Proceedings of the 2021 Winter Simulation Conference
S. Kim, B. Feng, K. Smith, S. Masoud, Z. Zheng, C. Szabo, and M. Loper, eds.

TECHNICAL SUPPLEMENT FOR PAPER "DATA GENERATION WITH PROSPECT: A PROBABILITY SPECIFICATION TOOL"

Alan Ismaiel
Ivan Ruchkin
Oleg Sokolsky
Insup Lee

Computer and Information Science Department
University of Pennsylvania
3330 Walnut St.
Philadelphia, PA 19104, UNITED STATES

Jason Shu

Department of Mathematics
University of Pennsylvania
209 S 33rd St.
Philadelphia, PA 19104, UNITED STATES

SUMMARY

This is a technical supplement for paper "Data Generation with PROSPECT: a Probability Specification Tool":

- Appendix A provides formal definitions of the concepts used in the original paper.
- Appendix B proves all lemmas found in both the supplement and the original paper.
- Appendix C describes the algorithm behind the PROSPECT tool.
- Appendix D lists the source code of probabilistic programming baselines.

A DEFINITIONS

In this section, we formally notate many of the definitions underlying the paper.

Running example. A fair coin, represented with a discrete random variable x, is tossed and lands on heads ($x = T$) or tails ($x = F$). Then, independently, a fair dice is tossed, represented with a discrete random variable y and resulting in an integer value from 1 to 6.

Definition 1 (Random Variable) A *random variable* v is a measurable function from some set of inputs to a set of outcomes C.

Definition 2 (Event) An *event* e in sample space $\Omega(V)$ is a (possibly trivial) subset of $\Omega(V)$: $e \subseteq \Omega(V)$.

Non-intersecting events are called *mutually exclusive*. When an event contains a single outcome, $|e| = 1$, we call it an *elementary event*. By these definitions, all elementary events are mutually exclusive. $\{(T,4)\}$ is an elementary event in the running example, but event $\{(F,5),(F,6)\}$ is not; it means that the coin landed on tails and the dice rolled either 5 or 6.

Definition 3 (Probability Distribution) A *probability distribution* $\mathbb{P}_V$ over sample space $\Omega(V)$ is a function from any event in $\Omega(V)$ to the interval $[0,1]$ that obeys the *Kolmogorov Axioms*:

1. *Axiom 1:* The probability $\mathbb{P}_V(e)$ of an event e is a real number between 0 and 1, inclusive: $\forall e \subseteq \Omega(V) : 0 \leq \mathbb{P}_V(e) \leq 1$.
2. *Axiom 2:* The probability that at least one elementary event will occur is 1: $\mathbb{P}_V(\Omega(V)) = 1$.
3. *Axiom 3:* For set $\{e_1,...,e_k\}$ of mutually exclusive events, the probability of at least one event occurring is the sum of the event probabilities: $\mathbb{P}_V(\bigcup_{i=1}^{k} e_i) = \sum_{i=1}^{k} \mathbb{P}_V(e_i)$.

A *conditional probability distribution* on $\Omega(V)$ is derived from some distribution $\mathbb{P}_V$ on $\Omega(V)$ as follows. For any $e_1, e_2 \subseteq \Omega(V)$ s.t. $\mathbb{P}_V(e_2) > 0$, the probability of e_1 conditioned on e_2 is as follows:

$$\mathbb{P}_V(e_1 \mid e_2) = \frac{\mathbb{P}_V(e_1 \cap e_2)}{\mathbb{P}_V(e_2)}$$

A conditional probability distribution over all events $\bar{V}' = \bar{C}$ (assuming $V' \subseteq V, \bar{C} \in C(\bar{V}')$) given event $e \subseteq \Omega(V)$ is denoted as $\mathbb{P}_V(\bar{V}' \mid e)$. For two disjoint subsets $V_1, V_2 \subseteq V$, the set of conditional probability distributions over all $\bar{V}_1 = \bar{C}_1, \bar{C}_1 \in C(\bar{V}_1)$ when conditioned on each event $\bar{V}_2 = \bar{C}_2, \bar{C}_2 \in C(\bar{V}_2)$ is $\mathbb{P}_V(\bar{V}_1 \mid \bar{V}_2)$. When distributions are equal, all their conditionings are equal too.

Definition 4 (Chain Rule) The *chain rule* expresses a probability of $n > 1$ events $e_1, \ldots, e_n \subseteq \Omega(V)$ as a conditional "chain":

$$\mathbb{P}_V(\bigcap_{i=1}^{n} e_i) = \mathbb{P}_V(e_n \mid \bigcap_{i=1}^{n-1} e_i) \cdot \mathbb{P}_V(e_n \cap e_{n-1} \mid \bigcap_{i=1}^{n-2} e_i) \cdots \mathbb{P}_V(e_1)$$

Definition 5 (Law of Total Probability) The *law of total probability* relates an event's probability to those of its constituents. For sample spaces $\Omega(V)$ and $\Omega(V')$ s.t. $V' \subset V$, for any $\bar{C} \in C(\bar{V}')$:

$$\mathbb{P}_{V'}(\bar{V}' = \bar{C}) = \mathbb{P}_V(\bar{V}' = \bar{C}) = \sum_{\bar{C}' \in C(\overline{V \setminus V'})} \mathbb{P}_V(\bar{V} = \bar{C}, \overline{V \setminus V'} = \bar{C}')$$

A.1 Notions of Independence

We define independence and *conditional independence* for events, variable sets, and their pairs.

