Paper Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes]
   (c) Did you discuss any potential negative societal impacts of your work? [N/A]
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes]
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes]
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes]
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes]
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [N/A]
   (b) Did you mention the license of the assets? [N/A]
   (c) Did you include any new assets either in the supplemental material or as a URL? [Yes]
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

Appendix

A Implementation and Model Details

A.1 Structural Fact Base Embedding

To transform a fact base into a graph with node features, we employ a structurally-defined embedding scheme. Given a fact base $\mathcal{F}$, we apply the rules in Figure 6 to obtain node features and adjacency information. In the resulting graph, both a fact f and a constant c are represented as distinct nodes with features h_c and h_f, respectively. The relationships between facts and constants is encoded as node adjacency as defined by the neighborhood functions N_i. We use multiple neighborhood functions, i.e. multiple types of edges, to encode the argument position of a constant occurring in a fact. Hence, the number of neighborhood functions corresponds to the maximum number of arguments in a single fact, e.g. 3 in Figure 4. Integer arguments and unary facts are directly accounted for in the respective node or fact embedding. To handle unknown parameters, we replace the corresponding $emb(...)$ term with the learned $\mathbf{V_{hole}}$ embedding.

Considering the embedding function emb, the learned parameters of our embedding are $\mathbf{V_f} \in \mathbb{R}^D$ per fact type in $\mathcal{F}$, $\mathbf{W}_{\text{bool}} \in \mathbb{R}^{D\times 2}$ for boolean values, $\mathbf{W_{f_i}} \in \mathbb{R}^{D\times N}$ per integer argument of fact type f and $\mathbf{V_{hole}} \in \mathbb{R}^D$ to represent unknown parameters. Hyperparameter D represents the dimension of the latent space the synthesizer model operates in and N specifies the number of supported integer values.

A.2 Graph Attention Layer

The recurrence relation of a node j's representation h_j is defined as:

$$
\begin{aligned}
z_j &:= h_j + Drop(\sum_{N_i} h'_{GAT_i}) \\
h'_j &:= FFN(Norm(z_j)) \\
\\
FFN(x_j) &:= Norm(Lin2(Drop(Lin1(x_j)))))
\end{aligned}
$$

h'_{GAT_i} represents the output of a graph attention layer operating with neighborhood N_i according to [42], h_j the representation of node j at the previous step. $Norm$ refers to batch normalization [24], $Drop$ to dropout regularization [22] and $Lin1$ and $Lin2$ to linear layers with an inner dimension of $4N$. The mechanic of combining the results of multiple graph attention layers h'_{GAT_i} directly corresponds to how multiple attention heads are combined in the original Transformer architecture [40].

$$
\begin{aligned}
&emb(f) := \mathbf{V_f} && \textit{(learned fact type embedding)} \\
&emb(b) := \mathbf{W}_{\text{bool}}\ onehot(b) && \textit{(learned boolean embedding)} \\
&emb(f,i,v) := \mathbf{W_{f_i}}\ onehot(v) && \textit{(learned integer embedding)} \\
&emb(f,i,?) := \mathbf{V_{hole}} && \textit{(learned hole embedding)} \\
&h_c := \sum_{f(c)\in\mathcal{F}} emb(f) && \textit{(constant embedding)} \\
&h_{f(a_0,\ldots,a_n)} := emb(f) + emb([f(a_0,\ldots,a_n)]_{\mathcal{B}}) + \sum_{(i,v)\in\mathcal{I}(f(a_0,\ldots,a_n))} emb(f,i,v) && \textit{(fact embedding)} \\
&\mathcal{I}(f(a_0,\ldots,a_n)) := \{(i,v) \mid f(a_0,\ldots,a_{i-1},v,\ldots,a_n)\in\mathcal{F} \wedge v : Int \wedge i\in\mathbb{N}\} && \textit{(integer arguments)} \\
&N_i(h_c) := \{h_f \mid \exists i\in\mathbb{N}. f(a_0,\ldots,a_{i-1},c,a_{i+1},\ldots,a_n)\in\mathcal{F}\} && \textit{(constant node neighbors)} \\
&N_i(h_f) := \{h_c \mid \exists i\in\mathbb{N}. f(a_0,\ldots,a_{i-1},c,a_{i+1},\ldots,a_n)\in\mathcal{F}\} && \textit{(fact node neighbors)}
\end{aligned}
$$

Figure 6: Translating a fact base $\mathcal{F}$ to node features. f denotes a fact type, v integer values, b boolean values as 0 or 1, i argument indices and $[f(a)]_{\mathcal{B}}$ the truth value of a fact.

