Probabilistic Surrogate Networks for Simulators with Unbounded Randomness - Supplementary Material

Andreas Munk[1] **Berend Zwartsenberg**[1] **Adam Ścibior**[2,1] **Atılım Güneş Baydin**[3] **Andrew Stewart**[4]
Goran Fernlund[4] **Anoush Poursartip**[4,5] **Frank Wood**[2,6,1]

[1]Department of Computer Science, University of British Columbia [2]Inverted AI Ltd.
[3]Department of Engineering Science, University of Oxford
[4]Convergent Manufacturing Technologies Inc.
[5]Composites Research Network, University of British Columbia [6]Mila, CIFAR AI Chair

A PROOFS

A.1 PROOF OF THEOREM 1

For an address a define $\mathcal{C}$ and $\mathcal{K}$ as specified in Section 3. That is $\mathcal{C}$ is the set of address transitions we know are possible and $\mathcal{K}$ is the set of newly encountered address transitions found in a sample of traces drawn from a reference simulator. Let $C = |\mathcal{C}|$ and $K = |\mathcal{K}|$ be the size of each set respectively. We consider the set of previous unknown address transitions $\mathcal{U}$ and denote the new set of unknown transitions $\tilde{\mathcal{U}} = \mathcal{U} \setminus \mathcal{K}$. Finally, define the probability measures $\mathbb{P}$ and $\tilde{\mathbb{P}}$ both associated with the sample space Ω and σ-algebra $\mathcal{F}$ according to

$$\mathbb{P}(E) = \frac{1}{Z}\begin{cases} e^{\boldsymbol{v}_{\gamma(c)}}, & \text{if } E = \{c\} \text{ and } c \in \mathcal{C} \\ e^{\boldsymbol{v}_{C+1}} & \text{if } E = \mathcal{U} \end{cases}$$

$$\tilde{\mathbb{P}}(E) = \frac{1}{\tilde{Z}}\begin{cases} e^{\boldsymbol{v}_{\gamma(c)}}, & \text{if } E = \{c\} \text{ and } c \in \mathcal{C} \\ e^{\boldsymbol{v}_{C+1}-\log(K+1)}, & \text{if } E = \{k\} \text{ and } k \in \mathcal{K} \\ e^{\boldsymbol{v}_{C+1}-\log(K+1)}, & \text{if } E = \tilde{\mathcal{U}}, \end{cases}$$

where $\boldsymbol{v} \in \mathbb{R}^{C+1}$, Z and $\tilde{Z}$ are normalization constants, and $\gamma : \mathcal{C} \to \{1, \dots, C\}$ is a mapping from observed addresses to a unique "address index".

Observe that the relationship between $\tilde{\mathbb{P}}$ and $\mathbb{P}$ is equivalent to the relationship between $\mathbb{P}^{\tilde{\zeta}}_{a_t}$ and $\mathbb{P}^{\zeta}_{a_t}$ defined in Section 3. In particular, we consider the functional mapping $h : \mathcal{G} \to \mathcal{G}$ such that $\tilde{\zeta} = h(\zeta)$, where $\tilde{\zeta}, \zeta \in \mathcal{G}$. The proof of Theorem 1 therefore reduces to proving that for all $E \in \mathcal{B} = 2^{\mathcal{C}} \cup \{\mathcal{U}\} \subseteq \mathcal{F}$, $\tilde{\mathbb{P}}(E) = \mathbb{P}(E)$ holds.

We start by comparing the normalization constants:

$$\begin{aligned} \tilde{Z} &= \sum_{c \in \mathcal{C}} e^{\boldsymbol{v}_{\gamma(c)}} + \sum_{k \in \mathcal{K}} e^{\boldsymbol{v}_{C+1}-\log(K+1)} + e^{\boldsymbol{v}_{C+1}-\log(K+1)} \\ &= \sum_{c \in \mathcal{C}} e^{\boldsymbol{v}_{\gamma(c)}} + (K+1)e^{\boldsymbol{v}_{C+1}-\log(K+1)} \\ &= \sum_{c \in \mathcal{C}} e^{\boldsymbol{v}_{\gamma(c)}} + e^{\boldsymbol{v}_{C+1}} \\ &= Z, \end{aligned} \tag{1}$$

