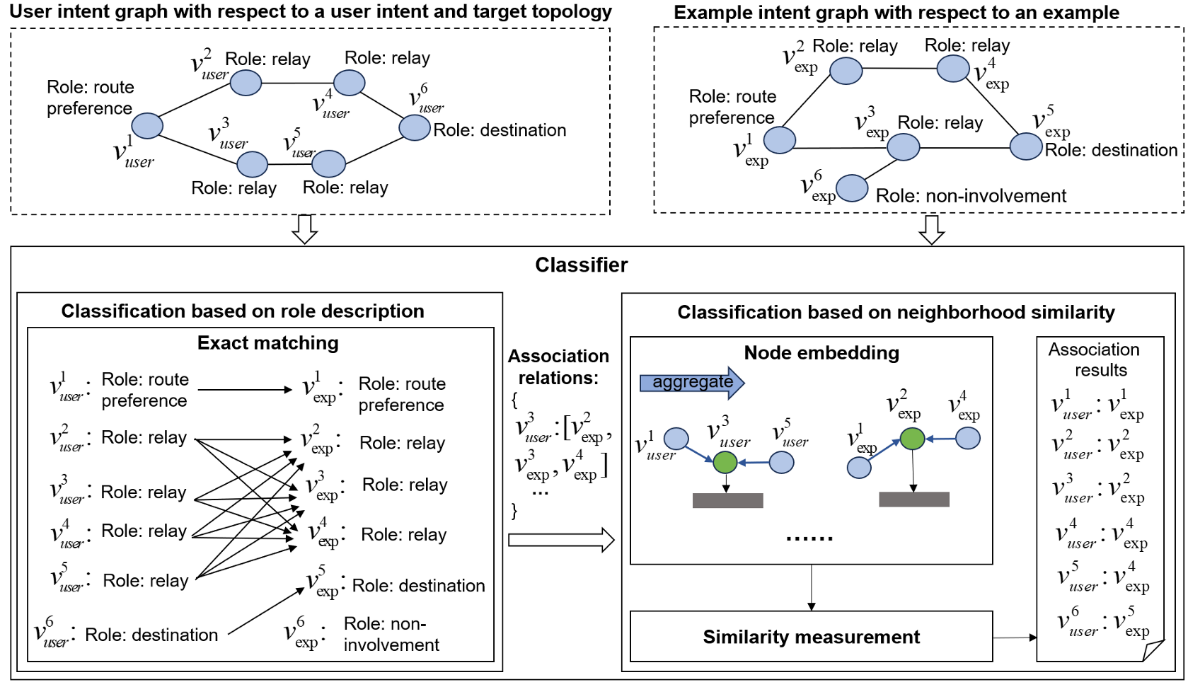
1. Querier根据示例意图与用户意图间语义的相似性，从设备文档中过滤掉与目标场景无关的配置示例。
2. Querier采用GNN算法根据示例意图图与用户意图图间的相似性推荐最佳配置示例。

**1.2.2 设备关联**

分类器：

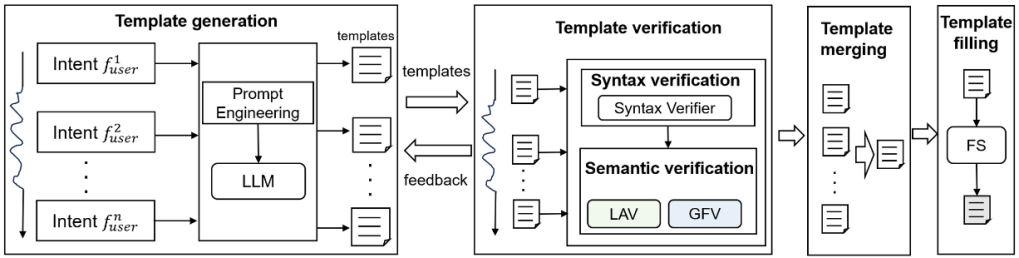
Classifier首先基于意图图中定义的"Role"属性的精确匹配来对节点进行关联。

解决多匹配问题：一个节点的配置也受其邻居节点影响。因此，Classifier进一步采用GNN算法通过同时评估节点属性和邻域相似性建立节点间的唯一关联关系。

**1.2.3 配置生成和验证**

任务：根据Querier检索到的配置示例和Classifier构建的设备关联关系，生成满足目标场景中用户意图的网络配置。

LLM驱动的高效合成方法：

1. 利用LLM根据配置示例和设备间的关联关系，生成目标场景中每个设备的配置模板，将网络策略参数的具体值保留为符号变量
2. 利用一个语法验证器（Syntax Verifier）和两个语义验证器(LAV, GFV)分别从语法结构和语义逻辑对LLM生成的配置模板进行正确性验证
3. 纠错直到LLM正确地生成所有设备的配置模板
4. CEGS利用一个形式化工具根据全网信息填充模板中的策略参数

### 1.3 实验结果

**1.3.1 实验数据**

Internet Topology Zoo数据集中选取了20个真实ISP网络拓扑，规模涵盖20至754台路由器。整合三个区域拓扑，构建了一个包含1094台路由器的更大规模拓扑。