A.3 Dataset Generation and Training

To train a synthesizer model, we need a large number of fact bases encoding a variety of network topologies, configurations and forwarding specifications. Further, we must provide the model with target values for unknown parameters as a supervision signal. This section discusses how we construct such a dataset for BGP/OSPF synthesis and train a corresponding synthesis model.

Dataset Generation To obtain network configurations and corresponding forwarding specifications we employ a generative process: We first randomly configure BGP and OSPF parameters for some topology. Then we simulate the OSPF and BGP protocol and compute the resulting forwarding plane. Based on this, we extract a forwarding specification which fixes a random subset of forwarding paths, reachability and isolation properties that are satisfied by the forwarding plane.

Topologies For training, our dataset is based on random topologies with 16-24 routers. We generate the physical layout of these graphs by triangulation of uniformly sampled points in two-dimensional space.

Simulation To compute the forwarding plane given an OSPF and BGP configuration, we simulate the protocols. For OSPF, we implement shortest path computation to obtain the forwarding plane. For BGP, we implement the full BGP decision process as shown in **??**, except for the MED attribute. In addition to the basic networking model discussed in Section 3, we implement support for eBGP and iBGP, fully-meshed BGP session layouts as well as layouts relying on route reflection. For more details of these BGP concepts see [3].

Fact Base Encoding We encode synthesis input using a small set of facts. For an example of a corresponding fact base see Figure 7. We use the `router`, `external` and `network` facts to mark routers, external peers and routing destinations respectively. Physical links between routers are encoded as `connected(R1,R2,W)` facts where `W` denotes the OSPF link weight. Further, we rely on `ibgp`, `ebgp`, `route_reflector` and `bgp_route` facts to represent different types of BGP sessions and imported routes. Using the notion of unknown parameters as discussed in Section 4.2, we encode configuration parameters, e.g., properties of imported routes or OSPF link weights.

Specification Language We support a small specification language with three types of forwarding predicates:

- `fwd(R1, Net, R2)` specifies that router `R1` forwards traffic destined for `Net` to its neighbor `R2`.
- `reachable(R1, Net, R2)` specifies that traffic destined for `Net` that passes through `R1` also passes through `R2` before reaching its destination.
- `trafficIsolation(R1, R2, N1, N2)` specifies that only one of $[\texttt{fwd(R1,N1,R2)}]_{\mathcal{B}}$ and $[\texttt{fwd(R1,N2,R2)}]_{\mathcal{B}}$ can be true at a time.

In the negative case of each predicate, the opposite must hold true for the fact to be satisfied. This specification language can easily be extended by including new specification predicates in the dataset generation process. One must simply provide a method of extracting positive and negative cases of a specification fact from a given forwarding plane. Once included in the training dataset of a synthesizer model, the resulting synthesis system will be able to consider specifications relying on the new type of predicates.

Training We generate a dataset of $10,240$ input/output samples and instantiate our synthesizer model with a hidden dimension $D = 64$ and supported parameter values $N = 64$. For optimization, we use the Adam optimizer [27] with a learning rate of 10^{-4}. We stop training after the specification consistency on a validation dataset no longer increases at ~ 2800 epochs.

```
# topology and configuration
router(c5)
router(c3)
router(c1)
router(c0)
network(c6)
network(c7)

external(c8)
external(c9)
external(c10)
external(c11)
external(c12)
external(c13)

route_reflector(c2)
route_reflector(c4)

connected(c2,c4,?)
connected(c2,c5,?)
connected(c2,c3,?)
connected(c4,c5,?)
connected(c4,c3,?)
connected(c4,c0,?)
connected(c5,c1,?)
connected(c5,c0,?)
connected(c3,c0,?)
connected(c3,c1,?)
connected(c1,c0,?)

ibgp(c2,c4)
ibgp(c2,c5)
ibgp(c2,c0)
ibgp(c2,c1)
ibgp(c4,c3)
ibgp(c4,c4)
```

```
ebgp(c2,c12)
ebgp(c5,c9)
ebgp(c5,c10)
ebgp(c5,c13)
ebgp(c0,c8)
ebgp(c0,c11)

bgp_route(c8,c6,?,?,1,?,1,8)
bgp_route(c9,c6,?,?,0,?,1,9)
bgp_route(c10,c6,?,?,0,?,1,10)
bgp_route(c11,c7,?,?,0,?,1,11)
bgp_route(c12,c7,?,?,0,?,1,12)
bgp_route(c13,c7,?,?,2,?,1,13)

# forwarding specification
fwd(c0,c6,c3)
fwd(c3,c6,c2)
fwd(c2,c6,c4)
not fwd(c3,c6,c0)
not fwd(c0,c6,c11)
not fwd(c11,c6,c0)

not trafficIsolation(c1,c0,c7,c6)
not trafficIsolation(c1,c0,c7,c6)
trafficIsolation(c0,c3,c6,c7)
trafficIsolation(c0,c3,c6,c7)

reachable(c5,c6,c10)
reachable(c1,c6,c4)
not reachable(c0,c6,c5)
reachable(c5,c6,c10)
```

Figure 7: An example of a fact base encoding a topology, a sketch of a BGP and OSPF configuration and a forwarding specification.