leading to,

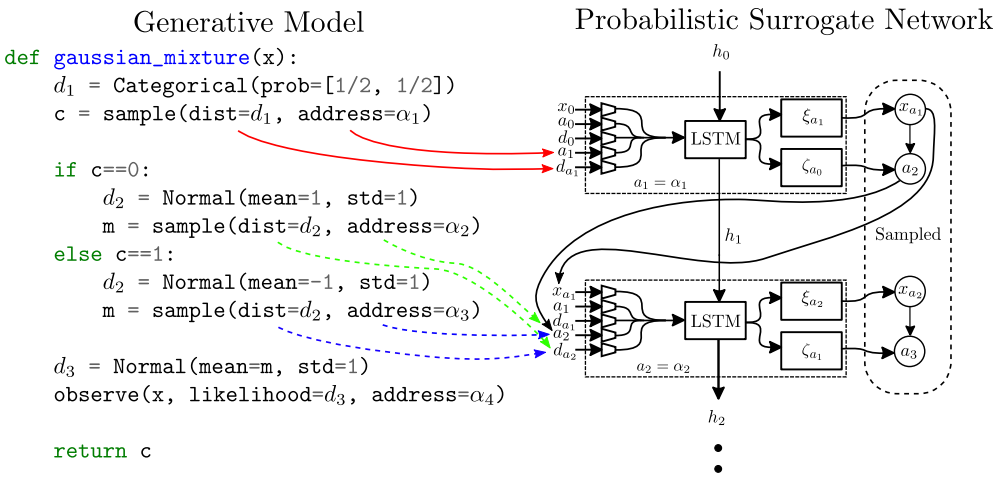

Figure 1: Illustration of the equivalence between a simple generative model and a probabilistic surrogate network. The red arrows represent what is extracted from the program and fed to the surrogate network during training. Generally, this would be an address a and the distribution type d_a at that address. This extraction happens at every address encountered when executing the program while training the surrogate. The dashed arrows represents possible extractions after one step of running the PSN. Which extraction depends on the sampled value c. If $c = 1$ then $a_2 = \alpha_2$ and the blue dashed arrow extraction happens otherwise $a_2 = \alpha_3$ and the green dashed arrow extraction happens.

$$\tilde{\mathbb{P}}(\{c\}) = \frac{1}{\tilde{Z}} e^{\boldsymbol{v}_{\gamma(c)}} = \frac{1}{Z} e^{\boldsymbol{v}_{\gamma(c)}} = \mathbb{P}(\{c\}) \quad \forall c \in \mathcal{C} \tag{2}$$

$$\begin{aligned} \tilde{\mathbb{P}}(\mathcal{K} \cup \tilde{\mathcal{U}}) &= \tilde{\mathbb{P}}(\mathcal{U}) = \tilde{\mathbb{P}}(\tilde{\mathcal{U}}) + \tilde{\mathbb{P}}(\mathcal{K}) \\ &= \frac{1}{\tilde{Z}} \left(e^{\boldsymbol{v}_{C+1} - \log(K+1)} + \sum_{k \in \mathcal{K}} e^{\boldsymbol{v}_{C+1} - \log(K+1)} \right) \\ &= \frac{1}{Z} e^{\boldsymbol{v}_{C+1}} = \mathbb{P}(\mathcal{U}). \end{aligned} \tag{3}$$

Since all events $\{\{c\} | c \in \mathcal{C}\}$ are mutually exclusive, it follows from Eq. (2) that

$$\tilde{\mathbb{P}}(E) = \sum_{e \in E} \tilde{\mathbb{P}}(\{e\}) = \sum_{e \in E} \mathbb{P}(\{e\}) = \mathbb{P}(E), \quad \forall E \in 2^{\mathcal{C}}. \tag{4}$$

Combining Eq. (3) and Eq. (4), we arrive at the final result,

$$\tilde{\mathbb{P}}(E) = \mathbb{P}(E), \quad \forall E \in \mathcal{B} = 2^{\mathcal{C}} \cup \{\mathcal{U}\},$$

which completes the proof. □

A.2 PROOF OF THEOREM 2

The proof of Theorem 2 only requires the consideration of two possible scenarios regarding a trace $(\boldsymbol{x}, \boldsymbol{a})$: (1) the trace either contains address transitions observed during the training of $s(\boldsymbol{x}, \boldsymbol{a})$ in which case its evaluation is straightforward. (2) $(\boldsymbol{x}, \boldsymbol{a})$ contains addresses and transitions not encountered during training. In the latter case, we would simply expand our PSN to account for those new transitions according to Eq. (8). □

B ALGORITHMS

The procedure we use to expand the address transition distribution at address a_t upon encountering a set of yet unseen transitions $\mathcal{K}_{a_t}$ is outlined in Algorithm 1. The procedure is applied to the final layer of a neural network which follows

an intermediate layer of size n_{emb}. The operation $\mathrm{detach}(\cdot)$ denotes duplication without copying the gradient information, hence detaching the argument from the computational graph. The $\mathrm{concat}(\cdot,\cdot)$ operation concatenates the second argument to the first, and re-attaches the newly created matrix or vector to the computational graph as a leaf.

Algorithm 1: PSN address transitions expansion. Definitions of the detach and concat operations are given in Appendix B

Input: A set $\mathcal{K}_{a_t}$ of new address transitions with size $K = |\mathcal{K}_{a_t}|$
Input: Weights $\boldsymbol{W} \in \mathbb{R}^{(C+1)\times n_{emb}}$ and biases $\boldsymbol{b} \in \mathbb{R}^{C+1}$, with $C = |\mathcal{C}_{a_t}|$
$\boldsymbol{w}^u = \mathrm{detach}\left(\boldsymbol{w}_{C+1}\right)$ // $\boldsymbol{w}_{C+1}$ denotes row $C+1$ of $\boldsymbol{W}$
$b^u = \mathrm{detach}\left(b_{C+1}\right) - \log(1+K)$ // b_{C+1} denotes element $C+1$ of $\boldsymbol{b}$
$\boldsymbol{W} = \boldsymbol{W}_{:C}$ // $\boldsymbol{W}_{:C}$ denotes the first C rows of $\boldsymbol{W}$
$\boldsymbol{b} = \boldsymbol{b}_{:C}$ // $\boldsymbol{b}_{:C}$ denotes the first C elements of $\boldsymbol{b}$
for $k = 0$ **to** $K+1$ **do**
  $\boldsymbol{W} = \mathrm{concat}(\boldsymbol{W}, \boldsymbol{w}^u)$
  $\boldsymbol{b} = \mathrm{concat}(\boldsymbol{b}, b^u)$
end

