NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion

Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev, ETH Zürich

https://www.usenix.org/conference/nsdi18/presentation/el-hassany

This paper is included in the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’18).
April 9–11, 2018 • Renton, WA, USA
ISBN 978-1-939133-01-4

Open access to the Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation is sponsored by USENIX.
**NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion**

**Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, Martin Vechev**

*ETH Zürich*

netcomplete.ethz.ch

**Abstract**

Network operators often need to adapt the configuration of a network in order to comply with changing routing policies. Evolving existing configurations, however, is a complex task as local changes can have unforeseen global effects. Not surprisingly, this often leads to mistakes that result in network downtimes.

We present NetComplete, a system that assists operators in modifying existing network-wide configurations to comply with new routing policies. NetComplete takes as input configurations with “holes” that identify the parameters to be completed and “autocompletes” these with concrete values. The use of a partial configuration addresses two important challenges inherent to existing synthesis solutions: (i) it allows the operators to precisely control how configurations should be changed; and (ii) it allows the synthesizer to leverage the existing configurations to gain performance. To scale, NetComplete relies on powerful techniques such as counter-example guided inductive synthesis (for link-state protocols) and partial evaluation (for path-vector protocols).

We implemented NetComplete and showed that it can autocomplete configurations using static routes, OSPF, and BGP. Our implementation also scales to realistic networks and complex routing policies. Among others, it is able to synthesize configurations for networks with up to 200 routers within few minutes.

**1 Introduction**

In a world where more and more critical services converge on IP, even slight network downtimes can cause large financial or reputational losses. This strategic importance contrasts with the fact that managing a network is surprisingly hard and brittle. Out of high-level requirements, network operators have to come up (often manually) with low-level configurations specifying the behavior of hundreds of devices running complex distributed protocols. A single misconfiguration can bring down the network infrastructure, or worse, a piece of the Internet in case of BGP-related misconfigurations. Every few months downtimes involving major players such as NYSE [1], Google [2], Facebook [3], or United Airlines [4] make the news. Actually, studies show that human-induced misconfigurations, not physical failures, explain the majority of downtimes [5].

To address these challenges, recently there has been an increased interest in configuration verification [6, 7, 8, 9, 10, 11, 12, 13] and synthesis [14, 15, 16, 17, 18, 19, 20]. Configuration synthesis in particular promises to alleviate most of the operator’s burdens by deriving correct configurations out of high-level objectives.

**Challenges in network synthesis** While promising, network operators can still be reluctant to use existing synthesis systems for at least three reasons: (i) interpretability: the synthesizer can produce configurations that differ wildly from manually provided ones, making it hard to understand what the resulting configuration does. Moreover, small policy changes can cause the synthesized configuration (or configuration templates in the case of PropaneAT [15]) to change radically; (ii) protocol coverage: existing systems [15, 16] are restricted to producing BGP-only configurations, while most networks rely on multiple routing protocols (e.g., to leverage OSPF’s fast-convergence capabilities); and (iii) scalability: recent synthesizers such as SyNET [20] handle multiple protocols but do not scale to realistic networks.

**NetComplete** We present a system, NetComplete, which addresses the above challenges with partial synthesis. Rather than synthesizing a new configuration from scratch, NetComplete allows network operators to express their intent by sketching parts of the existing configuration that should remain intact (capturing a high-level insight) and “holes” represented with symbolic values which the synthesizer should instantiate (e.g., OSPF weights, BGP import/export policies).
NetComplete then autocompletes these "holes" such that the resulting configuration leads to a network that exhibits the required behavior. Our approach supports a practically relevant scenario as few operators ever start from scratch but rather modify their existing configurations (e.g., OSPF weights) to handle new routing requirements. This evolving approach also has the benefit of better explainability as large parts of the existing configuration are preserved in the newly synthesized configuration. Further, because we focus on synthesizing parts of the configuration, there is an opportunity to scale the synthesizer to realistic networks. This opportunity arises even though NetComplete is quite expressive: it handles static routes, OSPF, and BGP\(^1\) as well as a variety of essential routing requirements such as waypointing, failure-resilience, load-balancing, and traffic isolation.

NetComplete reduces the autocompletion problem to a constraint satisfaction problem that it solves with SMT solvers (e.g., Z3 \([21]\)). The main challenge is that a naive encoding of the problem leads to complex constraints that cannot be solved in reasonable time (e.g., within a day). To scale, NetComplete relies on two key insights: (i) partial evaluation along with (ii) network-specific heuristics to efficiently navigate the search space. Specifically, it speeds up BGP synthesis by propagating symbolic announcements through partial BGP policies allowing it to eliminate many variables. For OSPF, NetComplete is 100x faster than a naive encoding via a new counter-example guided inductive synthesis algorithm. Overall, NetComplete autocompletes configurations for networks with up to 200 routers in few minutes.

**Contributions** Our main contributions are:

- A new approach to network-wide configuration synthesis based on autocompletion of partial configurations. It enables operators to evolve existing configurations so they match new requirements.

- A scalable synthesis procedure based on SMT constraints which relies on partial evaluation techniques along with domain-specific heuristics and counter-example guided inductive synthesis.

- An end-to-end implementation of our approach in a system called NetComplete which outputs actual Cisco configurations.

- A comprehensive evaluation of NetComplete using a variety of real-world topologies and complex requirements. Our results demonstrate that NetComplete can effectively autocomplete partial configurations for large networks with up to 200 routers within few minutes.

---

\(^1\)We plan to add support for more protocols and mechanisms in future work, including MPLS and route redistribution.

# 2 Motivating Scenarios

In this section, we motivate the need for NetComplete through three practical use cases rooted within existing network management practices. These use cases are difficult or practically impossible to solve today.

**Scenario 1:** Evolving configurations preserving existing semantics. Existing configurations typically embed deep knowledge of semantics and design guidelines. For instance, operators often use specific OSPF weights to identify primary/backup links, and specific BGP local-preferences or communities to identify their peers. This (often unwritten) semantic helps them reason about the network-wide configuration. At the same time, these rules also reduce the operators flexibility as it can complicate the implementation of new routing requirements, e.g., by requiring the modification of multiple weights instead of one.

NetComplete allows operators to communicate such semantics as constraints on the configuration sketch and let the synthesizer find a valid network-wide configuration that adheres to the operators style.

**Scenario 2:** Simplifying federated or constrained management. Network configurations are often maintained by multiple teams of operators \([22\,23]\), each responsible for some parts (e.g., edge vs core) or functionalities. Coordinating changes in these federated configurations tends to be challenging as multiple teams need to come together. With NetComplete, the operators can easily explore whether there is a way to implement the policy locally, for instance, without adapting the BGP configuration (i.e., by restricting changes to the OSPF configuration). Similar requirements appear in heterogeneous networks where not all routers support all protocols (e.g., due to licensing issues or device capabilities).