BGP Decision Process For completeness, we include the BGP decision process as assumed for our synthesis setting. The shown rules are applied in order until a single best BGP announcements remains.

1. Highest Local preference
2. Lowest AS path length
3. Prefer Origin (IGP > EBGP > INCOMPLETE)
4. Lowest Multi-exit discriminator (MED)
5. External over internal announcements
6. Lowest IGP cost to egress
7. Lowest BGP peer id (tie breaker)

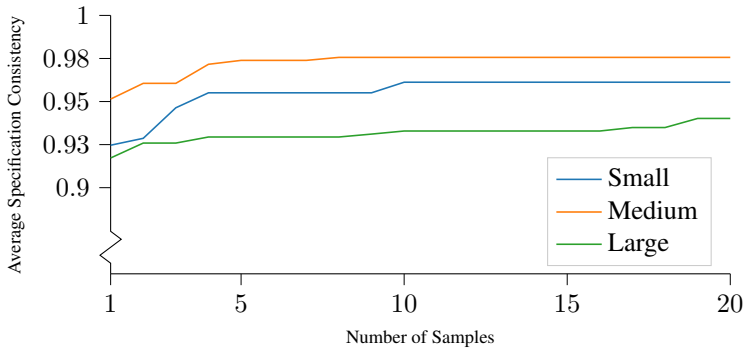

Figure 8: Average Best Specification Consistency with increasingly many samples from our synthesizer model with 3×16 BGP/OSPF requirements.

B More Evaluation Results

B.1 Number Of Samples

Figure 8 shows the average best consistency across our three datasets with an increasing number of samples for 3×16 BGP/OSPF requirements. As the number of samples increases, the graph indicates the best consistency value reached so far. We observe that sampling more than once improves the average specification consistency across all datasets. With increasing size of the topology, more samples appear to be necessary for the resulting best specification consistency to converge. Therefore, we note that increasing the number of samples can improve consistency but it will also affect overall synthesis time as the model needs to be executed again for each sample. In our other experiments we rely on 4/5 samples per specification, which we consider a good trade-off of fast synthesis time and good specification consistency.

B.2 Unsatisfiable Specifications

To asses how our model performs in the presence of unsatisfiable specifications, we evaluate its synthesis performance on multiple datasets of unsatisfiable OSPF synthesis tasks.

Specifications We construct the datasets UNSAT-N, where $N \in \{2, 4, 6, 8, 10, 12, 14, 16\}$, by first generating 16 solvable OSPF forwarding path requirements per topology in Small, Medium and Large. Then we replace N of the requirements per topology with paths obtained under other, random link weights. We repeat this process up to five times, until we can verify that all variants of a topology as contained in the different UNSAT-N datasets are indeed unsatisfiable using the SMT-based synthesizer NetComplete [15]. Overall, this leaves us with 8 UNSAT-N datasets of OSPF synthesis tasks, where we expect the maximum achievable specification consistency to decrease with increasing N. For some topologies, we were not successful in generating unsatisfiable variants for all N and therefore we removed them from the considered set of topologies for this part of our evaluation. We count 15 topologies per UNSAT-N dataset, where the number of nodes ranges between 16 and 153.

Methodology For synthesis, we use the same sampling configuration as in Section 5.1 where we again report the best consistency value across 5 different runs of our synthesis model per synthesis task. We further configure our trained synthesizer model to do OSPF synthesis by only predicting link weights.

Results We report the results of applying our synthesizer model to unsatisfiable OSPF specifications in Figure 9. For comparison, we also include the results of applying our synthesizer model to an UNSAT-0 dataset, i.e. a dataset of satisfiable OSPF synthesis tasks. The average specification consistency drops with unsatisfiable specifications which is expected since the theoretical upper bound for specification consistency is lower with conflicting requirements. At the same time, specification consistency remains high, which suggests that our model still attempts to maximize specification consistency, even when provided with an unsatisfiable specification with many conflicting requirements.