3种协议：

- 静态路由

- OSPF

- BGP

6类意图：

- 静态路由

- 负载均衡

- 特定路径选择

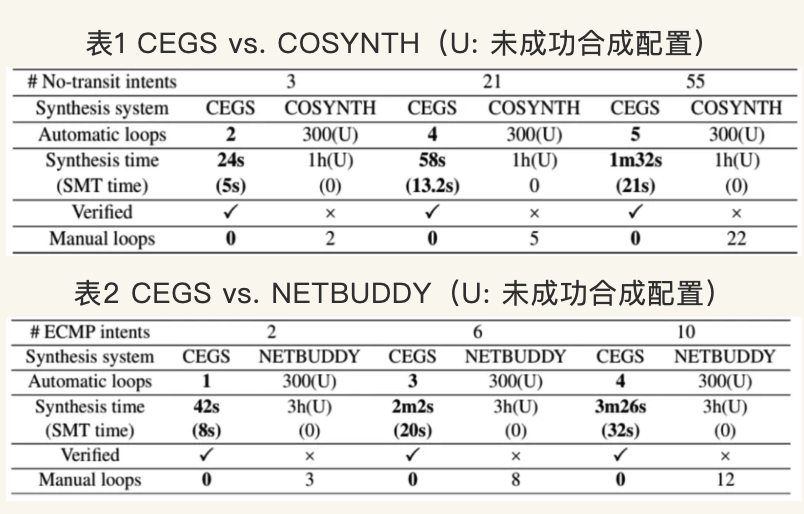
- 路径偏好

- 数据隔离

每种意图均定义了特定源-目的节点对之间的数据转发策略

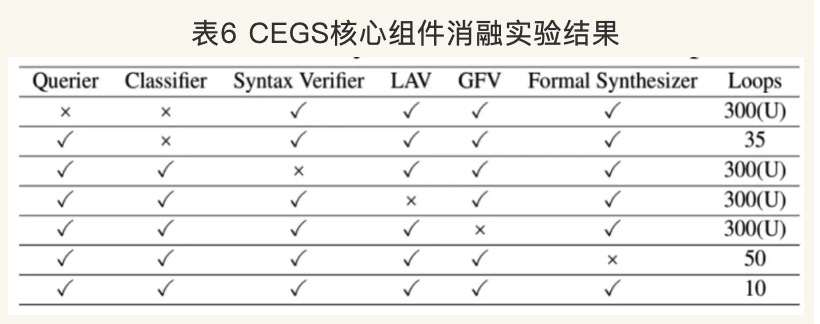
**1.3.2 实验结果**

vs COSYNTH&NETBUDDY



**1.3.3 消融实验**

去掉各核心组件对结果的影响

1.4 思考-扩展

传统的RAG：通过自有垂域数据库检索相关信息，然后合并成为提示模板，给大模型生成漂亮的回答。

RAG：数据准备、数据检索、LLM生成

数据准备阶段：数据提取——>文本分割——>向量化（embedding）——>数据入库

应用阶段：用户提问——>数据检索（召回）——>注入Prompt——>LLM生成答案

我理解本文是 做了一个 GNN 的 RAG，为传统的 RAG 做了网络领域的定制功能。

## Campion: Debugging Router Configuration Differences

1.1 摘要

检查两个配置是否是 behaviorally equivalent.

Network manageability Network Verification, Equivalence Checking, Error Localization, Modular Reasoning

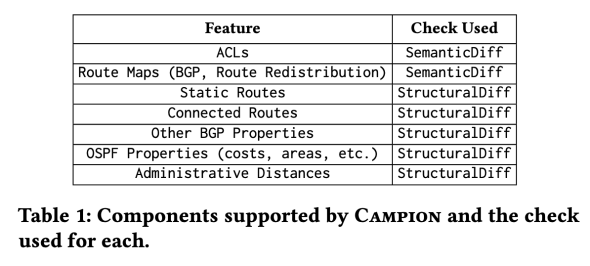
1.2 Introduction

**任务：**

1. 路由器备份的配置更新（备用路由器）
2. 路由器替换的配置更新

minesweeper可以用于验证路由器配置的等价性，但是对调试错误的帮助微乎其微

Campion 不是将路由器行为单一表示（SMT），而是分别比较两个配置之间的对应组件对，然后报告行为上不等价的组件。

 **两种差异：**

1. 结构等价性检查：任何结构上的差异都意味着存在行为上的差异。
2. 语义等价性检查：结构截然不同但行为相同。—— 将组件建模为函数，检查网络控制平面结构（路由器图）等价性