NetComplete allows operators to simply communicate such constraints as part of the sketch and let the synthesizer find a multi-protocol configuration.

**Scenario 3:** Configuration Refactoring and Network Merging. Configurations evolve over time and this increases their complexity. Design decisions that made sense in the past may no longer do, requiring refactoring. Other examples calling for large refactoring include merging and acquisitions; e.g., when a company buys another one and wishes to integrate their networks \([24]\).

NetComplete helps operators to refactor configurations by enabling them to morph entire pieces of their existing configurations, e.g., to adopt the configuration guidelines of one network and let the synthesizer compute and propagate the changes network-wide.
3 Overview

We now show how given a network topology, high-level routing requirements, and a partial configuration, NetComplete autocompletes the partial configuration to a correct network-wide configuration. First, we present a small running example and define NetComplete’s inputs. We then present the key synthesis steps to produce the output configuration before explaining the more complex steps in detail in the following sections.

3.1 Running Example

In Fig. [2], we show how a network operator would use NetComplete to synthesize a network-wide configuration that enforces routing requirements. We consider that the Autonomous System (AS) of the operator’s network is AS300 and consists of four routers: A, B, C, and D. This network is further connected to one customer peer AS100 and three external peers: AS200, AS300, and AS400.

High-level Routing Policy The policy for our example is given in Fig. [1] Rule (1) disallows transit traffic between external peers; e.g., AS200 cannot send traffic to AS300 through the network. Rule (2) defines how the customer peer accesses prefixes announced by external peers: AS300 is most preferred, followed by AS400, and then AS200. Traffic to AS200 may exit via B or C, where B is preferred. Rules (3) and (4) capture traffic engineering requirements. Note that this policy can be formalized in a high-level SDN-like language, such as Propane [15], Genesis [13], Frenetic [23], or SyNET [20].

3.2 NetComplete Inputs

NetComplete takes three inputs: (i) network topology, (ii) routing requirements, and (iii) a configuration sketch.

(1) Network Topology The network topology is given via a graph over the set of routers (A, B, C, and D) and external peers (AS100, AS200, AS300, and AS400). An edge represents a physical link that connects two nodes.

(2) Routing Requirements We now describe the type of requirements supported by NetComplete. We start with some basic notation. A routing path is of the form: \[ P := \text{Src} \rightarrow R_1 \rightarrow \cdots \rightarrow R_n \rightarrow \text{Dst} \], where Src and Dst are source and destination routers, respectively, and \( R_1, \ldots, R_n \) are router identifiers. We use a wildcard notation to denote sets of simple paths, i.e., paths without repeated nodes. For example, \( \text{Src} \rightarrow * \rightarrow \text{Dst} \) denotes all simple paths from Src to Dst.

NetComplete supports positive and negative requirements. Positive requirements have the form

\[ \text{Req} := (P, \cdots, P) \] \( \lor \) \( (P = \cdots = P) \) \( \Rightarrow \) \( \text{Req} \Rightarrow \text{Req} \)

Rule 1 No transit between AS200, AS300, and AS400.
Rule 2 Traffic from the customer peer AS100 to the external peers prefers exit routers in order: AS300, AS400, AS200 via B, AS200 via C.
Rule 3 Traffic from AS100 to AS300 is load-balanced along \( A \rightarrow C \) and \( A \rightarrow D \rightarrow C \); if both paths are unavailable, then the path \( A \rightarrow B \rightarrow C \) is used.
Rule 4 Traffic from AS100 to AS400 must follow the path \( A \rightarrow B \rightarrow C \).

Figure 1: High-level policy for our running example

where \( P \) is a routing path. All routing paths that appear in a requirement must have identical source and destination. The semantics of requirements is as follows:

An any-path requirement \( (P_1, \ldots, P_k) \) is satisfied if the traffic from the source to the destination is forwarded along any available path in \( \{P_1, \ldots, P_k\} \). The requirement is not-applicable if all paths \( P_1, \ldots, P_k \) are unavailable. We remark that any-path requirements are used to ensure failure-resilience. We will refer to any-path requirements \( P \) consisting of a single path \( P \) as simple requirements.

An ECMP requirement \( (P_1 = \cdots = P_k) \) is satisfied if the traffic fromSrc to Dst is load-balanced among all available paths in the set \( \{P_1, \ldots, P_k\} \). The requirement is not-applicable if all paths \( P_1, \ldots, P_k \) are unavailable. We remark that ECMP requirements are useful to capture load-balancing.

An ordered requirement \( \text{Req}_1 \gg \text{Req}_2 \) defines a preference over requirements. This requirement is satisfied if the most preferred applicable requirement is satisfied, and it is not-applicable if both requirements are not-applicable. For example:

\[ (\text{AS}100 \rightarrow A \rightarrow B \rightarrow C \rightarrow \text{AS}300) \] \( \gg \) \( (\text{AS}100 \rightarrow A \rightarrow C \rightarrow \text{AS}300) \)

is satisfied if traffic from AS100 to AS300 is forwarded along this path if it is available:

\[ \text{AS}100 \rightarrow A \rightarrow B \rightarrow C \rightarrow \text{AS}300 \]

Otherwise traffic is forwarded along the path:

\[ \text{AS}100 \rightarrow A \rightarrow C \rightarrow \text{AS}300 \]

NetComplete also supports negative requirements of the form \( \neg \{P_1, \ldots, P_k\} \), where \( \{P_1, \ldots, P_k\} \) is a set of routing paths. This requirement is satisfied if traffic is not forwarded along any path in this set. Negative requirements are useful to express traffic isolation.

The requirements for our running example are given in Fig. [2]. We interpret sets of paths, such as \( \text{AS}100 \rightarrow * \rightarrow \text{AS}300 \), as any-path requirements. Policy rules 1, 2, 3, and 4, given Fig. [1] are specified as requirements 1, 2, 7, 8, and 9, respectively. We use a natural assignment
of requirements to protocols. For example, requirements 1 – 7 pertain to external peers and they are assigned to the Border Gateway Protocol (BGP). Requirement 8 pertains to traffic engineering within the network and is assigned to the Open-Shortest Path First (OSPF) protocol, which forwards traffic along the shortest path. Note that requirements 8 and 9 cannot be both enforced by OSPF. To enforce requirement 9, the cost of A → B → C must be lower than that of A → C and A → D → C. However, this would also divert traffic from AS100 to AS300 to be forwarded along routers A → B → C, which would violate requirement 8. To this end, requirement 9 is enforced using a static route.