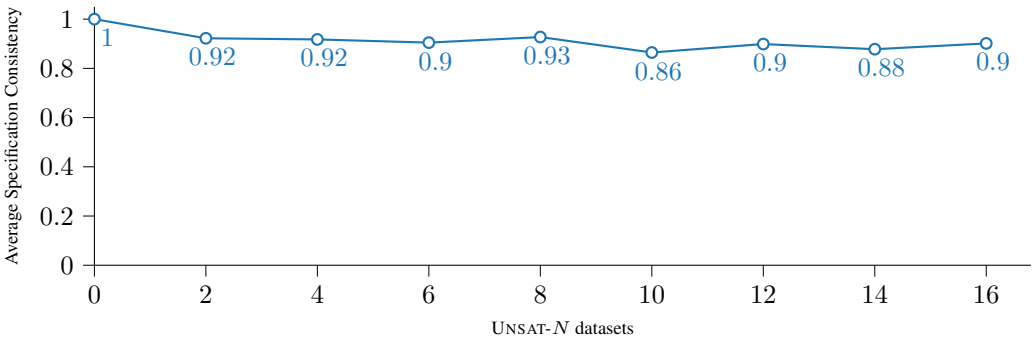

Figure 9: Average OSPF specification consistency of our synthesizer model when applied to unsatisfiable specifications.

In contrast, SMT-based synthesizers like NetComplete will not be able to produce any configurations for the synthesis tasks in our UNSAT-N datasets.

B.3 BGP/OSPF Synthesis Time (GPU)

In the following table we report BGP/OSPF consistency and synthesis time of our method (Neural) running on a GPU, compared with the SMT-based state-of-art synthesis tool NetComplete. The notation $_{n/8\ \text{TO}}$ indicates the number of timed out synthesis runs out of 8 (25+ minutes). Dataset Large is omitted due to GPU memory restrictions.

BGP Synthesis Time (GPU, 5 samples)

		Synthesis Time (s)			Accuracy (Neural)	
# Reqs.		NetComplete	Neural (GPU)	Speedup	∅ Consistency	∅ No. Full Matches
2 reqs.	S	18.07s±14.55	0.44s±0.26	**41.1x**	1.00±0.00	8/8
	M	60.86s±33.39	0.74s±0.69	**81.8x**	0.94±0.13	6/8
8 reqs.	S	247.69s±436.90	0.91s±0.70	**270.7x**	0.95±0.08	5/8
	M	>25m $_{8/8\ \text{TO}}$	1.23s±0.72	**1220.6x**	0.97±0.04	4/8
16 reqs.	S	1416.83s±235.25 $_{7/8\ \text{TO}}$	1.55s±0.51	**913.3x**	0.92±0.06	1/8
	M	>25m $_{8/8\ \text{TO}}$	1.65s±0.51	**906.5x**	0.93±0.05	1/8

B.4 OSPF Synthesis Time

We also compare consistency and synthesis time of our method (Neural) with the SMT-based synthesis tool NetComplete doing OSPF-only synthesis. The notation $_{n/8\ \text{TO}}$ indicates the number of timed out synthesis runs out of 8 (25+ minutes). For this experiment, our neural synthesizer can run on the GPU, as OSPF-only synthesis consumes less GPU memory.

Below, we report results of OSPF-only synthesis sampling from the synthesizer model 4 times and 5 times respectively. We observe that sampling 4 times can be enough with smaller specifications to obtain full matches for all synthesis tasks. For larger specifications and networks, sampling more than 4 times can improve average consistency. Note however, that sampling 5 times for small specifications and topologies can lead to synthesis times that are longer than with NetComplete (0.7x with 2 reqs., 5 samples, dataset S).

OSPF Synthesis Time (4 samples)

		Synthesis Time (s)			Accuracy (Neural)	
		NetComplete	Neural (GPU)	Speedup	∅ Consistency	∅ No. Full Matches
2 reqs.	S	0.27s±0.09	0.09s±0.00	**2.9x**	1.00±0.00	8/8
	M	0.40s±0.09	0.10s±0.00	**4.1x**	1.00±0.00	8/8
	L	0.94s±0.33	0.14s±0.08	**6.7x**	1.00±0.00	8/8
8 reqs.	S	0.88s±0.18	0.16s±0.12	**5.3x**	0.98±0.04	6/8
	M	1.52s±0.40	0.10s±0.01	**14.7x**	1.00±0.00	8/8
	L	3.50s±1.11	0.32s±0.15	**11.1x**	0.98±0.03	4/8
16 reqs.	S	1.65s±0.37	0.21s±0.14	**7.9x**	0.99±0.01	5/8
	M	2.65s±0.70	0.25s±0.16	**10.8x**	0.99±0.01	4/8
	L	566.64s±772.90 $_{3/8\ \text{TO}}$	0.23s±0.11	**2430.4x**	0.99±0.04	7/8