HeaderLocalize：将差异定位到输入空间 BDD（二进制决策图）+ 原始配置 ——> 生成BDD中所有目标IP地址的代表

证明：Campion 是无协议的，不对协议建模。每对配置组件的等价性意味着协议在路由器行为上是完全相同的。

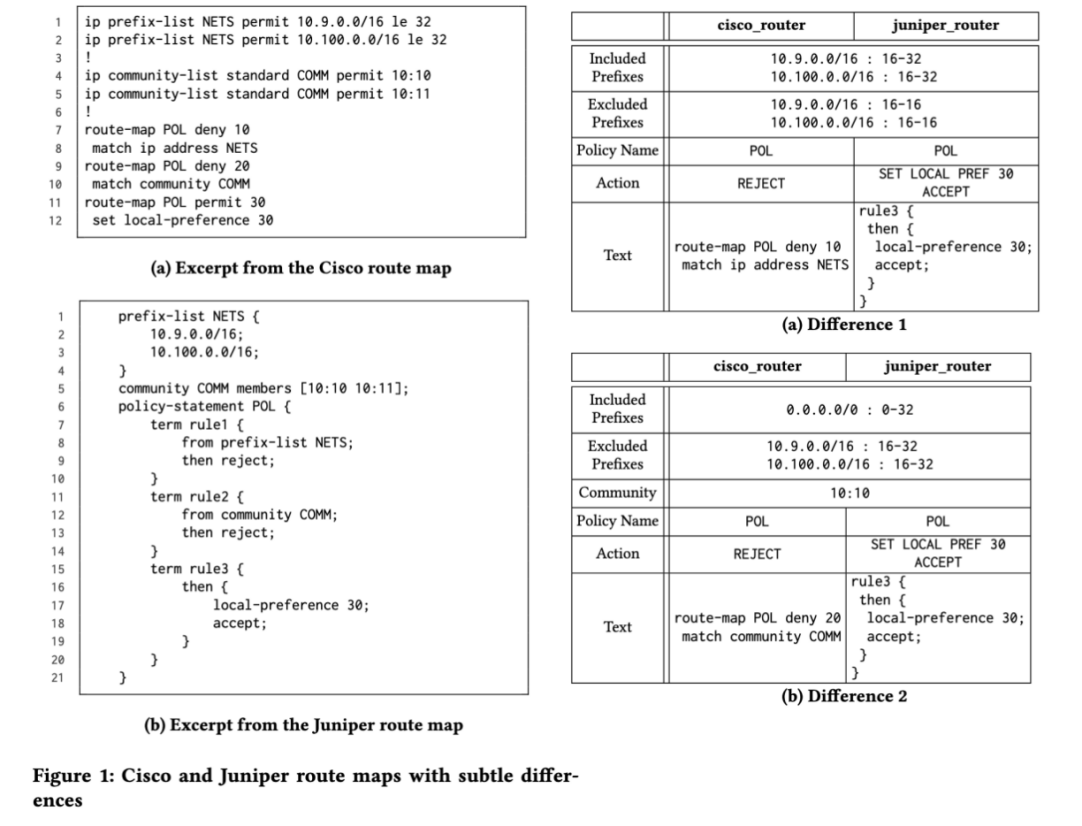
潜在缺点：误报 —— 两个配置组件导致某些配置行为差异，但加入路由器配置上下文后，行为仍然等价 —— 实验表明误报很少发生

**本文贡献：**

1. 方法可识别两种配置间的所有行为差异，并将差异定位回相关配置上，提出了SemanticDiff、StructuralDiff、HeaderLocalize
2. 一个定理证明模块化等价方法
3. Campion 工具 ： 支持所有 Minesweeper 支持的 routing and forwarding components
4. <https://github.com/atang42/batfish/tree/rm-localize>

1.3 Campion by Example

**1.3.1 Route Map Diffs via Semantic Checks**



- a prefix list NETS

- a community list COMM

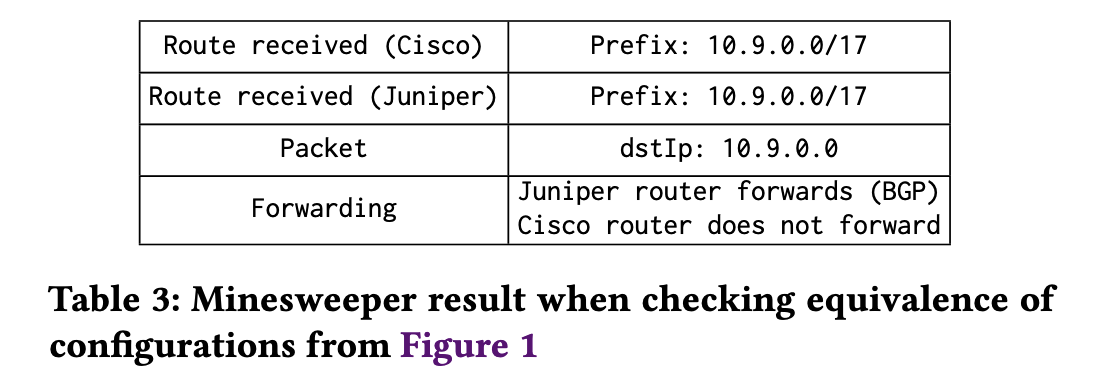
header localization：identifying and representing all problematic inputs

text localization：identifying all relevant lines of the configuration

**Minesweeper的结果：**

通过检查两个路由器整个配置的逻辑表示是否等价实现，难以判断错误来源。

1. 只提供了单一行为差异的信息，但是实际上保护两个不相关的配置差异
2. 对Minesweeper识别出的错误，只提供了一个具体的例子，无法指出是配置只的那部分导致的行为差异