C SURROGATE NETWORK ARCHITECTURE

The PSN architecture is dynamically constructed during training and uses an LSTM core as well as embeddings of the addresses, distribution types, and other random variables. These embeddings are referred to as a_i, d_i, x_i respectively. In particular, each address is associated with a fixed distribution type. These deterministic and fixed pairings between addresses and distribution types are stored and made part of the surrogate model. In other words, when constructing the PSN we know the distribution type associated with each address. The dynamic construction is driven by the program, where the embeddings are fed to the LSTM core whose output is then fed to so-called "distributions layers" ξ_{a_t} and ζ_{a_t}, that for each unique address a_t produces the parameters for $s(x_{a_t}|\xi_{a_t}(x_{<a_t}, a_{\leq t}, \theta))$ and $s(a_{t+1}|\zeta_{a_t}(x_{<a_{t+1}}, a_{\leq t}, \theta))$ respectively. Note that the value sampled from $s(x_{a_t}|\xi_{a_t}(x_{<a_t}, a_{\leq t}, \theta))$ is additionally fed to ζ_{a_t}. In practice, this means that all conditional probabilities of the PSN are conditioned on the distribution types and therefore their embeddings d_i. While not part of the problem formulation of PSN, as they are not theoretically necessary, we use them as additional inputs to the LSTM as they might help training. This construction is illustrated in Fig. 1. New embeddings and distribution layers are created upon encountering new addresses during training. In practice this is implemented by sweeping through the samples used to calculate the gradient estimator. It is similarly during these sweeps that new address transitions are identified. For each address a_t we construct $\mathcal{K}_{a_t}$ when new address transitions are found. Algorithm 1 is then used for each of those addresses.

When replacing the reference simulator with the PSN, it is initialized using h_0 and embeddings x_0, d_0, and a_0. These initial values are typically set to zero, but could be learnable parameters. The unique first address a_1 (which is guaranteed to be unique as the first point of stochasticity in a program is always the same) is fed to the PSN and the surrogate program starts its execution. At each subsequent time step t the PSN produces a sample x_{a_t} and address a_{t+1}, which then propagates the PSN forward where until an `end-execution` address is sampled. This process is illustrated in Fig. 1.

D EXPERIMENTS

Here we provide various model, training, and validation specifications, along with additional results and evidence that support the claims made in the main paper.

D.1 MODEL SPECIFICATIONS

We largely use the default specifications found in PyProb [Baydin and Le, 2018]. We report the configurations whenever they differ from those default values. We use the same configuration names found in PyProb, so that they can be directly transferable from this paper. A description to each configuration will be given the first time the configuration appears and only when the configuration is not obvious (such as learning rate and optimizer).

D.1.1 Stochastic Control Flow Experiment

Fig. 2 shows learning curves (training and validation) for (a) the PSN and (b) the inference network. For this experiment we continuously generate traces during training in an online fashion. Therefore there is no risk of overfitting to a specific dataset and no validation set is used.

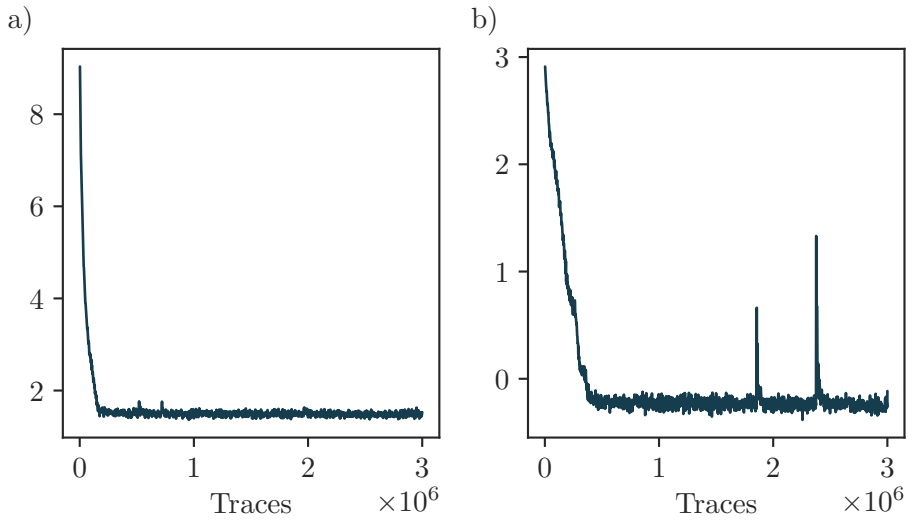

Figure 2: Learning curves for (a) the PSN and (b) the inference network associated with the stochastic control flow experiment.