OSPF Synthesis Time (5 samples)

		Synthesis Time (s)			Accuracy (Neural)	
		NetComplete	Neural (GPU)	Speedup	∅ Consistency	∅ No. Full Matches
2 reqs.	S	0.27s±0.09	0.36s±0.01	**0.7x**	1.00±0.00	8/8
	M	0.40s±0.09	0.37s±0.00	**1.1x**	1.00±0.00	8/8
	L	0.94s±0.33	0.52s±0.21	**1.8x**	1.00±0.00	8/8
8 reqs.	S	0.88s±0.18	0.70s±0.63	**1.3x**	0.98±0.04	6/8
	M	1.52s±0.40	0.37s±0.00	**4.1x**	1.00±0.00	8/8
	L	3.50s±1.11	1.69s±1.08	**2.1x**	0.98±0.02	4/8
16 reqs.	S	1.65s±0.37	0.81s±0.69	**2.0x**	0.99±0.02	6/8
	M	2.65s±0.70	1.01s±0.73	**2.6x**	0.99±0.01	5/8
	L	566.64s±772.90 $_{3/8\ \text{TO}}$	1.09s±0.89	**518.1x**	0.99±0.04	6/8

C Future Research Directions

The key challenge with our learning-based system remains its imprecision, i.e. that it produces only partially-consistent configurations in some cases. On one hand, parts of the configuration synthesis problem are NP-hard [7, 16, 44], which means that practical and scalable synthesis tools have to compromise on precision unless P=NP. At the same time, satisfying the complete specification may be crucial, depending on the scenario. Therefore, our learning-based synthesis will not be applicable in the same way as precise, SMT-based synthesizers. Based on this insight, we envision a whole range of future directions for which imprecise, learning-based synthesis will be very useful:

ML-assisted Configuration Imprecise synthesis may be used to enable ML-assisted configuration in the form of a semi-automated process: users provide a specification, apply the synthesizer and perform additional tweaking after they obtain a sufficiently consistent configuration. Optionally, our synthesizer model can also be applied again to predict alternative values for some of the configuration parameters, by masking only the desired parameters in a configuration (cf. Section 4.2). Recently, similar systems for code completion, like GitHub CoPilot [11, 20], have demonstrated that this query-predict-modify workflow can be highly effective at combining an imprecise ML-guided recommendation system with user interaction [13].

Hybrid Systems Given that further tweaking of a configuration may be necessary, future work may also explore seeding SMT-based, partial synthesis methods [15] or configuration repair methods [17] with generated candidate solutions. This can further automate the process of obtaining fully consistent configurations while maintaining some of the performance benefits of a learning-based system. Existing work on partial synthesis has already shown that SMT-based synthesis can be much faster when a rough sketch of a configuration is already provided as an input [15].

Imprecision in SMT-based synthesis Tweaking the result of a synthesizer is undesirable, but it is also a common practice with existing SMT-based synthesis, as it can be imprecise too. This can be caused by hand-coded SMT synthesis rules which do not always hold up in practice, given the complex behavior of real-world routers [6]. Similarly, our learning-based method is imprecise, but could actually be trained on real-world data, possibly bridging the gap between an assumed formal model and real-world behavior. Future work is necessary to clarify how the configuration tweaking process differs in practice between SMT-based and learning-based synthesis.

Optimality with Unsatisfiable Specifications As demonstrated in Appendix B.2, our learning-based system produces highly consistent configurations even in the presence of an unsatisfiable specification. In practice, this can be helpful as operators do not know whether their specification is unsatisfiable ahead of time. In these cases, a partially consistent configuration is the best one can hope for. In contrast, using SMT solvers in this scenario will not produce a configuration at all, while unsatisfiable specifications can be quite difficult to debug.

Service Level Agreements Lastly, the notion of relaxed specification consistency is not entirely unbeknownst to the world of networking. Networks often operate in accordance with so-called Service Level Agreements (SLA), contracts that specify the guaranteed properties of service in terms of relative availability over time. Given this concept of SLAs, partially violated specifications in some scenarios can be tolerable, as explored by previous work like [38].

D Additional Experiments

In response to the reviews we carried out a number of additional experiments. We will integrate these results into our draft where applicable.