We remark that the requirements above can be specified manually by the operator, or using existing systems [15][16][18][25] that compile high-level policies to forwarding paths.

(3) Configuration Sketch Configuration sketches are router configurations where some of the parameters are left symbolic. To specify symbolic values, the operator tags parts of the configurations with a question mark symbol ? (instead of writing concrete values). The symbol ? represents: (i) specific attributes (e.g., OSPF link cost, BGP local preference, etc.); or (ii) entire import / export policies, e.g., match?, action ?.

As an example, we depict the sketch of router B’s configuration in Fig. 2c. We remark that operators can write additional constraints to restrict how NetComplete instantiates symbolic parameters. For example, the symbolic OSPF link cost in the sketch of router B is constrained to values between 10 and 100.

This sketching language enables NetComplete to be used in different scenarios. For example, changes can be restricted to certain parts of the network [Scenario 2]. By leaving most of the configurations symbolic, an operator can explore a large range of possible configurations that implement a given set of requirements [Scenarios 1 and 3]. Moreover, an operator can also provide a fully concrete configuration to verify its correctness.

\[\text{Input} \rightarrow \text{NetComplete} \rightarrow \text{Output}\]

Figure 2: Overview of NetComplete. The inputs are: (a) network topology, (b) routing requirements, and (c) a configuration sketch. The output is a configuration for each router, for the configuration of router B see (e).

---

\[\text{router bgp 6500} \]
\[\text{neighbor AS200 export route-map exp-p1} \]
\[\text{neighbor AS200 import route-map imp-p1} \]
\[\text{ip community-list C1 permit} \]
\[\text{ip community-list C2 permit} \]
\[\text{route-map imp-p1 permit 10} \]
\[\text{set ?} \]
\[\text{route-map exp-p1 ?} \]
\[\text{match community C2} \]
\[\text{match community C1} \]

---

\[\text{ip address 10.0.0.0 255.255.255.254} \]
\[\text{network 10.0.0.1 0.0.0.1 area 0.0.0.0} \]
\[\text{neighbor AS200 export route-map exp-p1} \]
\[\text{neighbor AS200 import route-map imp-p1} \]
\[\text{ip community-list C1 permit} \]
\[\text{ip community-list C2 permit} \]
\[\text{route-map imp-p1 permit 10} \]
\[\text{set ?} \]
\[\text{route-map exp-p1 ?} \]
\[\text{match community C2} \]
\[\text{match community C1} \]
3.3 Configuration Synthesis

NetComplete synthesize a network-wide configuration that enforces the requirements in three steps.

First, it synthesize the sessions between routers that have a physical link between them and may be necessary to enforce the routing requirements. Further, it configurates any static routes defined in the requirements. For example, for requirement AS100 → A → B → C → AS400, NetComplete establishes a session between A → B and B → C, and configures static routes at A and B.

Second, NetComplete synthesize router-level BGP configurations based on the BGP routing requirements. To this end, NetComplete computes a propagation graph that captures which BGP announcements are exchanged between the routers and in what order they must be selected. NetComplete then synthesize BGP configurations that enforce the constructed propagation graph. We explain this step in detail in §4. Note that BGP may select routes based on path costs (computed by OSPF). Therefore, whenever this is necessary to enforce the requirements, the BGP synthesizer outputs additional OSPF requirements to be enforced by the OSPF synthesizer.

Third, NetComplete synthesize OSPF costs that enforce all OSPF requirements. This is a well-known hard problem that is difficult to scale to large networks. We solve the problem in §4 via a novel counter-example guided inductive synthesis algorithm.

If all synthesis steps succeed, NetComplete outputs a configuration that is guaranteed to enforce the requirements. Otherwise, a counter-example is returned to indicate that the requirements cannot be enforced for the given inputs. Based on this counter-example, the network operator can modify the partial configuration (by making more parameters symbolic) or adapt the requirements. We present a detailed evaluation of NetComplete with practical topologies and requirements in §6.

4 BGP Synthesis

We now present NetComplete’s BGP synthesizer which takes as input BGP requirements and computes router-level BGP policies. It also outputs a set of OSPF requirements (to be fed to NetComplete’s OSPF synthesizer) if the BGP requirements cannot be enforced by BGP policies alone. In the following, we first overview the BGP protocol ([4.1]), then present the construction of a BGP propagation graph which defines correct propagation of BGP announcements ([4.2]). We illustrate NetComplete’s BGP sketches in §4.3 and propagation of (symbolic) announcements over them in §4.4. Finally, we describe our BGP synthesis procedure ([4.5]).

### Table 3: BGP attributes supported by NetComplete

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix</td>
<td>A value that represents a set of destination IPs that belong to the same traffic class</td>
</tr>
<tr>
<td>LocalPref</td>
<td>A positive integer that indicates the degree of preference for one route over the other routes</td>
</tr>
<tr>
<td>Origin</td>
<td>The origin of the announcement: IGP, EGP, or Incomplete</td>
</tr>
<tr>
<td>MED</td>
<td>(Multi-Exit Discriminator) A positive integer that indicates which of the multiple routes received from the same AS is selected</td>
</tr>
<tr>
<td>ASPath</td>
<td>The AS path to reach the destination</td>
</tr>
<tr>
<td>ASPathLen</td>
<td>The length of the AS path to the destination</td>
</tr>
<tr>
<td>NextHop</td>
<td>The router to which to forward packets</td>
</tr>
<tr>
<td>Communities</td>
<td>A list of tags carried with the announcement.</td>
</tr>
</tbody>
</table>

Figure 3: BGP attributes supported by NetComplete.

4.1 BGP Protocol

The BGP protocol is used to exchange information between ASes. An AS sends announcements to its neighboring ASes to inform them that it can carry traffic to prefixes (i.e., sets of IP addresses). Announcements are also exchanged within an AS to disseminate routing information among routers. Note that operators may partition their network into multiple ASes to use BGP to enforce routing requirements within the network. We refer to the ASes under the operator’s control as private and to the remaining as public ASes.

Announcements have attributes, which are used to select a single best route out of (possibly) multiple routes to the same prefix; see Fig.3. A router processes each received announcement using import filters, which may drop the announcement or modify its attributes. Then, the router selects the best route according to a local BGP policy, processes it using export filters, and forwards the result to its neighboring routers.

Each router uses the following preferences when selecting the best route:
1. Prefer higher LocalPref;
2. Prefer shorter ASPathLen;
3. Prefer lower origin type: IGP < EGP < Incomplete;
4. Prefer lower MED;
5. Prefer announcements from external routers;
6. Prefer lower IGPCost, calculated by the network’s Internal Gateway Protocol (IGP), such as OSPF.