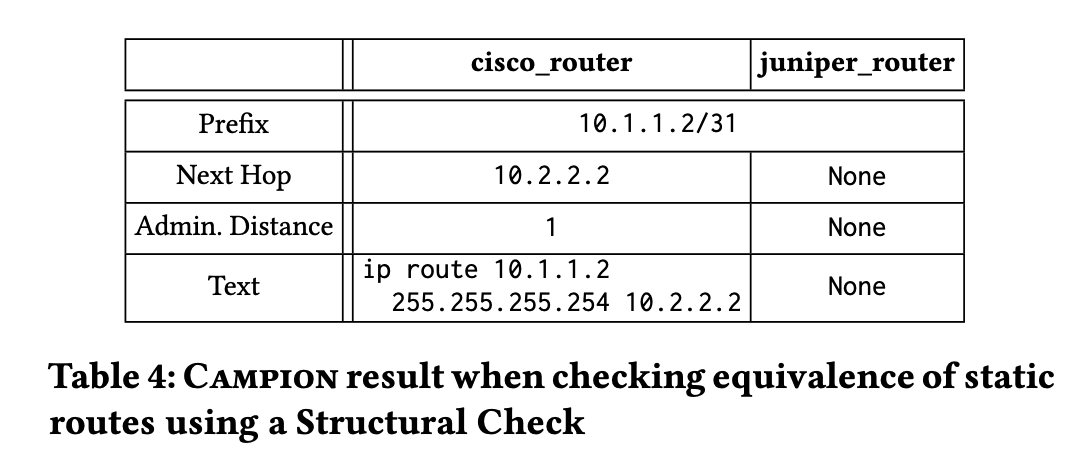


**1.3.2 Static Route Diffs via Structural Checks**

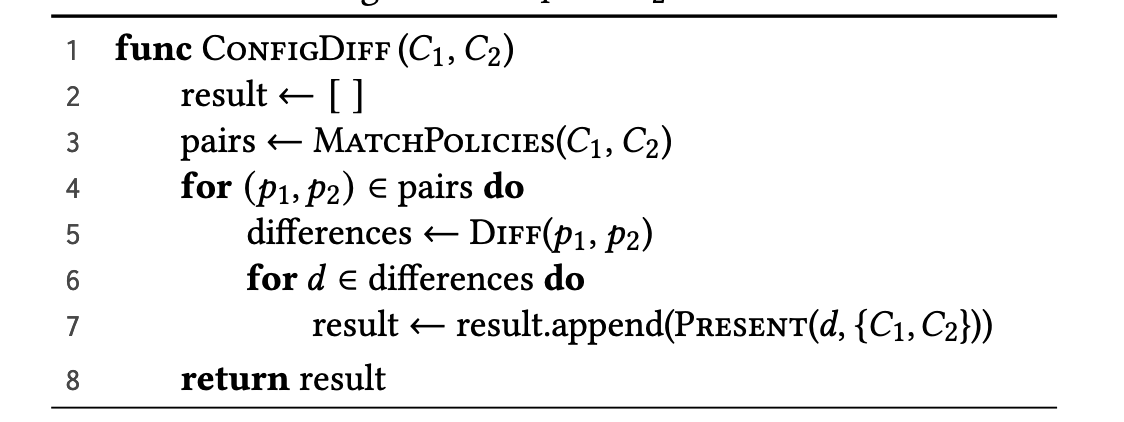
Campion 通过结构等价性检查来检测静态路由和OSPF成本等配置组件的差异。

例子：一个路由配置中有一条路由，另一个没有；或者两个都有，但是属性不同（下一跳、administrative distance）

简单的结构检查与通过语义表示进行的行为检查一样精确：将配置组件与配置的其他部分隔离开来进行检查



1.4 设计与算法



1. C1 和 C2 的相应组件（ACL 或 BGP 路由映射）由 MatchPolicies 函数配对。这可以通过启发式方法完成，例如按名称匹配组件或匹配与同一相邻节点相关的组件，或者此信息可以由用户提供。
2. 对于每个组件对，Diff 函数调用 SemanticDiff 或 StructuralDiff 来生成一组差异，每个差异都可以包含一组输入、每个组件执行的动作以及配置中的位置。
3. Present 函数对结果进行格式设置，以便输出给用户，包括对 SemanticDiff 的结果调用 HeaderLocalize，以便生成输入集的可理解表示形式。

**1.4.1 SemanticDiff**

输入：一对配置组件

返回：所有行为差异的列表

差异定义：(i, a1, a2, t1, t2) 五元组

i：a set of inputs to the components,For dataplane ACLs the inputs are sets of packets, and for route maps they are route advertisements.

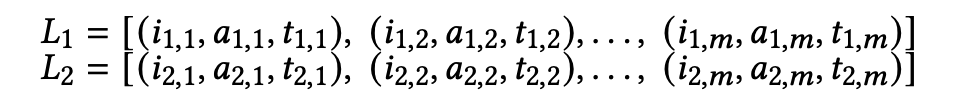
a1 a2:the respective actions taken by the two components when given an input from i. The action for ACLs is either accept or reject, but for route maps the accept action can also set fields such as local preference.

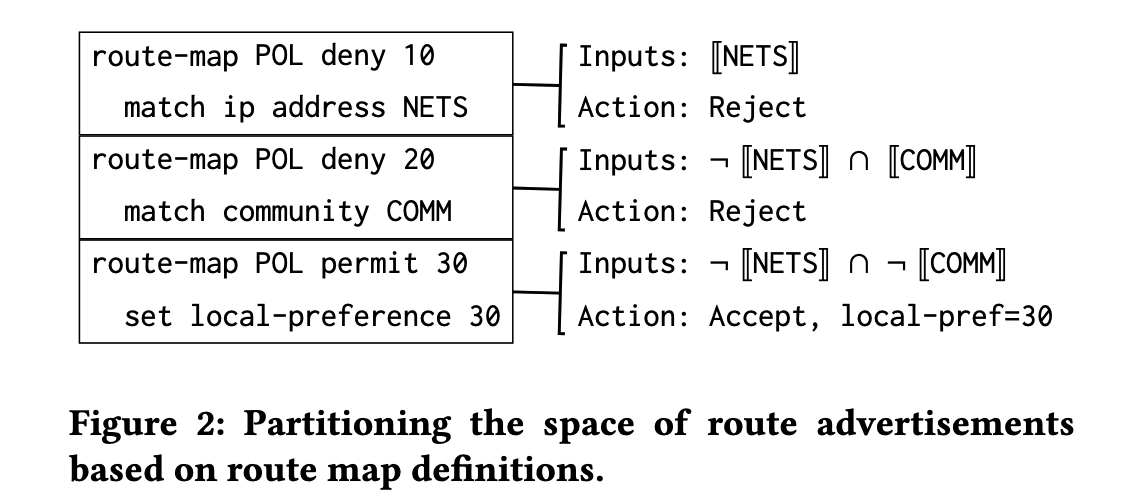
t1 t2:the respective lines of text from the two components that process inputs from i and result in a1 and a2