Table 1: Experiment configuration for the stochastic control flow experiment

Parameter/setting	IC	PSN	Description
Optimizer	Adam	Adam	
Learning rate	5×10^{-4}	5×10^{-4}	
Training data size	500,000	500,000	
Batch Size	512	512	
`sample_embedding_dim`	10	10	The size of each variable embedding
`address_embedding_dim`	24	24	The size of the address embedding which are learnable parameters
`distribution_type_embedding_dim`	24	24	The size of the distribution type embedding which are learnable parameters
`observe_embedding`	{x: {{depth: 4, dim: 10, hidden_dim: 10}}}	N/A	`depth` is the number of linear layers mapping from the value x each with `hidden_dim` number of neurons. The output size (going into the LSTM) is `dim`
`lstm_depth`	1	1	Number of stacked LSTMs
`lstm_dim`	150	150	Size of hidden state in each LSTM
`inf_variable_embedding`	{theta: {{num_layers: 2, hidden_dim: 50}}}	N/A	The names should be self-explanatory and are similar to `observe_embedding` except the input to these layers are the output from the LSTM
`surr_variable_embedding`	N/A	{theta: {{num_layers: 2, hidden_dim: 50}}}	Same meaning as above but for the PSN

D.1.2 Process Simulation of Composite Materials

Fig. 3 shows learning curves (training and validation) for (a) the PSN and (b) the inference network. In this experiment we construct a training set containing 200,000 traces which is iterated through until the number of traces specified in Table 2 has been encountered. The validation set contains 7680 traces.

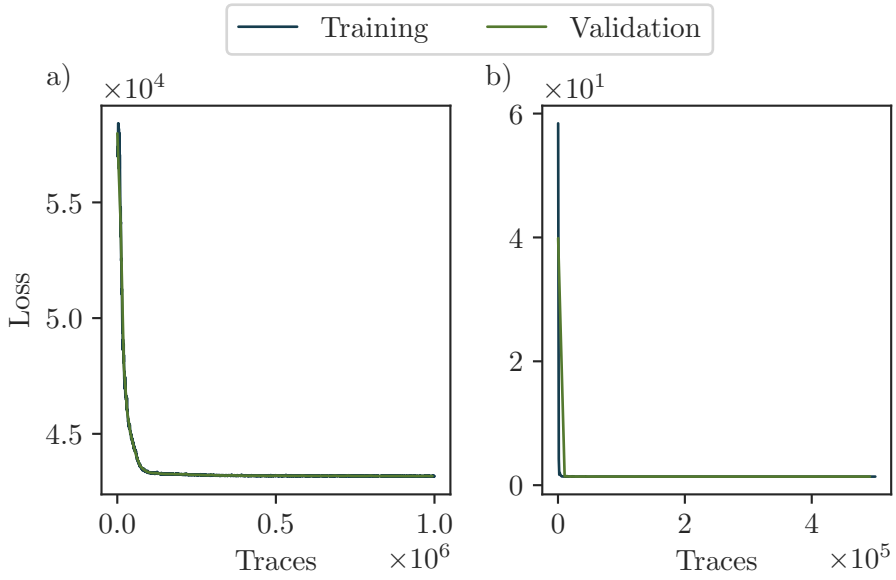

Figure 3: Training and validation learning curves for (a) the PSN and (b) the inference network associated with the process simulation of composite materials experiment.

Table 2: Experiment configuration for the process simulation of composite materials experiment

Parameter/setting	IC	PSN
Optimizer	Adam	Adam
Learning rate	10^{-3}	10^{-4}
Training data size	500,000	1,000,000
Batch Size	256	256
`sample_embedding_dim`	256	256
`address_embedding_dim`	24	24
`distribution_type_embedding_dim`	24	24
`observe_embedding`	{temps_bottom: {depth: 2, dim: 500, hidden_dim: 500}, air_temp_bot: {depth: 2, dim: 500, hidden_dim: 500}, air_temp_top: {depth: 2, dim: 500, hidden_dim: 500}, temps_config: {dim: 10, hidden_dim: 256}}	N/A
`lstm_depth`	2	2
`lstm_dim`	512	512
`inf_variable_embedding`	{config: {{hidden_dim: 256}}}	N/A
`surr_variable_embedding`	N/A	{latent_temps: {{num_layers: 2, hidden_dim: 500}, temps_config: {hidden_dim: 256}}}

D.1.3 Program synthesis Flow Experiment

The configurations used for training the surrogate in the program synthesis experiment are the same as those found in Table 1, while Fig. 4 presents learning curves for the trained surrogate.

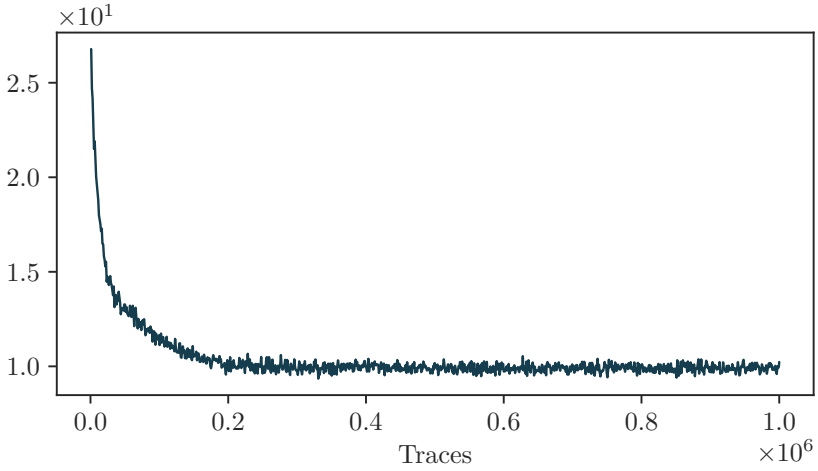

Figure 4: Learning curves for the PSN.