D.1 Ablation and Parameter Study

We conducted an ablation study to explore both the effectiveness of our architecture as well as the use of different components (e.g. different GNN layer modules). We compare across the following configurations. Configurations marked with (*) correspond to the configuration presented in the main body of the paper.

Noise and Edge Types NoNoise corresponds to our model without adding gaussian noise before applying the processor network. NoEdgeTypes corresponds to our model with only one edge type. NoiseEdgeTy (*) correspond to our model as presented in the paper.

GNN Modules We also experiment with different GNN modules used internally by the encoder and processor network (cf. Section 4.3). In addition to a graph attention module (GAT, [42]), we also evaluate models that employ a simple message-passing layer (MPNN-max, [19]) and a graph convolutional layer (GCN-max, [28]), both aggregating by maximization.

Hidden Dimensionality We also experiment with different values (16, 32, 64 and 128) for dimensionality D of the synthesizer model.

Separate Latent Spaces According to our graph-based encoding, we embed information on topology, configurations and specification all in a common latent space. However, we also experiment with embedding the topology and configuration information into three separate latent spaces. For this, we train models with three different pairs of encoder+processor networks, one per type of facts/nodes relating to topology, configuration and specification respectively. At each layer and synthesizer iteration, the different encoder/processor GNNs can attend to the intermediate representations of the adjacent nodes according to the underlying fact base graph. As this induces a threefold increase in parameters, we compare the results of such synthesizer models SepSpace-16, SepSpace-64 and our model ComSpace-64 (*). Due to the long training time of these models, we compare after training these configuration for 2000 epochs only.

Sampling, Dataset and Metrics For the examined configurations, we train a synthesizer model using the same training setup as described in A.3. Using the resulting models, we perform synthesis for all datasets S, M, and L, relying on 5 samples per task. For NoNoise and NoiseEdgeTy (*) we additionally perform synthesis for the same datasets, but with 20 samples. Here, the test datasets S, M and L include all generated synthesis tasks of the specification size classes 3x2, 3x8 and 3x16 as used in other parts of our evaluation. Overall, this leads to 24 synthesis tasks on 8 different topologies per dataset S/M/L. We measure average best specification consistency, as well as number of full (Full) and partial, good ($> 90\%$) matches in terms of specification consistency.

Results The results of our experiments are provided in Table 4 and Table 5. For supporting multiple edge types in NoiseEdgeTy, we observe a clear benefit over NoEdgeTypes, both with respect to average best consistency as well as the total number of full and good matches. With just 5 samples, the effect of NoNoise is not very pronounced. However when increasing the number times we sample from the synthesizer model per task, we observe a clear performance improvement. When sampling up to 20 times from the model, NoiseEdgeTy (*) outperforms NoNoise in almost all metrics (cf. Table 5), leading to higher average consistency and a higher number of full and good matches. We hypothesize that adding noise helps the model in producing a wider variety of solutions, eventually leading to better results when selecting the best one.

Regarding the choice of GNN module, we observe that GAT is the most effective, but MPNN-max can also be a good choice.

Regarding the choice of hidden dimension, we observe that 64 is a good balance of performance and memory usage, while 128 only brings slight improvements. For our purposes we selected $D = 64$, since this can improve synthesis time significantly when running on a GPU (cf. Appendix B.3). For

Ablation Results (5 samples)

Configuration	S	Full	> 90%	M	Full	> 90%	L	Full	> 90%
NOISEEDGETY (*)	**0.97±0.05**	**15/24**	**20/24**	**0.96±0.05**	**12/24**	**19/24**	0.95±0.03	7/24	18/24
NONOISE	0.96±0.07	**15/24**	19/24	0.95±0.05	10/24	**19/24**	**0.96±0.04**	**8/24**	**21/24**
NOEDGETYPES	0.94±0.07	11/24	18/24	0.93±0.08	9/24	17/24	¸0.92±0.07	6/24	16/24
GAT [42] (*)	**0.96±0.05**	12/24	**21/24**	**0.96±0.06**	**12/24**	**20/24**	**0.94±0.05**	**6/24**	**18/24**
MPNN-Max [19]	**0.96±0.05**	**14/24**	20/24	0.95±0.05	9/24	18/24	0.93±0.05	5/24	14/24
GCN-Max [28]	0.93±0.09	11/24	16/24	0.94±0.08	9/24	17/24	0.92±0.06	4/24	14/24
Hidden Dim 16	0.96±0.05	14/24	19/24	0.93±0.07	7/24	18/24	0.90±0.08	4/24	11/24
Hidden Dim 32	0.96±0.05	12/24	18/24	0.94±0.06	8/24	18/24	0.94±0.07	6/24	18/24
Hidden Dim 64 (*)	**0.97±0.05**	**15/24**	20/24	**0.96±0.05**	**12/24**	19/24	**0.95±0.03**	7/24	18/24
Hidden Dim 128	**0.97±0.05**	**15/24**	19/24	**0.96±0.05**	**12/24**	**20/24**	**0.95±0.06**	**9/24**	**20/24**
COMSPACE-64 (*)	**0.96±0.05**	14/24	**20/24**	**0.95±0.06**	**11/24**	19/24	0.95±0.05	8/24	19/24
SEPSPACE-64	0.95±0.09	**15/24**	**20/24**	**0.95±0.06**	10/24	18/24	**0.96±0.03**	**9/24**	**21/24**
SEPSPACE-16	0.94±0.07	10/24	16/24	**0.95±0.05**	**11/24**	**21/24**	0.93±0.07	6/24	17/24