We assume prefixes in announcements do not overlap (as we can use known techniques [3] to ensure this).

4.2 BGP Propagation Graph

We present how NetComplete builds, for each prefix, a propagation graph that defines a correct enforcement of the BGP routing requirements for that prefix. In more
details, NetComplete first constructs a graph $G_{ebgp}$ that only considers announcements learned over eBGP. Then, it refines $G_{ebgp}$ into $G_{bgp}$, which also defines how announcements are propagated internally (using iBGP). In Fig. 4, we illustrate the steps on our running example.

**Construct eBGP propagation graph** The graph $G_{ebgp}$ contains one node for each private/public AS. For our example, $G_{ebgp}$ has one private AS, AS500, and four public ones AS100, . . . , AS400; see Fig. 4.

The graph $G_{ebgp}$ has two kinds of labeled edges: propagate and block edges, labeled with the preference order over announcements and, respectively, announcements that must be dropped.

To add propagate edges, NetComplete traverses each positive BGP requirement backwards and appends edges along the traversed ASes. For example, for the requirement AS100 $\rightarrow$ * $\rightarrow$ AS300, NetComplete traverses three ASes and adds the propagate edges AS300 $\rightarrow$ AS500 and AS500 $\rightarrow$ AS100. While adding these edges, NetComplete tracks the set of announcements that must be propagated along them and labels the edges with the preference order based on the requirements.

To add block edges, NetComplete traverses each negative requirement and adds block edges to enforce it. For example, for the requirement AS200 $\rightarrow$ * $\rightarrow$ AS300, it adds the block edge AS500 $\rightarrow$ AS200, label with !AS300, to enforce the requirement.

Once $G_{ebgp}$ is fully constructed, NetComplete checks if preferences over announcements are consistent. To illustrate, suppose AS1 must select announcements from AS2, and AS2 must select from AS3. Then, the preferences over announcements labeled along the edges AS3 $\rightarrow$ AS2 and AS2 $\rightarrow$ AS1 must match.

**Construct iBGP propagation graph** Next, NetComplete refines $G_{ebgp}$ into a detailed propagation graph, $G_{bgp}$, that also accounts for iBGP.

First, for each private AS in $G_{ebgp}$, NetComplete adds to $G_{bgp}$ all BGP-enabled routers within that AS. For our example, NetComplete adds the routers A, B, C, and D.

Second, NetComplete connects the neighbor routers between ASes that have an edge in $G_{ebgp}$. For example, for edge AS200 $\rightarrow$ AS500 in $G_{ebgp}$, NetComplete adds the edges AS200 $\rightarrow$ B and AS200 $\rightarrow$ C to $G_{bgp}$.

Finally, NetComplete extends the paths learned via eBGP. Note that in iBGP routers will not export routes learned from another iBGP router. Similar to $G_{ebgp}$, nodes in $G_{bgp}$ are labeled with the preferences over announcements and NetComplete check if the preferences over announcements are consistent.

### 4.3 BGP Policies

We now present the semantics of BGP policies. A BGP policy applies on a set of announcements and has a match expression followed by zero or more actions. The match expression is a boolean formula over the announcement’s attributes. If the match expression holds for the input announcement, then the actions are executed which modify the announcement’s attributes or drop the announcement.

For example, the following policy:

```
1 BGPPolicy
2 match next-hop AS200
3 set local-pref 10
```

matches an announcement whose NextHop attribute is set to AS200 and sets the value of attribute LocalPref to 10.

---

3While NetComplete does not support Route Reflectors, we plan to add support for them as their functionality is similar to eBGP.
Sketching BGP Policies

NetComplete allows the network operator to define the policy sketch at three levels of details: (i) everything is concrete (no holes), (ii) define the types of matches and actions but leave the specific values empty (see Fig. 5a), (iii) or leave the matches and actions as holes (see Fig. 5b). We formalize the encoding of such sketches using SMT constraints in Appendix B. In §4.5 we show how NetComplete synthesizes BGP policy and instantiates the symbolic values in the given sketch to enforce the BGP propagation graph.

4.4 Processing Symbolic Announcements

We present how NetComplete processes symbolic BGP announcements passing through the various BGP policy sketches in the network. Given an announcement $A$, we write $\text{attr}_A$ to denote the attribute $\text{attr}$ of $A$. For instance, $\text{LocalPref}_A$ returns $A$’s local preference. Each announcement attribute either has a concrete value, if its value is fixed by the partial configuration, or a symbolic value, if a correct concrete value is yet to be discovered by the BGP synthesizer.

We represent announcements symbolically as their attribute values are constrained by the BGP policies, which are yet to be synthesized by NetComplete. Each announcement $A$ is represented with symbolic variables $\text{Prefix}_A, \ldots, \text{NextHop}_A$. The set of possible attribute values of $A$ is captured by a conjunction of constraints over these variables. For example, the constraint

$$ (\text{NextHop}_A = \text{AS}200) \land (0 < \text{LocalPref}_A < 50) $$

captures all announcements whose next hop is AS200 and local preference is a positive integer smaller than 50.

In addition to the attributes listed in Fig. 3 we introduce two boolean variables: $\text{Permitted}_A$, which indicates whether the announcement $A$ is dropped, and $\text{eBGP}_A$, which indicates whether $A$ is sent via eBGP or iBGP.

Processing Announcements with Policy Sketches

A BGP policy sketch takes as input a symbolic announcement $A_{in}$ (a set of constraints over $A_{in}$’s attributes) and outputs another symbolic announcement $A_{out}$. To compute the set of possible output announcements for a given input announcement, we take the conjunction of the BGP sketch constraints with the constraint that captures the set of possible concrete input announcements.

To illustrate this step, consider the input announcement $\text{NextHop}_{A_{in}} = \text{AS}200$ and the BGP sketch in Fig. 5a. Since the $\text{NextHop}$ attribute is concrete and equal to AS200, NetComplete knows that the input announcement would match this policy. Therefore, NetComplete captures the set of possible output announcements with the constraint:

$$ (\text{LocalPref}_{A_{out}} = \text{Var}_1) \land (0 < \text{Var}_1 < 50) $$

$$ \land (\text{NextHop}_{A_{out}} = \text{NextHop}_{A_{in}}) \land \cdots $$

Namely, the local preference of the output announcement is set to the value of $\text{Var}_1$, which is constrained to positive values below 50 (to be synthesized by NetComplete), and all remaining attributes are identical to those in the input announcement (captured with equality constraints, such as $\text{NextHop}_{A_{out}} = \text{NextHop}_{A_{in}}$).