D.2 RUNNING TIMES FOR PROCESS SIMULATION OF COMPOSITE MATERIALS

Table 3: Runtime [traces/s] comparisons. We calculate the number of traces produced per second when (1) running just the simulator or PSN and (2) when performing SIS in either model. We see a slowdown in traces per second for the PSN when performing inference, as the inference engine adds additional overhead. However, as the simulator is considerably slower, it remains the computational bottleneck during inference. The reported run-times are achieved using an Intel(R) Xeon(R) CPU E3-1505M v5 @ 2.80GHz.

	Simulator ($t_{\text{sim}}[\text{traces/s}]$)	PSN ($t_{\text{PSN}}[\text{traces/s}]$)	**Speedup** $[t_{\text{PSN}}/t_{\text{sim}}]$
PSN	0.32	28.87	**90.16**
IC in PSN	0.31	4.75	**15.32**

D.3 RESULTS FOR THE PROCESS SIMULATION OF COMPOSITE MATERIALS EXPERIMENT

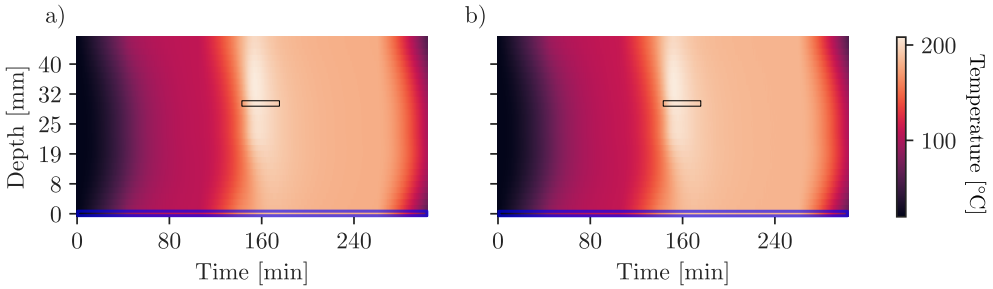

Figure 5: Illustration of a process simulation of composite materials. Each subfigure shows a temperature profile in degrees Celsius as a function of time along the x axis and depth along the y-axis. (a) shows the output of the Convergent Composite material simulator RAVEN [Convergent Manufacturing Technologies, 2019], simulating the curing process of a particular part. (b) shows the same process but originating from our *probabilistic surrogate network*. We perform inference in this process, where we infer the expected temperature in a specific time window (black box) conditioned on observed surface temperature measurements (blue boxes).

Fig. 5 compares output from our PSN and the reference simulator. As these outputs are indistinguishable, it provides further evidence that our PSN accurately models the reference simulator.

D.4 STOCHASTIC CONTROL FLOW ADDRESS TRANSITIONS

```
def control_flow_program(x):
    d_1 = Beta(50, 7)
    θ = sample(dist=d_1)
    μ = 0
    while True:
        d_2 = Categorical(prob=[1/5, 4/5])
        b = sample(dist=d_2)
        if b:
            d_3 = Normal(mean=0, std=1/2)
            z = sample(dist=d_3)
        else:
            d_3 = Normal(mean=2, std=1/2)
            z = sample(dist=d_3)
        μ += z
        d_4 = Categorical(prob=[1-θ,θ])
        c = sample(dist=d_4)
        if c:
            break
    d_5 = Normal(mean=μ, std=1)
    observe(x, likelihood=d_5)
    return θ
```

Figure 6: Program containing stochastic control flow in the form of a for-loop with a nested if-else statement. The task here would be to perform posterior inference of θ given the observed value of x.

For reference we re-illustrate the program Fig. 6 also shown in the main paper. The program contains two nested layers of stochastic control flow, allowing for an assessment of PSNs' capacity to learn the associated address transitions. Fig. 7(a) and (b) complements the results reported in the main paper by showing that the address transition paths and their associated estimated probabilities (using 50,000 traces each) of the program and the trained PSN are near indistinguishable. Only for long traces does small deviations begin to appear. It is reasonable to expect slight discrepancies between the address transition probabilities for increasingly long traces. The address occurrence probability decreases exponentially in the number of times n the original program stays in the for-loop – i.e. θ^n. Therefore we can expect (with reasonable probability) either the PSN or the program to produce addresses not produced by the other, when those addresses originate from executions with large `for loop` iterations. We conclude that these results show that the PSN indeed has learned accurate address transitions and support the claim made in the main paper.