|  |  |  |
| --- | --- | --- |
| 名称 | 含义 | 举例 |
| i | 组件的一组输入，表示为消息标头上的逻辑公式 | 对于数据平面 ACL，输入是数据包集，对于路由映射，它们是路由通告。 |
| a1 a2 | 当给定来自 i 的输入时，两个组件所采取的相应动作 | ACL 的动作是 accept 或 reject，但对于路由映射，accept动作还可以设置本地首选项等字段 |
| t1 t2 | 两个组件中的相应文本行，用于处理来自 i 的输入并生成 a1 和 a2 |  |

SemanticDiff algorithm

1. 对每个配置组件，inputs 的空间根据它们通过组件的路径划分为等价类。ACL 和路由映射都可以被视为 if-then-else 语句序列，因此当且仅当两个输入通过这些语句采用同一组分支时，它们才属于同一等价类。每个等效类都以符号方式表示为输入（数据包标头或路由通告）上的逻辑谓词。我们的实现使用 BDD 来表示这些谓词。每个等价类还与相应路径上的文本行以及执行的作相关联。因此，此步骤将生成两个三元组列表：





1. 将输入划分为两个组件的等价类后，SemanticDiff 算法将执行成对比较以识别行为差异。对于来自两个分量的每对等价类 （i1，i， a1，i， t1，i ） 和 （i2，j ， a2，j ， t2，j ），如果 i1，i 和 i2，j 具有非空交集，并且作 a1，i 和 a2，j 不同，则存在行为差异。添加(i1,i ∩ i2,j , a1,i, a2,j , t1,i, t2,j )到SemanticDiff结果返回列表。

**1.4.2 HeaderLocalize**

HeaderLocalize 算法根据配置中出现的常量（例如 IP 前缀）生成更易于理解的表示形式，从而处理header localization问题。具体而言，HeaderLocalize 会生成与 ACL 差异相关的所有目标 IP 地址集以及与路由映射差异相关的所有 IP 前缀范围集的紧凑表示。为了便于演示，我们只描述了查找与route map差异相关的前缀范围，但 ACL 的过程是类似的。原则上，HeaderLocalize 也可以扩展到其他路由字段，比如 communities，但我们还没有这样做。目前，Campion 输出一个示例，而不是生成与route map差异相关的所有communities。

对于路由映射，IP 前缀集由前缀范围表示，每个前缀范围都是一对前缀和长度范围。例如，（1.2.0.0/16， 16-32） 是一个前缀范围，其中前缀为 1.2.0.0/16，长度范围为 16-32。如果以下两个条件都成立，则前缀 p 是前缀范围 R 的成员：

1. p 的 IP 地址与 R 的前缀匹配
2. p 的长度包含在 R 的范围内

例如，1.2.3.0/24 是前缀范围 （1.2.0.0/16， 16-32） 的成员，（0.0.0.0/0， 0-32） 是所有前缀的集合，（1.0.0.0/8， 24-24） 是长度为 24 且 1 作为第一个八位字节的所有前缀的集合。如果 R1 的成员是 R2 成员的子集，则我们说前缀范围 R1 包含在另一个前缀范围 R2 中，表示为 R1 ⊂ R2。

HeaderLocalize 的输入是一个 BDD S，表示受已识别策略差异影响的消息集，以及原始配置 C1 和 C2。输出是 S 的前缀范围（根据两种配置中的前缀范围）的表示形式。首先，提取两种配置中的所有前缀范围，以获得集合 R = {R1， R2， . . .Rn }.如果所有前缀的集合 （0.0.0.0/0， 0-32）（我们称之为 U ）不在 R 中，则添加它。此外，R 被延伸，以便在交集下闭合。由于路由映射的每一行都可以根据配置中的前缀范围允许或拒绝路由通告，因此始终可以将集合 S 表示为来自 R 的集合的补码、联合和交集的组合。HeaderLocalize 的目标是确定最小的此类表示形式。

为了找到这种最小表示形式，HeaderLocalize 构建了一个有向无环图 （DAG），该图将 R 中的前缀范围彼此关联。这种数据结构类似于以前用于数据包标头空间的 ddNF 数据结构 [8]，但在这里我们将每个节点与前缀范围相关联，而不是与表示数据平面数据包的三态位向量相关联。HeaderLocalization 的 ddNF 数据结构由一组节点 N、一组边 E ⊆ N × N、一个将节点映射到前缀范围的标记函数 l 和一个根节点组成。它满足以下属性：

1. 根节点标有 U ，这是所有前缀的集合，所有其他节点都可以从它访问。
2. 每个节点都有一个唯一的标签（因此在下面的解释中，我们有时会通过其前缀范围来引用节点，反之亦然）。
3. 用作标签的前缀范围集包含 R，并在交集下闭合。
4. 对于任何节点 m， n ∈ N ，当 l （n） ⊂ l （m） 时，有一条边 （m， n） ∈ E 正好存在，并且没有节点 m′ 使得 l （n） ⊂ l （m′） ⊂ l （m）。