Table 4: The main results of our ablation and parameter study.

Ablation Results (20 samples)

Configuration	S	Full	> 90%	M	Full	> 90%	L	Full	> 90%
NOISEEDGETY (*)	**0.98±0.03**	16/24	**22/24**	**0.99±0.03**	**16/24**	**22/24**	**0.97±0.03**	**12/24**	**23/24**
NONOISE	0.97±0.06	**18/24**	**22/24**	0.97±0.04	12/24	21/24	0.96±0.05	9/24	19/24

Table 5: Synthesis performance when sampling 20 times from a synthesizer model with and without adding noise (NOISEEDGETY (*) vs. NONOISE).

$D = 128$ this may not always be possible, especially when limited to a single GPU and working with large topologies.

Lastly, we cannot observe a significant benefit of separating the latent spaces of topology, configuration and specification as described above using multiple encoder+processer networks. SEPSPACE-64 appears to perform mostly on par with our COMSPACE-64 configuration, while having significantly more parameters. SEPSPACE-16 is more comparable in terms of the number of parameters, but it performs worse than COMSPACE-64 in most metrics.

D.2 Dataset Statistics and Distribution Shift

To examine the synthesis performance of our model with real topologies, we source our evaluation datasets from the Topology Zoo [29], a collection of real-world topologies. For training on the other hand, we only rely on synthetic data based on random topologies, configurations and specifications that are easy to generate by simulation (cf. A.3). As a consequence, we observe distribution shift between the synthetic data we train on and the closer-to-real-world data we evaluate on.

Training and Evaluation Datasets Figure 10 illustrates parts of the distribution shift with respect to the number of nodes and specifications size. Our synthetic training dataset (Training Dataset) only contains samples of very limited size (15-25 nodes) with ~20-40 different specification predicates per traffic class. In contrast, our Topology Zoo datasets used for evaluation contain topologies of much larger size (5-153) and larger specifications (~1-60 predicates per traffic class). Considering this distribution shift and the good performance of learned synthesis on the evaluation datasets, our model appears to generalize well to larger topologies/specifications, even without having seen similarly-sized problem instances during training. Comparably strong generalization properties have also been previously observed with other NAR-based models [43].

Specification Distribution Shift (SmallSpec) Our evaluation datasets rely on real topologies but have to resort to synthetic specifications due to the lack of a large, practical dataset in this space. Nonetheless, we want to provide some preliminary insight into the possible effect of a specification

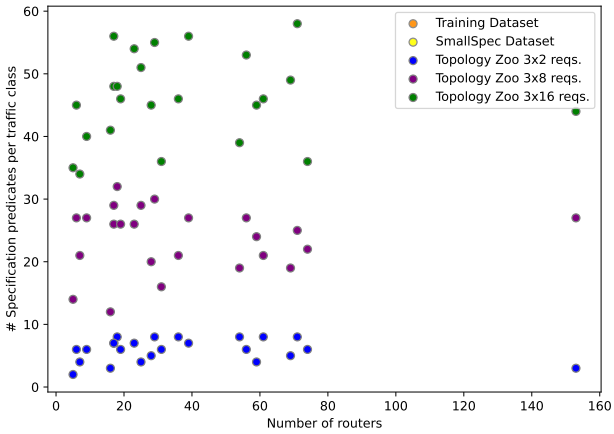

Figure 10: Dataset distribution of network and specification size in terms of routers and the number of specification predicates per traffic class respectively.