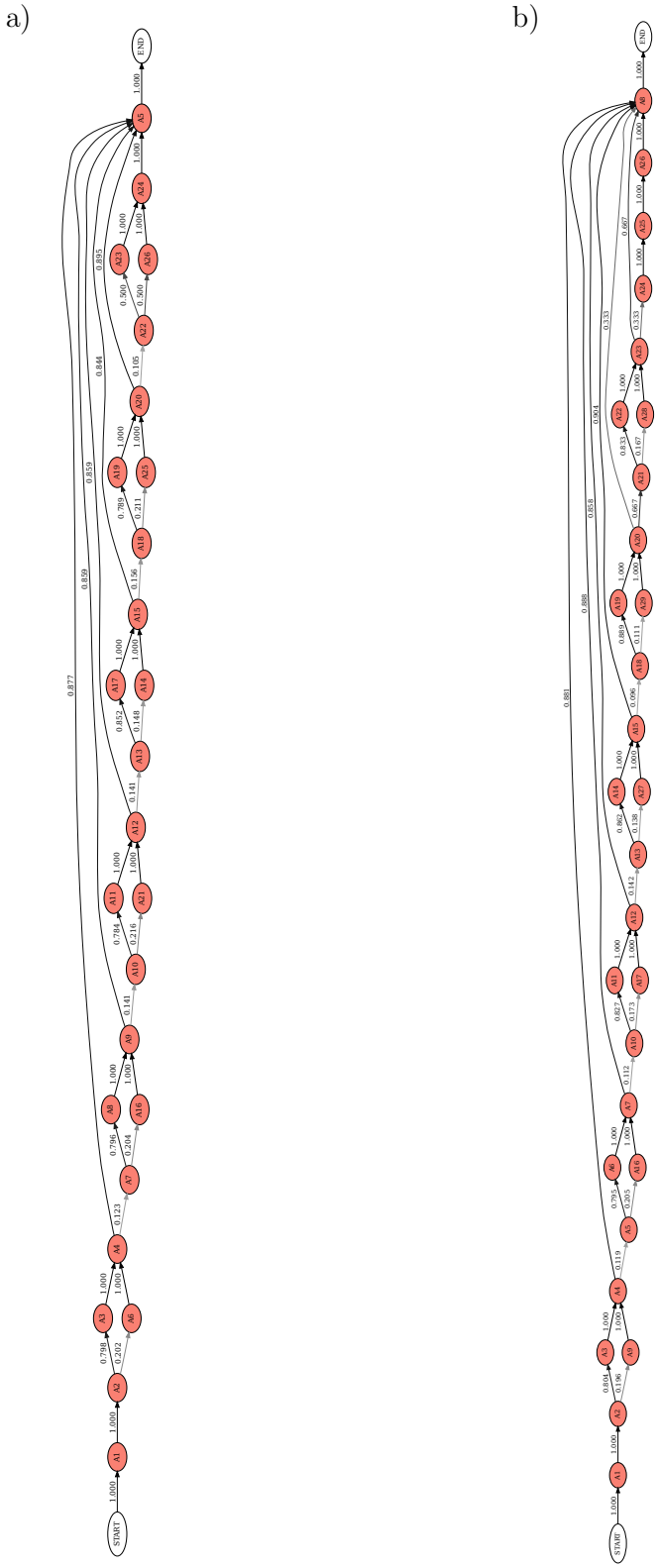

Figure 7: (a) Address transitions sampled from the original model shown in Fig. 6 with Table 4 mapping the address id A[i] to the actual address. (b) Address transitions sampled from the PSN, with Table 5 mapping the address id A[i] to the actual address allowing us to compare (a) and (b). For each plot the address transition probabilities are estimated across 50,000 traces.

Table 4: Address ID to address name for Fig. 7.

Address ID	Address
A1	30__forward__theta__Beta__1
A2	bern_0__Categorical(len_probs:2)__1
A3	z_1_0__Normal__1
A4	c_0__Categorical(len_probs:2)__1
A5	280__forward__?__Normal__1
A6	z_2_0__Normal__1
A7	bern_1__Categorical(len_probs:2)__1
A8	z_1_1__Normal__1
A9	c_1__Categorical(len_probs:2)__1
A10	bern_2__Categorical(len_probs:2)__1
A11	z_1_2__Normal__1
A12	c_2__Categorical(len_probs:2)__1
A13	bern_3__Categorical(len_probs:2)__1
A14	z_2_3__Normal__1
A15	c_3__Categorical(len_probs:2)__1
A16	z_2_1__Normal__1
A17	z_1_3__Normal__1
A18	bern_4__Categorical(len_probs:2)__1
A19	z_1_4__Normal__1
A20	c_4__Categorical(len_probs:2)__1
A21	z_2_2__Normal__1
A22	bern_5__Categorical(len_probs:2)__1
A23	z_1_5__Normal__1
A24	c_5__Categorical(len_probs:2)__1
A25	z_2_4__Normal__1
A26	z_2_5__Normal__1

Table 5: Address ID to address name for Fig. 7.

Address ID	Address
A1	30__forward__theta__Beta__1
A2	bern_0__Categorical(len_probs:2)__1
A3	z_1_0__Normal__1
A4	c_0__Categorical(len_probs:2)__1
A5	bern_1__Categorical(len_probs:2)__1
A6	z_1_1__Normal__1
A7	c_1__Categorical(len_probs:2)__1
A8	280__forward__?__Normal__1
A9	z_2_0__Normal__1
A10	bern_2__Categorical(len_probs:2)__1
A11	z_1_2__Normal__1
A12	c_2__Categorical(len_probs:2)__1
A13	bern_3__Categorical(len_probs:2)__1
A14	z_1_3__Normal__1
A15	c_3__Categorical(len_probs:2)__1
A16	z_2_1__Normal__1
A17	z_2_2__Normal__1
A18	bern_4__Categorical(len_probs:2)__1
A19	z_1_4__Normal__1
A20	c_4__Categorical(len_probs:2)__1
A21	bern_5__Categorical(len_probs:2)__1
A22	z_1_5__Normal__1
A23	c_5__Categorical(len_probs:2)__1
A24	bern_6__Categorical(len_probs:2)__1
A25	z_1_6__Normal__1
A26	c_6__Categorical(len_probs:2)__1
A27	z_2_3__Normal__1
A28	z_2_5__Normal__1
A29	z_2_4__Normal__1