As another example, consider the input announcement $\text{NextHop}_{A_{in}} = \text{Var}_1$ where the $\text{NextHop}$ attribute is symbolic. When evaluating this announcement with the BGP sketch in Fig. 5a, NetComplete captures the set of possible output announcements with the following constraint:

$$ \text{if } \text{Var}_1 = \text{AS}200 $$

$$ \text{then } (\text{LocalPref}_{A_{out}} = \text{Var}_2) \land (0 < \text{Var}_2 < 50) $$

$$ \land (\text{NextHop}_{A_{out}} = \text{NextHop}_{A_{in}}) \land \cdots $$

$$ \text{else } (\text{NextHop}_{A_{out}} = \text{NextHop}_{A_{in}}) \land \cdots $$

This constraint is more complex because the result of the match expression depends on the symbolic next hop ($\text{Var}_1$). If the next hop is AS200, then the local preference is set to $\text{Var}_2$ and all remaining attributes remain unchanged. Otherwise, all attributes in the output announcement $A_{out}$ are identical to those in the input announcement $A_{in}$.

Encoding Selection of Announcements

When a BGP router receives different announcements for the same prefix, it uses the preference ordering to select the best route; see §4.1. We encode the selection process into two SMT predicates: $\text{PrefNoIGP}(A_1, A_2)$ and $\text{Pref}(A_1, A_2)$. The predicate $\text{PrefNoIGP}(A_1, A_2)$ holds if and only if $A_1$ is preferred over $A_2$ without considering the IGP costs of $A_1$ and $A_2$. While the predicate $\text{Pref}(A_1, A_2)$ holds if and only if $A_1$ is preferred over $A_2$ with considering the IGP costs of $A_1$ and $A_2$. We show the encoding of these predicates in Appendix A.

4.5 BGP Policy Synthesis

We now describe how NetComplete synthesizes BGP policies from requirements and policy sketches.

Encoding Requirements
Suppose that a router receives multiple announcements $A_1, \ldots, A_n$ to the same prefix. The BGP propagation graph identifies a preference with which the announcements must be selected by the router.
Suppose the router must select announcements \(A_1, A_2,\) and \(A_3\) in the order \(A_1 \gg A_2 \gg A_3\). We encode this requirement with the following constraint:

\[
\text{Pref}(A_1, A_2) \land \text{Pref}(A_2, A_3) \land (\forall i \in [4, \ldots, n]. \text{Pref}(A_i, A_i))
\]

Note that simpler requirements that do not stipulate a particular order are a special case. For example, if a requirement stipulates that an announcement \(A_4\) is selected as the best route, the above constraint becomes:

\[
\forall i \in [1, n]. k \neq i \implies \text{Pref}(A_i, A_i)
\]

### Overall Synthesis Algorithm

Putting all pieces together, the complete algorithm employed by NetComplete to synthesize concrete BGP policies is as follows:

**Step 1 (4.2):** Construct a BGP propagation graph \(G_{\text{bgp}}\) from the given requirements and network topology.

**Step 2 (4.3):** Encode the routers’ BGP policy sketches.

The result is a constraint \(\varphi_{\text{bgp}}\) over variables \(\overline{S}\). Each concrete instantiation of the variables \(S\) identifies concrete BGP policies.

**Step 3 (4.4):** Declare symbolic variables \(\overline{A}\) to represent all announcements propagated through the BGP propagation graph.

Propagate all symbolic announcements through the policy sketches. The result is an SMT constraint \(\varphi_{\text{announcements}}\) over the variables \(\overline{S}\) and \(\overline{A}\).

**Step 4 (Synthesis without additional OSPF requirements):** Encode the route selection process and the requirements with the selection predicate \(\text{Pref}NoIGP\), resulting in SMT constraints \(\varphi_{\text{select}}\) and \(\varphi_{\text{req}}\) over the variables \(\overline{A}\). If a model of \(\varphi_{\text{select}} \land \varphi_{\text{req}}\) exists, then derive concrete BGP policies and return; otherwise, go to Step 5.

**Step 5 (Synthesis with additional OSPF requirements):**

Find the unsatisfiable core of \(\varphi_{\text{select}} \land \varphi_{\text{req}}\) and derive a set \(S\) of pairs \((A_1, A_2)\) of announcements that cannot be correctly selected without considering their IGP costs. Modify the constraint to:

\[
(\bigwedge_{(A_1, A_2) \in S} \text{IGPCost}_{A_1} < \text{IGPCost}_{A_2}) \implies \varphi_{\text{select}} \land \varphi_{\text{req}}
\]

If a model of this constraint exists, then derive BGP policies, create OSPF requirements from the set \(S\), and return; otherwise, return that the requirements cannot be satisfied.

### 5 OSPF Synthesis

We now present NetComplete’s OSPF synthesizer. OSPF is a Dijkstra-based routing protocol that forwards traffic along the shortest path, where path costs are computed based on the OSPF cost attached to each link.
AS100 → A → B → C → AS300. Note, in this example, Paths(AS100, AS300) = S. Therefore, we have Paths(AS100, AS300) \ S = ∅ and condition (iii) vacuously holds.

**Naive OSPF Synthesis** A naive synthesis solution is to encode all requirements with constraints, as described above, and to then use a constraint solver to discover a model that identifies correct link costs. Unfortunately, phrasing the OSPF synthesis problem directly into SMT does not scale to large networks; cf. [20]. The main issue is the for-all (∀) quantifier in the constraints used to encode that a path has a lower cost among all other simple paths with the same source and destination.

### 5.2 Counter-Example Guided Inductive Synthesis for OSPF

We now present our new counter-example guided inductive synthesis (CEGIS) algorithm for OSPF. CEGIS is a contemporary approach to synthesis, where a correct solution is iteratively learned from counter-examples [26]. CEGIS algorithms tend to work quite well in practice because often a small number of counter-examples (that is, few iterations) is sufficient to discover a correct solution.