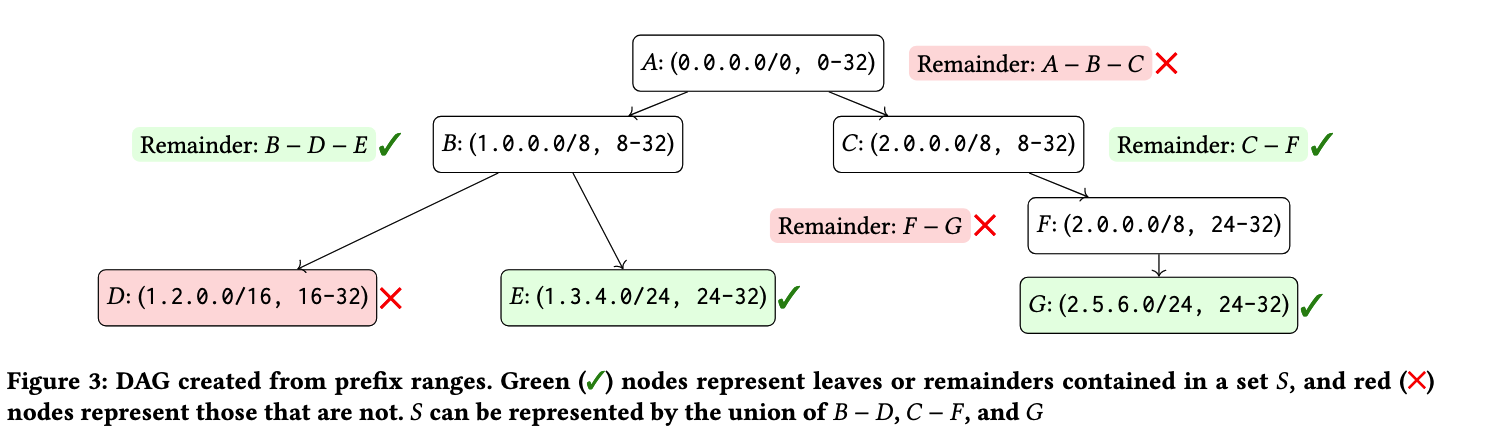


图 3 显示了一组 7 个前缀范围的示例 DAG。每个前缀范围有一个节点，每个节点的前缀范围是其上级前缀范围的子集。例如，D 包含在 B 和 A 中。DAG 是通过一次插入一个前缀范围来构建的，从 U [8] 开始。我们还将每个具有前缀范围 R 的内部节点和传出边缘与标记为 C1、C2、. . . 、Ck 的节点与前缀集 R −C1 −C2 · · ·−Ck .我们将此集称为余数集，因为它是在删除子节点的前缀后保留在 R 中的前缀集。例如，图 3 中节点 B 的余数集是 B − D − E。余数节点集和叶节点集彼此不相交，它们的 union 是 U 。重要的是，由于感兴趣的集合 S 是通过 R 中前缀范围的并集、交集和补码创建的，因此每个余数集和叶前缀范围都具有它包含在 S 中或与 S 不相交的属性。

接下来，HeaderLocalize 使用 DAG 根据 R 中的前缀范围生成 S 的表示形式。这是通过使用如下所示的递归函数 GetMatch 遍历 DAG 来完成的。如果当前节点是叶，则如果该范围包含在 S 中，则其前缀范围 R 将包含在结果中。如果当前节点是 internal，则有两种情况。如果节点的余数包含在 S 中，则在删除 DAG 中未包含在 S 中的任何节点子前缀后，其前缀范围 R 应包含在结果中。后一个过程是通过对 GetMatch 的递归调用和 S 的补集来完成的。如果节点的余数不包含在 S 中，那么我们只需递归子项并合并结果。

GetMatch 算法生成 S 的表示形式，该表示形式为 R − X1 − X2 − .Xk ，其中 R 是前缀范围，但每个 X j 也采用 R − X1 − X2 − . . .Xk .例如，在图 3 中的 DAG 上运行 GetMatch 会生成 {B − D， C − （F − G）}，并且图中的节点用颜色表示算法的过程。作为最后的简化步骤，我们通过一次遍历结果来删除结果中的所有嵌套差异。在我们的示例中，结果 C − （F − G） 被转换为 {C −F， G }，因此集合 S 的最终表示是 {B −D， C −F， G }。

**1.4.2 StructuralDiff**

SemanticDiff可以用于处理所有配置组件。但是一些配置有非常固定的结构格式。作为单个原子值（例如，整数或布尔值）或此类值的简单集合。因此，当模块化考虑时，两个此类组件的等效性与它们的结构有关。

我们的 StructuralDiff 函数实现了这种方法。所有组件都表示为原子值、元组或无序集。测试原子值是否相等。通过测试相应的值是否相等来比较元组。最后，使用集差值比较集。

例如，要检查两个 OSPF 配置是否等效（不包括由 SemanticDiff 处理的路由重分发），只需检查所有相应链路上所有相应属性的等效性就足够了。这意味着两个路由器必须具有指向相同对等体的 OSPF 边缘，并且相应的边缘配置了相同的costs, areas, passive status等。我们可以将每个 OSPF 链路的配置视为其已配置属性的元组，并检查每个相应的属性。相同的方法适用于未使用路由映射实现的 BGP 属性，例如，哪些 Edge 将路由反射器客户端以及是否传播communities。