D.5 PROGRAM SYNTHESIS DETAILS

The python code describing the generative model we approximate with a surrogate is given in Fig. 8. Note that the `depth_allow_else` data structure is in effect a stack that keeps track of the nesting of `if` and `else` statements. To generate valid programs, the surrogate has to learn that valid programs can only sample an `else` statement if an `if` statement has preceded it on the same nesting level. Furthermore, in our generative model, a valid program can only end at the lowest nesting level. Expanding on the results presented in the main text, additional example programs for both the original and the surrogate are displayed in Fig. 10. Address transitions for the synthetic programs can be found in Fig. 9. The structure of these transitions makes it clear that the program can only finish from specific addresses, corresponding to those sampled at the lowest nesting level. It is evident from the transitions presented for the surrogate that these dependencies are accurately captured.

```
def synthetic_programs():

    control_flow = {0: 'if',
                    1: 'else',
                    2: 'for',
                    3: 'body',
                    4: 'end'}

    nlines = 2
    depth = 1
    maxdepth = depth

    # set up probs
    probs_with_else = [1.0, 2.5, 1.0, 2.5, 1.0]
    probs_with_else = [p / sum(probs_with_else) \
      for p in probs_with_else]

    probs_no_else = [1.0, 0, 1.0, 2.5, 1.0]
    probs_no_else = [p / sum(probs_no_else) \
      for p in probs_no_else]

    depth_allow_else = {depth: False}

    while True:
       if depth_allow_else[depth]:
            probs = probs_with_else
       else:
            probs = probs_no_else

        # sample the statement type
        s = sample(Categorical(probs),
                   address=f"stat_{depth}_{nlines}")
        statement = control_flow[s]

        if statement == 'body':
            s = sample(Categorical([0.33, 0.33, 0.34]),
                       address=f"op_{depth}_{nlines}")
            num = sample(Normal(0.0, 1.0),
                         address=f"mod_{depth}_{nlines}")
            nlines += 1
            probs[0] *= 0.5
            probs[1] *= 0.5
            probs[2] *= 0.5
            probs[3] *= 0.5
            probs = [p / sum(probs) for p in probs]

        elif statement == 'if':
            s = sample(Categorical([0.5, 0.5]),
                       address=f"cond_{depth}_{nlines}")
            num = sample(Normal(0.0, 1.0),
                         address=f"comp_{depth}_{nlines}")
            depth += 1
            nlines += 1

            if depth > maxdepth:
                maxdepth = depth

            probs[0] *= 0.5
            probs[1] *= 0.5
            probs[2] *= 0.5
            probs = [p / sum(probs) for p in probs]

            depth_allow_else[depth] = True

        elif statement == 'else':
            if depth > maxdepth:
                maxdepth = depth

            probs[0] *= 0.5
            probs[1] *= 0.5
            probs[2] *= 0.5
            probs = [p / sum(probs) for p in probs]

            depth_allow_else[depth] = False

        elif statement == 'for':
            depth += 1
            nlines += 1
            range_val = sample(Categorical([1,5,2,8,4,2,5,7,98]),
                               address=f"range_{depth}_{nlines}")

            depth_allow_else[depth] = False

            if depth > maxdepth:
                maxdepth = depth

            probs[0] *= 0.5
            probs[1] *= 0.5
            probs[2] *= 0.5
            probs = [p / sum(probs) for p in probs]

        else: # is end
            probs[0] *= 0.5
            probs[1] *= 0.5
            probs = [p / sum(probs) for p in probs]
            del depth_allow_else[depth]
            depth -= 1

        if depth == 0:
            break

    return maxdepth
```

Figure 8: Model describing the program synthesis generative model.