The OSPF synthesis problem amounts to finding a model of logical constraints of the form:

\[ \exists \mathcal{C}. \text{ENCODEOSP}(\mathcal{C}, r, \text{Paths}(r)) \]

where \( \mathcal{C} \) is the set of variables that represent link costs, \( r \) is an OSPF requirement, \( \text{Paths}(r) \) is the set of all paths from the source \( \text{Src} \) and destination \( \text{Dst} \) provided in the requirement \( r \), and \( \text{ENCODEOSP}(\mathcal{C}, r, \text{Paths}(r)) \) returns a logical formula that encodes the requirement’s satisfaction (as described in [5.1]). Finding a model of this formula directly using a constraint solver is difficult due to the large number of paths in \( \text{Paths}(r) \). To avoid this quantifier, CEGIS restricts the constraint to a (small) set of paths \( S = \{P_1, \ldots, P_n\} \subseteq \text{Paths}(r) \). The resulting constraint is:

\[ \exists \mathcal{C}. \text{ENCODEOSP}(\mathcal{C}, r, S) \]

which is easier to solve by existing constraint solvers. A model of this constraint identifies link costs that imply that the requirement holds over the paths in \( S \). However, it may not hold over all paths in \( \text{Paths}(r) \). The idea of CEGIS is to check the requirement over all paths and to obtain a concrete counter-example that violates it, if one exists; we remark that the step of checking is usually efficient. The set \( S \) is then iteratively expanded with counter-examples until a correct solution is found.

**Algorithm** We show the main steps of our CEGIS algorithm in Alg. 1. For each requirement \( r \in \text{Reqs} \), the algorithm declares a set \( S_r \) (line 3). The algorithm then iteratively repeats the following steps. For each requirement \( r \in \text{Reqs} \), the algorithm samples \( b \) paths from the source to the destination of the requirement \( r \) and adds these to \( S_r \) (line 7). It then encodes the requirement’s satisfaction with respect to \( S_r \) (line 8) and conjoins the result to \( \varphi \) (line 9). If the resulting constraint \( \varphi \) is unsatisfiable, it means the requirements cannot be satisfied and the algorithm returns ⊥ to indicate this. Otherwise, it obtains a model \( M \) of the constraints \( \varphi \) (line 12), which defines a concrete value for each link cost variable.

The algorithm then checks whether these costs defined by \( M \) enforce the requirements \( \text{Reqs} \) (over all paths). If the requirements are satisfied, the algorithm returns \( M(\mathcal{C}) \) (line 14), i.e. it returns the values associated to the link cost variables \( \mathcal{C} \). Otherwise, it obtains a concrete counter-example as a pair \((r, \text{path})\) of a path \( \text{path} \) that violates a requirement \( r \), and expands the set \( S_r \) with \( \text{path} \) (line 15). This ensures that the counter-example is avoided in the next iteration. Further, to reach a solution faster, the algorithm samples additional \( b \) paths for each requirement \( r \) and adds them to \( S_r \). These steps are repeated until a solution is found or the requirements are deemed unsatisfiable.

### 6 Implementation and Evaluation

We implemented NetComplete in around 10K lines of Python code using SMT-LIB v2 [23] and Z3 [21]. Our implementation is based on the theories of linear in-
For simple path requirements, we choose a random feature. We first choose two paths randomly selected source Src and destination Dst pair. For simple path requirements, we choose a random feasible path from Src to Dst. For the other requirements, we first choose two paths \( P_1 \) and \( P_2 \) from Src to Dst and then we construct \( (P_1, P_2) \) for any-path requirements, \( (P_1 = P_2) \) for ECMP, or \( P_1 \gg P_2 \) for ordered requirements. For each topology, we generate multiple sets of requirements of size 2, 8, and 16. We generate all four types of requirements for the OSPF evaluation, and only generate simple and ordered path requirements for the BGP evaluation. Indeed, any-path and ECMP requirements are typically internal requirements and are therefore typically enforced by IGP protocols.

**Sketches** We construct configuration sketches for each topology from a fully concrete configuration (which we synthesize using NetComplete) for which we randomly make a given percentage of the variables symbolic. For instance, to generate partial OSPF (resp. BGP) configurations that are 50% symbolic, we randomly make 50% of the edges (resp. BGP import/export policies) in the synthesized concrete configurations symbolic.

**Validation** We validate that our synthesized configurations comply with the corresponding requirements in an emulated environment composed of Cisco routers [30].

### 6.1 Methodology and datasets

**Topologies** We sample 15 network topologies from Topology Zoo [29] that we classify according to their size: small (from 32 to 34 routers), medium (from 68 to 74 routers), and large (from 145 to 197 routers). We select 5 topologies per category.

**Requirements** We generate four types of routing requirements (simple, any-path, ECMP, and ordered) in each topology. Each requirement is defined between a randomly selected source Src and destination Dst pair. For simple path requirements, we choose a random feasible path from Src to Dst. For the other requirements, we first choose two paths \( P_1 \) and \( P_2 \) from Src to Dst and then we construct \( (P_1, P_2) \) for any-path requirements, \( (P_1 = P_2) \) for ECMP, or \( P_1 \gg P_2 \) for ordered requirements. For each topology, we generate multiple sets of requirements of size 2, 8, and 16. We generate all four types of requirements for the OSPF evaluation, and only generate simple and ordered path requirements for the BGP evaluation. Indeed, any-path and ECMP requirements are typically internal requirements and are therefore typically enforced by IGP protocols.

**Sketches** We construct configuration sketches for each topology from a fully concrete configuration (which we synthesize using NetComplete) for which we randomly make a given percentage of the variables symbolic. For instance, to generate partial OSPF (resp. BGP) configurations that are 50% symbolic, we randomly make 50% of the edges (resp. BGP import/export policies) in the synthesized concrete configurations symbolic.

**Validation** We validate that our synthesized configurations comply with the corresponding requirements in an emulated environment composed of Cisco routers [30].

### 6.2 Results

We now present our results focusing first on OSPF synthesis, before considering BGP synthesis, and finishing with a comparison with SyNET. We run all our experiments on a server with 128GB of RAM and a 12-core dual-processors running at 2.3GHz. Unless indicated, we report averaged results over 5 runs and across topologies of the same class.

**OSPF Synthesis** We first illustrate the effectiveness of synthesizing OSPF configuration using our CEGIS algorithm versus a naive algorithm in which the entire \( \exists \forall \phi \) constraint is directly fed to the solver. We then evaluate how sketches affect synthesis time.

Our results are reported in Fig. 7 and convey four important insights. First, CEGIS significantly outperforms naive OSPF synthesis, especially in large networks where naive synthesis does not even terminate within a day. Second, we see that the synthesis time is proportional to the topology size and the number of requirements. Indeed, the number of symbolic variables...
is equal to the number of symbolic edge costs, while the number of constraints is proportional to the requirements size and the number of available paths. Third, ordered path requirements take more time to synthesize than the other requirements. This is expected as such requirements specify a strict sequence of paths making the search space more sparse. Fourth, the use of more concrete values significantly reduces the synthesis time, especially for ordered path requirements with reductions up to 70%. We further illustrate this behavior in Fig. 8 which depicts the times required to synthesize 16 ordered path requirements for the large networks as a function of the percentage of symbolic values. We see that NetComplete indeed leverages the concrete variables and the reduced search space to synthesize configurations faster.