影响路由的其他组件包括 connected 路由和 static 路由。连接的路由由连接到路由器接口的一组子网形成，路由器之间的区别在于一个路由器中存在的此类子网集，而另一个路由器中不存在。同样，单个静态路由可以表示为一个元组，该元组由目标前缀、下一跃点、管理距离和标签等可选字段组成;所以区别在于一个 Router 中存在的 Tuples 集，而不是另一个 Router 中存在的 Tuples 集。管理距离也可以作为每个协议配置的值进行比较。

如前所述，这些组件的本地化非常简单，因为等价检查是直接在组件的结构上执行的。此外，与路由映射和 ACL 不同，这些组件没有明确的输入概念。因此，对于此类差异，不需要或类似 HeaderLocalize 。

**1.4.2 Debugging an Entire Router**

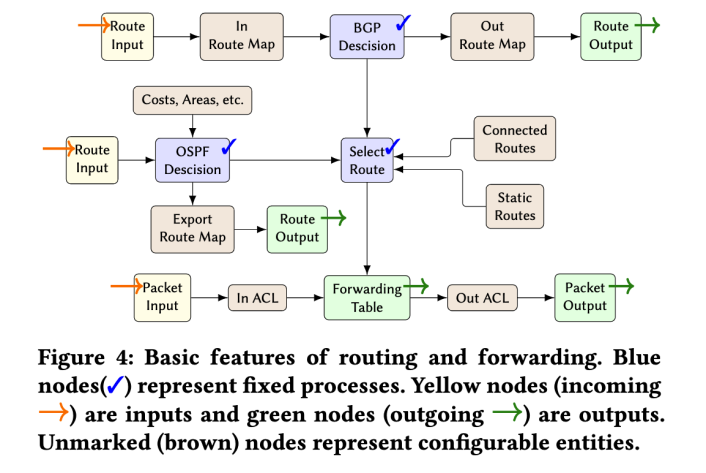
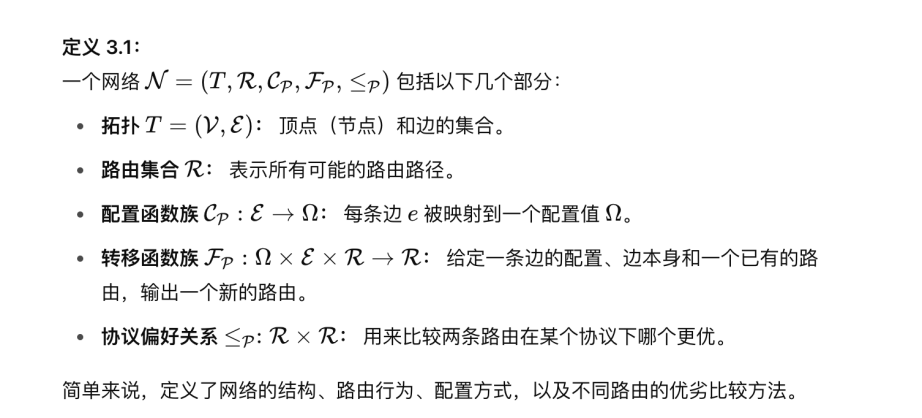
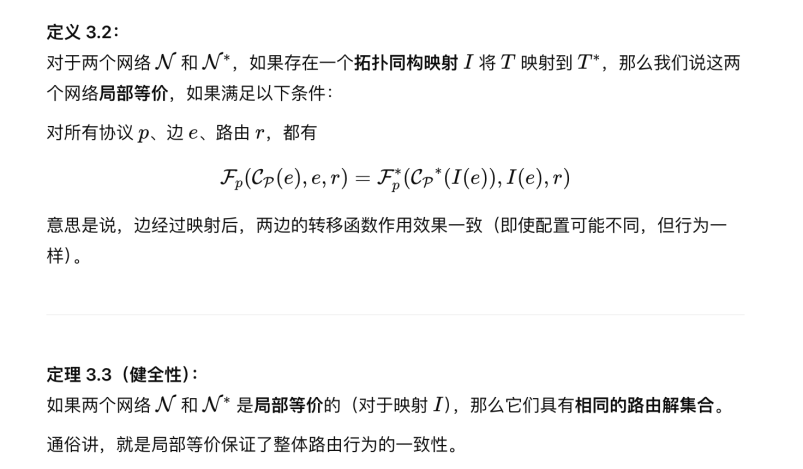


图 4 提供了一个流程图，说明了 Campion 支持的流程。对于路由，有一个 BGP 进程（图顶部）和一个 OSPF 进程（图中），因为这些是最常见的域间和域内路由协议;其他协议也可以类似地添加。图底部显示了路由器转发路由的过程。棕色（未标记）节点代表路由器配置的一部分，而其他组件是固定进程，如路由协议（蓝色 （✓）），或输入路由和数据包（黄色（传入）），或输出和副产品，如所选路由和转发的数据包（绿色（传出））。

假设这些是被比较的配置中使用的唯一路由组件，那么 Campion 是路由器配置等效性的可靠验证器：如果 Campion 没有发现任何差异，则两个路由器配置在行为上是等效的。我们正式化了这样一个事实，即无需对路由协议进行推理即可验证行为等效性，如下所示（我们的正式化考虑了整个网络的行为等效性，但因此也适用于单个路由器的特殊情况）。