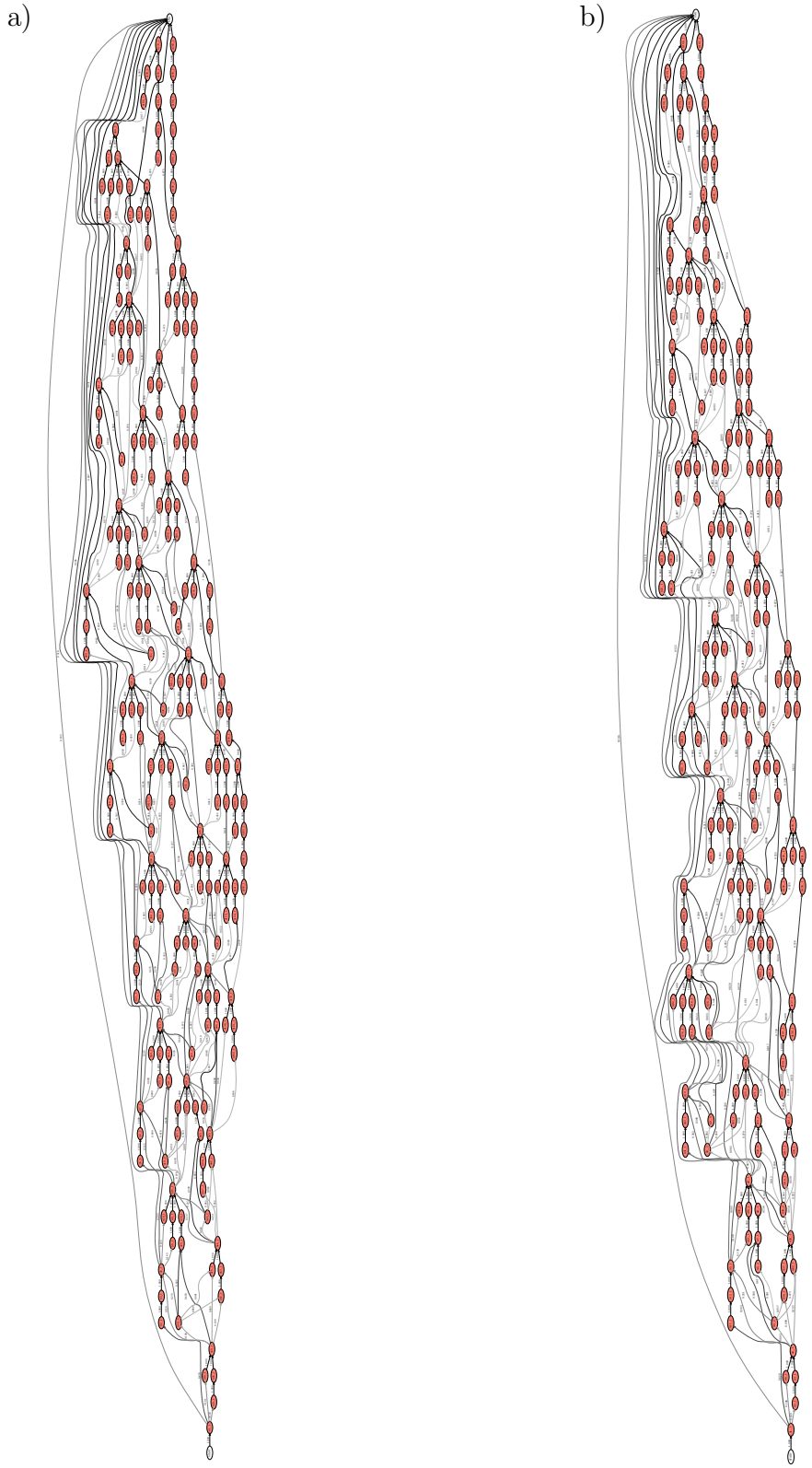

Figure 9: (a) Address transitions sampled from the original model shown in Fig. 8 (b) Address transitions sampled from the PSN. For each plot the address transition probabilities are estimated across 50,000 traces.

a)

```
def function(x):
    x = x - -0.4087
    for y in range(8):
        pass
        x = x - 0.4198
    x = x * 0.1882
    return x

def function(x):
    for y in range(8):
        pass
    x = x + -0.2114
    x = x * 0.3071
    return x

def function(x):
    for y in range(5):
        pass
        x = x * -2.2118
    return x

def function(x):
    for y in range(8):
        pass
    x = x * 0.9308
    return x

def function(x):
    for y in range(8):
        pass
        x = x + 0.7492
        x = x - -0.0710
    return x

def function(x):
    for y in range(8):
        pass
        x = x * -1.4855
        x = x + 0.2987
        if x < -0.9295:
            pass
    return x

def function(x):
    x = x - 0.5329
    x = x + -1.3560
    for y in range(8):
        pass
    x = x + 0.5513
    return x


def function(x):
    x = x + -0.4373
    for y in range(8):
        pass
        if x > -0.6187:
            pass
            for y in range(7):
                pass
            x = x * -0.6503
            x = x - 0.2484
    return x

def function(x):
    x = x - 0.7312
    if x > 0.0509:
        pass
        for y in range(4):
            pass
            x = x - -0.4496
    else:
        pass
    return x

def function(x):
    if x > -0.6631:
        pass
        for y in range(8):
            pass
    else:
        pass
    return x

def function(x):
    if x > -0.7258:
        pass
        x = x * 0.4430
        for y in range(8):
            pass
            x = x + 1.1464
    else:
        pass
    return x
```

b)

```
def function(x):
    for y in range(8):
        pass
        x = x + -0.2186
    return x

def function(x):
    for y in range(8):
        pass
    if x < 1.8181:
        pass
        x = x - -0.2057
    x = x + 0.0787
    x = x - -0.9475
    return x

def function(x):
    if x > 1.8695:
        pass
        x = x - 0.9109
        for y in range(3):
            pass
            x = x + 0.5025
    return x

def function(x):
    for y in range(8):
        pass
    return x

def function(x):
    for y in range(8):
        pass
    for y in range(8):
        pass
        x = x - 1.3633
    return x

def function(x):
    for y in range(8):
        pass
        if x < 0.6867:
            pass
            for y in range(8):
                pass
                x = x + 0.0118
                x = x - 1.1458
    return x

def function(x):
    for y in range(8):
        pass
    return x

def function(x):
    if x < -0.2176:
        pass
        x = x * -0.1251
        if x < 0.1617:
            pass
    else:
        pass
        for y in range(8):
            pass
        if x < -1.8860:
            pass
    if x < 0.0689:
        pass
    return x

def function(x):
    if x > -0.8971:
        pass
        for y in range(8):
            pass
            x = x + 0.9104
    else:
        pass
    x = x * -0.1137
    return x

def function(x):
    x = x * 1.7570
    if x < 1.2314:
        pass
        for y in range(8):
            pass
        x = x - 0.2301
        if x < -0.5254:
            pass
            x = x - -1.7400
    else:
        pass
    return x
```

Figure 10: (a) Example programs sampled from the original model shown in Fig. 8 (b) Example programs sampled from the learned surrogate.

References

Atilim Gunes Baydin and Tuan Anh Le. *pyprob*. https://github.com/probprog/pyprob, 2018.

Convergent Manufacturing Technologies. RAVEN Simulation Software. Technical report, Vancouver, 2019.