**BGP Synthesis** We now evaluate the effectiveness of the BGP synthesizer and how it leverages partial evaluation to concretize up to 25% of the symbolic variables and therefore speed up the overall synthesis time.

In Fig. 9, we show the average number of generated symbolic variables for each group (see Appendix C for detailed numbers). We see that the number of generated symbolic variables is not directly related to the topology size and the number of requirements: the number of variables for medium topologies can exceed the ones of larger topologies. For BGP, the number of variables indeed depends on: (i) the number of routers (and their connectivity) in the computed propagation graph; (ii) the complexity of the configuration sketch; and (iii) the effectiveness of partial evaluation.

Regarding partial evaluation, we observe that NetComplete manages to evaluate between 7% and 25% of the generated symbolic variables (Fig. 9), which makes BGP synthesis proportionally faster. Indeed, in Fig. 10, we show how the BGP synthesis time evolves linearly as a function of the number of symbolic variables. We also see that NetComplete always manages to synthesize BGP configurations in less than 14min.

**Comparison to SyNET** We now compare the synthesis time of NetComplete to SyNET. Specifically, we compare NetComplete and SyNET running times for the worst-case scenario reported in [20] involving 10 requirements defined in topologies with 49 and 64 routers. Since SyNET defines requirements in terms of the number of traffic classes and not forwarding paths as NetComplete, we first translate each traffic class to a set of simple path requirements. To ensure a fair comparison, we provide NetComplete with entirely symbolic sketch since SyNET does not accept sketches.

Our results (Fig. 11) shows that NetComplete is at least $600 \times$ faster than SyNET and is able to synthesize configurations for larger topologies that SyNET timed out on. This speed up stems from two factors. First, NetComplete does not use an SMT solver for the requirements that it can solve directly (such as synthesizing static routes). Second, NetComplete relies on domain-specific heuristics (CEGIS and partial evaluation) to reduce the search space, while SyNET relies on the generic optimizations of the underlying SMT solver.

### 7 Related Work

**Intent-based Networking and SDN Languages** The importance of relying on high-level abstractions in network management has received considerable attention, specifically in the context of Software-Defined Networking (SDN) [18, 25, 31, 32, 34, 35, 36, 37]. This influence goes beyond academic with two of the largest SDN controllers (ONOS and OpenDayLight) now providing declarative network management [38, 39].

Our work brings programmability to traditional networks, by enabling operators to enforce policies expressed in high-level SDN-like languages such as Genesi [18] or Frenetic [25]. Our work, therefore, complements the above initiatives and enables them to be used...
Network Verification Network verification approaches are used to check a configuration with respect to requirements. FSR [40] encodes BGP preferences using routing algebra [41] and verifies safety properties (e.g., BGP stability) using SMT solvers. Batfish [6] encodes routing protocols in Datalog and uses Datalog solvers to check conformance with routing requirements. Bagpipe [9] and Minesweeper [8] formalize BGP and present an analyzer for BGP configurations. In contrast, NetComplete focuses on synthesis, which subsumes verification.

Network Configuration Synthesis Recently, multiple works have aimed at synthesizing configurations out of high-level requirements [14, 15, 16, 20].

ConfigAssure [14] supports requirements expressed using first-order constraints. As shown in our evaluation, a direct encoding of routing computations into constraints goes beyond what existing solvers can handle.

Route Shepherd [41, 42] takes a partial specification of BGP preferences and derives constraints over link costs that capture the absence of BGP instability. In contrast, NetComplete models the derivation of BGP preferences and also synthesizes the BGP configuration.

Propane [15] and PropaneAT [16] produce BGP configurations out of high-level requirements. Having the freedom to output any configuration enables these systems to use templates and, in turn, to scale to large networks. In contrast, NetComplete supports partial configurations for multiple protocols (OSPF, BGP, and static routes), which prevents us from leveraging specific templates. While NetComplete pays for this flexibility in terms scalability, it is still fast, synthesizing configurations within seconds.

SyNET [20] is another network-wide configuration synthesizer supporting multiple protocols. It differs from NetComplete in two ways. First, SyNET supports any protocol that can be specified in stratified Datalog [43], while NetComplete supports specific protocols (OSPF, BGP). Since BGP cannot be fully captured in stratified Datalog, SyNET supports a simplified BGP though, while NetComplete supports it fully. Second, SyNET uses a generic synthesis procedure for Datalog, while NetComplete uses custom procedures for each protocol. Consequently, SyNET does not scale to large networks and is orders of magnitude slower than NetComplete.

Synthesizers such as NetEgg [19, 44] and NetGen [17] target SDN environments and aim to derive controller programs (instead of configurations) out of requirements. While their goal is similar to ours, our target is different (distributed protocols vs. centralized controller).

Program Synthesis Our work also relates to program synthesis. In particular, we showed a novel instantiation of counter-example guided inductive synthesis (CEGIS) [20] for synthesizing weights in OSPF. CEGIS is a general concept that has become popular in the program synthesis community. A key challenge in using it is finding effective ways to specialize it (e.g., efficient representation of the hypothesis space, interaction with the SMT solver) to the particular application domain (e.g., networking and the OSPF protocol in our case).

8 Conclusion

We presented NetComplete, the first scalable network-wide configuration synthesizer to support multiple protocols and a partial sketch of the desired configuration.

NetComplete features a new BGP synthesis procedure that supports BGP configuration sketches and partial computations over symbolic announcements. It also introduces an efficient synthesis procedure for the widely-used OSPF protocol. This procedure is based on counter-example guided inductive synthesis and achieves significant speedups (> 100x) over existing solutions.

Finally, we presented a comprehensive set of experimental results, which demonstrate that NetComplete can autocomplete configurations for large networks with up to 200 routers within few minutes.

Acknowledgments

We are grateful to our shepherd, Vyas Sekar, and the anonymous reviewers for their constructive feedback.
References


A SMT Encoding of the BGP Selection Process

Encoding Selection of Announcements When a BGP router receives different announcements for the same prefix, it uses the preference ordering to select the best route. We encode the logic used by routers to select the best route in Fig. 12. The predicate <code>PrefNoIGP(A1, A2)</code> holds if and only if an announcement <code>A1</code> is preferred over <code>A2</code> without considering the IGP costs of <code>A1</code> and <code>A2</code>. The constraint is a disjunction over the different cases defined by the BGP selection process. First, if <code>A2</code> is dropped, then <code>A1</code> is selected as a best route. Second, if both announcements are permitted, the router selects <code>A1</code> over <code>A2</code> if <code>A1</code>’s local preference is lower than that of <code>A2</code>. Analogously, the constraint encodes cases 3 – 5 described in §4.4.