1.5 实现和局限性

基于Batfish开发的独立于供应商的配置统一表示实现。主要关注路由和转发中最常用的部分。目前，支持Minesweeper支持的所有配置部分和功能。包括：BGP route map(communities, local preference, and MEDs) ,OSPF属性，静态路由和ACL。数据包集和路由公告（advertisements）由BDD表示。

如上一节所述，有时需要在两个路由器之间匹配相应的组件。我们使用了一些简单的启发式方法，而不是手动指定匹配的组件。对于 BGP 属性和路由映射，我们将具有相同邻居 ID 的连接进行匹配，并报告一个路由器中出现的邻居，而不是另一个路由器中出现的邻居。我们匹配具有相同名称的 ACL。对于 OSPF 属性，我们使用接口名称、Batfish 推断的拓扑及其 IP 地址掩码的组合来匹配接口。这是必需的，因为备份路由器中的接口通常具有不同的 IP 地址。虽然这些启发式方法并不完美，但它们允许快速轻松地运行 Campion。

Campion 可以识别差异并为 Batfish 支持的任何供应商格式执行标头本地化。但是，目前 Campion 只能为 Cisco IOS 和 Juniper JunOS 格式的配置输出确切的文本行，因为我们必须编写解解析器才能将 Batfish 的表示形式转换回原始配置文本。对于其他格式，Campion 不会生成精确的文本行，但它仍然提供大量的本地化信息，包括组件名称、受影响的标题和作。同样，对于某些格式，我们不显示 StructuralDiff 结果的确切文本行，例如 OSPF 成本。但在这些情况下，Campion 提供的本地化信息通常允许作员通过简单的文本搜索找到相关行。

路由映射的 HeaderLocalize 目前仅提供 IP 前缀范围的详尽信息。对于路由通告的其他相关部分（如社区标签），Campion 提供了一个示例。可以扩展 HeaderLocalize 以跨路由通告的多个部分提供详尽的信息，但这样做会增加算法及其输出的复杂性。目前的方法足以让作员理解 Campion 的结果并定位错误。

1.6 Campion使用

|  |
| --- |
| docker pull modnetv/campion  docker run --name campion  -p 9997:9997 -p 9996:9996 modnetv/campion  sudo python3 -m pip install --upgrade pip  sudo python3 -m pip install --upgrade git+https://github.com/batfish/pybatfish.git  python3 run\_diff.py  python3 run\_time\_diff.py |

如果要比较两台备份路由器，请创建一个目录，并将配置分别放在其中名为 “configs ”的子目录下。然后将脚本中的 “curr\_snapshot ”变量设置为该目录的名称。例如，如果将配置放在 “A/configs ”中，则在脚本中设置 “curr\_snapshot = ‘A’”。在脚本中设置变量 “router\_regex”，使其与要测试的路由器名称相匹配。在比较两个备份时，应正好匹配 2 个路由器。将变量 “output\_directory ”设置为结果应写入的目录名称。

如果要比较路由器在不同时间的不同版本，则应创建两个目录，并将配置分别放在其中一个名为 configs 的子目录中。然后用这两个目录的名称更新脚本中的 “ref\_snapshot ”和 “curr\_snapshot ”变量。例如，如果您将旧版本的配置放在 “A/configs ”目录下，而将新版本的配置放在 “B/configs ”目录下，那么请编辑 run\_time\_diff.py，使其包含 “ref\_snapshot = ‘A’”和 “curr\_snapshot = ‘B’”。在脚本中设置变量 “router\_regex”，使其与要测试的路由器名称相匹配。将变量 “output\_directory ”设置为结果应写入的目录名称。

1.6 代码阅读

以BDDAcl引入

BDDAcl 用于将一个访问控制列表（ACL，IpAccessList）转换成一个 二进制决策图（BDD，Binary Decision Diagram）的形式。

* **输入：**

输入一个 IpAccessList（ACL对象），它定义了一组匹配条件（源IP、目标IP、端口等）和动作（允许/拒绝）。

同时需要有一个 BDDPacket，它描述了如何在 BDD 里面建模网络数据包（包头的字段对应哪些BDD变量）。

* 解析 ACL 规则

遍历 ACL 中每一条规则（称为 "lines"）。

对于每条规则，把匹配条件（比如源IP在某个范围）翻译成BDD表达式。

例子：IP在 192.168.0.0/16 → 转成一个BDD，代表满足这个条件的所有包。动作（允许/拒绝）也会在BDD中编码成相应的处理。

* 构造整体 ACL 的 BDD

按照 ACL 中的规则顺序，组合每条规则的 BDD。

遇到第一个匹配的规则，就确定了结果（因为ACL通常是第一个匹配原则）。

使用逻辑运算（AND、OR、NOT）把每一条规则正确拼接成一个大的 BDD，表示整个 ACL 的效果。

允许规则：累加允许路径。

拒绝规则：屏蔽掉拒绝路径，不再继续往下检查。

* 处理嵌套 ACL 和 Named IP Spaces

如果某条规则引用了另一个ACL或命名的IP空间（比如aclEnv、ipSpaceEnv中定义的），递归处理引用。

把引用的内容也转成对应的 BDD，再替换进来。

* 输出

最终生成一个整体的 BDD，它可以直接用来：

判断任意一个包是否被允许。

进一步进行集合运算（比如两个ACL是否冲突）。

BDDAcl 类封装了这个 BDD，并提供常见操作，比如：

restrict：只保留匹配某些前缀的包。

equals：比较两个ACL逻辑是否一样。