SmallSpec Distribution Shift in Training (5 samples)

Configuration	S	Full	> 90%	M	Full	> 90%	L	Full	> 90%
Training Dataset (*)	**0.97±0.05**	**15/24**	20/24	**0.96±0.05**	**12/24**	**19/24**	**0.95±0.03**	**7/24**	18/24
SmallSpec	**0.97±0.04**	14/24	**22/24**	0.95±0.06	10/24	18/24	0.94±0.05	6/24	**19/24**

Table 6: Examining the effect of specification distribution shift by comparing synthesis performance of a model trained on the SmallSpec dataset (smaller specifications) with a model trained on our regular training dataset.

distribution shift, as would be the case when applying our model to real-world synthesis tasks. For this, we construct a SmallSpec dataset (cf. Figure 10) which is similar to our training dataset but with smaller specifications. In Table 6 we report the results of training a synthesizer model on the SmallSpec dataset and comparing its performance with a model trained on our regular Training Dataset. Both models are evaluated on the same Topology Zoo evaluation datasets as in our evaluation. The resulting SmallSpec model achieves slightly lower but comparable synthesis performance. This provides some evidence regarding the robustness of our models when it comes to a distribution shift of the specifications used for synthesis.

D.3 Varying Number of Samples

In a previous revision of the paper, our evaluation relied on 5 samples to obtain consistency results and only 4 to determine synthesis times. In practice, this can be a sensible choice, trading off accuracy for speed. However, to rectify this discrepancy in our results, we carried out the respective dual experiments to also determine average synthesis times and consistency results with 5 and 4 samples respectively. The experiments in the main body of this revision all rely on 5 samples.

Consistency We compare synthesis performance when sampling 4/5 times from our synthesis model. Below we list the results for average best consistency, number of full matches as well as the number of > 90% matches for each of the datasets S, M, L. The results correspond directly to the results in column "Overall" in Table 2, where we report the results for sampling 5 times.

BGP/OSPF Consistency (4 samples vs. 5 samples)

Req.	Samples	S	Full	> 90%	M	Full	> 90%	L	Full	> 90%
3x2	4 samples	**0.96±0.07**	**6/8**	**6/8**	0.91±0.11	4/8	4/8	0.91±0.11	**5/8**	**5/8**
	5 samples	**0.96±0.07**	**6/8**	**6/8**	**0.94±0.08**	**5/8**	**5/8**	**0.94±0.006**	4/8	4/8
3x8	4 samples	**0.97±0.04**	3/8	**7/8**	**0.98±0.02**	3/8	**8/8**	**0.95±0.04**	**1/8**	7/8
	5 samples	0.96±0.04	**4/8**	**7/8**	**0.98±0.03**	**4/8**	**8/8**	**0.95±0.03**	**1/8**	**8/8**
3x16	4 samples	**0.95±0.03**	**2/8**	**8/8**	0.95±0.04	2/8	6/8	**0.95±0.04**	**1/8**	**6/8**
	5 samples	**0.95±0.03**	**2/8**	**8/8**	**0.96±0.04**	**3/8**	**7/8**	0.93±0.05	**1/8**	**6/8**

These results suggest that a higher number of samples can increase average consistency as well as the number of full and good, partial matches. Especially small specifications (cf. 3x2), seem to benefit from more samples. Overall, however there is not a very large difference between sampling 4/5 times.

Synthesis Time We also compare synthesis time of sampling 4/5 times below. We report the synthesis time of running our synthesizer model with 4 samples as reported in the paper. Next, we report synthesis time of running our synthesizer model with 5 samples. Last, we report the speedup over NetComplete when running our synthesizer model with 4 and 5 samples, respectively.

BGP/OSPF Synthesis Time (4 samples vs. 5 samples)

# Requirements		4 samples (s)	5 samples (s)	Speedup (4 samples)	Speedup (5 samples)
2 reqs.	S	0.64s±0.38	0.72s±0.54	**28.2x**	25.2x
	M	2.75s±3.29	3.18s±4.32	**22.2x**	19.1x
	L	22.30s±26.86	24.25s±28.35	**62.3x**	57.3x
8 reqs.	S	1.07s±0.84	1.25s±1.02	**232.5x**	198.7x
	M	3.47s±3.34	4.55s±4.30	**432.2x**	329.8x
	L	30.96s±28.18	31.28s±28.53	**48.4x**	48.0x
16 reqs.	S	2.53s±1.85	2.88s±1.66	**560.5x**	492.0x
	M	5.48s±3.90	6.53s±5.10	**273.7x**	229.8x
	L	69.09s±108.17	87.99s±141.97	**21.7x**	17.0x

Overall, running our synthesizer model with 5 samples means that the model is invoked one more time per synthesis task. This is clearly reflected by the resulting synthesis times. However, as the number of samples is only a linear factor for overall synthesis time, the speedup over NetComplete remains very significant.