In addition, we define the constraint <code>Pref(A1, A2)</code> that also compares the announcements’ IGP costs. The constraint holds if announcement <code>A1</code> is preferred over <code>A2</code> without considering IGP costs (i.e., <code>PrefNoIGP(A1, A2)</code> holds) or the IGP cost of <code>A1</code> is lower than that of <code>A2</code>.

B SMT Encoding of BGP sketches

We illustrate the encoding of BGP sketches using SMT constraints. Consider the following BGP sketch:

```plaintext
1 AttributesSketch
2  match next-hop AS200
3  set local-pref ? < 50
```

This sketch would match any announcement that has the value AS200 set for the next hop attribute. If an announcement is matched, this policy would set the local preference of the output announcement to a value that is yet to be synthesized by the BGP synthesizer. As defined by the sketch, this local preference value must be smaller than 50. Note that this BGP policy does not change the remaining attributes (as there are no further actions).

We encode this BGP sketch as follows:

```plaintext
if NextHopAin = AS200
  then (LocalPrefAout = Var1) \land (0 < Var1 < 50)
    \land (\forall X \in Attrs \setminus \{LocalPref\}, X_{Aout} = X_{Ain})
  else \forall X \in Attrs. X_{Aout} = X_{Ain}
```

where <code>Attrs = \{NextHop, \ldots\}</code> and <code>Var1</code> are fresh variables.

Here, the variable <code>Var1</code> represents the local preference value that will be set by the BGP policy. <code>Ain</code> represents the input announcement (before it is processed by the BGP policy) and <code>Aout</code> the output one. The constraint formalizes that only input announcements with next hop equal to AS200 are matched. For matched announcements, the <code>then</code> constraint encodes that the output announcement has local preference set to <code>Var1</code>, which is a value smaller than 50, and all remaining attributes are identical to those in the input announcement (and thus remain unchanged). Finally, the <code>else</code> constraint ensures that if an announcement is not matched (its local preference is not AS200), then all attributes remain unchanged.

C Symbolic Variables in BGP Synthesis

In Fig. 13 we show the number of generated symbolic variables when synthesizing BGP configurations for each topology we used in our data set. We observe that the number of generated variables depends on the number of routers (and their connectivity) in the computed propagation graph, the complexity of the configuration sketch, and on the effectiveness of partial evaluation.
<table>
<thead>
<tr>
<th>Size</th>
<th>Topo</th>
<th>Req. type</th>
<th>2 reqs.</th>
<th>8 reqs.</th>
<th>16 reqs.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Total</td>
<td>Total</td>
<td>Total</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>100%</td>
<td>50%</td>
<td>100%</td>
</tr>
<tr>
<td>Small</td>
<td>Arnes</td>
<td>simple</td>
<td>1997</td>
<td>13565</td>
<td>68355</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order</td>
<td>962</td>
<td>2466</td>
<td>35993</td>
</tr>
<tr>
<td></td>
<td>Bics</td>
<td>simple</td>
<td>2045</td>
<td>17960</td>
<td>51890</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order</td>
<td>758</td>
<td>2592</td>
<td>40581</td>
</tr>
<tr>
<td></td>
<td>Canerie</td>
<td>simple</td>
<td>764</td>
<td>16451</td>
<td>55787</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order</td>
<td>1238</td>
<td>2569</td>
<td>40243</td>
</tr>
<tr>
<td></td>
<td>CrINet</td>
<td>simple</td>
<td>653</td>
<td>13112</td>
<td>41693</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order</td>
<td>854</td>
<td>2078</td>
<td>27607</td>
</tr>
<tr>
<td></td>
<td>Renater</td>
<td>simple</td>
<td>2717</td>
<td>18983</td>
<td>75167</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order</td>
<td>758</td>
<td>2884</td>
<td>43884</td>
</tr>
<tr>
<td>Medium</td>
<td>Columbus</td>
<td>simple</td>
<td>2057</td>
<td>13400</td>
<td>52672</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order</td>
<td>854</td>
<td>2520</td>
<td>28545</td>
</tr>
<tr>
<td></td>
<td>Esnet</td>
<td>simple</td>
<td>2324</td>
<td>28403</td>
<td>105879</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order</td>
<td>1526</td>
<td>4206</td>
<td>51135</td>
</tr>
<tr>
<td></td>
<td>Latnet</td>
<td>simple</td>
<td>5111</td>
<td>44012</td>
<td>149394</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order</td>
<td>3530</td>
<td>4288</td>
<td>104606</td>
</tr>
<tr>
<td></td>
<td>Sinet</td>
<td>simple</td>
<td>1139</td>
<td>33905</td>
<td>114648</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order</td>
<td>2966</td>
<td>4282</td>
<td>77823</td>
</tr>
<tr>
<td></td>
<td>Uninet2011</td>
<td>simple</td>
<td>2705</td>
<td>24275</td>
<td>70821</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order</td>
<td>1610</td>
<td>3317</td>
<td>32511</td>
</tr>
<tr>
<td>Large</td>
<td>Cogentco</td>
<td>simple</td>
<td>3293</td>
<td>22565</td>
<td>85708</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order</td>
<td>1046</td>
<td>4420</td>
<td>29726</td>
</tr>
<tr>
<td></td>
<td>Colt</td>
<td>simple</td>
<td>3578</td>
<td>47795</td>
<td>85997</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order</td>
<td>1046</td>
<td>6087</td>
<td>29726</td>
</tr>
<tr>
<td></td>
<td>GotoCe</td>
<td>simple</td>
<td>3566</td>
<td>9992</td>
<td>33887</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order</td>
<td>854</td>
<td>2544</td>
<td>31073</td>
</tr>
<tr>
<td></td>
<td>TataNld</td>
<td>simple</td>
<td>2348</td>
<td>29627</td>
<td>67705</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order</td>
<td>662</td>
<td>4948</td>
<td>31424</td>
</tr>
<tr>
<td></td>
<td>UsCarrier</td>
<td>simple</td>
<td>1460</td>
<td>19643</td>
<td>103731</td>
</tr>
<tr>
<td></td>
<td></td>
<td>order</td>
<td>758</td>
<td>3776</td>
<td>21715</td>
</tr>
</tbody>
</table>

Figure 13: The number of symbolic variables generated for each topology and the number of partially evaluated variables when the configuration sketch is 100% and 50% symbolic.