Definition 6 (Event Independence) Events $e_1, e_2 \subseteq \Omega(V)$ are *independent*, $e_1 \perp e_2$, if the occurrence of one does not affect the other. That is, $\mathbb{P}_V(e_1 \cap e_2) = \mathbb{P}_V(e_1)\mathbb{P}_V(e_2)$.

Definition 7 (Variable Set Independence) A set of variables $V' = \{v_1 \ldots v_k\}$, $V' \subseteq V$ is *independent*, denoted as $\perp V'$, if any subset of variables in V' take values independently from each other:

$$\forall j \in \{2, \ldots, k\} : \forall a_1 < \cdots < a_j \in \{1, \ldots, k\} : \forall (c_1, \ldots, c_j) \in C((v_{a_1}, \ldots, v_{a_j})) \ :$$
$$\mathbb{P}_V(v_{a_1} = c_1, \ldots, v_{a_j} = c_j) = \prod_{i \in \{1, \ldots, j\}} \mathbb{P}_V(v_{a_i} = c_i)$$

Our example tosses coins and rolls dice independently: $\perp\{x, y\}$

Definition 8 (Variable Set Pair Independence) Two sets of variables $V_1, V_2 \subseteq V$ are *independent*, denoted as $V_1 \perp V_2$, if all subsets of V_1 and V_2 take values independently from each other:

$$\forall V_1' \subseteq V_1 : \ \forall V_2' \subseteq V_2 \ : \forall \bar{C}_1 \in C(\bar{V}_1') : \forall \bar{C}_2 \in C(\bar{V}_2') : (\bar{V}_1' = \bar{C}_1) \perp (\bar{V}_2' = \bar{C}_2)$$

The above definitions extend naturally to conditional distributions.

Definition 9 (Conditional Event Independence) Given events $e_1, e_2, e_3 \subseteq \Omega(V)$, e_1 and e_2 are *conditionally independent* given e_3, denoted as $e_1 \perp e_2 \mid e_3$, if e_1 and e_2 do not affect each other after e_3. That is, $\mathbb{P}_V(e_1 \cap e_2 \mid e_3) = \mathbb{P}_V(e_1 \mid e_3) \cdot \mathbb{P}_V(e_2 \mid e_3)$.

Definition 10 (Conditional Variable Set Independence) Given sets of variables $V_1, V_2 \subseteq V$, $|V_1| = k$, V_1 is *conditionally independent* given V_2, denoted as $\perp V_1 \mid V_2$, if events for V_1 are independent given on every possible event $\bar{V}_2 = \bar{C}, \bar{C} \in C(V_2)$:

$$\forall j \in \{2, \ldots, k\} : \forall a_1 < \cdots < a_j \in \{1, \ldots, k\} : \forall (c_1, \ldots, c_j) \in C((v_{a_1}, \ldots, v_{a_j})) : \forall \bar{C} \in C(\bar{V}_2) :$$
$$\mathbb{P}_V(v_{a_1} = c_1, \ldots, v_{a_j} = c_j \mid \bar{V}_2 = \bar{C}) = \prod_{i=1}^{j} \mathbb{P}_V(v_{a_i} = c_i \mid \bar{V}_2 = \bar{C})$$

Definition 11 (Conditional Variable Set Pair Independence) Given sets of variables $V_1, V_2, V_3 \subseteq V$, V_1 and V_2 are *conditionally independent* on V_3, denoted as $V_1 \perp V_2 \mid V_3$, if all subsets of V_1 and V_2 are independent given every possible event $\bar{V}_3 = \bar{C}_3, \bar{C}_3 \in C(V_3)$:

$$\forall V_1' \subseteq V_1 :\ \forall V_2' \subseteq V_2\ : \forall \bar{C}_1 \in C(\bar{V}_1') : \forall \bar{C}_2 \in C(\bar{V}_2') : \forall \bar{C}_3 \in C(\bar{V}_3) : (\bar{V}_1' = \bar{C}_1) \perp (\bar{V}_2' = \bar{C}_2) \mid (\bar{V}_3 = \bar{C}_3)$$

A.2 Time and Assumptions

Definition 12 (Markov Property) The Markov Property asserts that the conditional probability distributions of future states of a stochastic process depend only on the present state: in other words, given the present, the future is independent of the past.

The Markov Property exists in our stochastic process as follows. Given a time index t, there exists a set B_t defined by the shape vector. For all such indices, the following holds:

$$\mathbb{P}_V(\bar{V}_{*,t} \mid \overline{(V_{*,1} \cup \cdots \cup V_{*,t-1})}) = \mathbb{P}_V(\bar{V}_{*,t} \mid \bar{B}_t)$$

We note further by the definition of B_t that $B_{t+1} \subseteq V_{*,t} \cup B_t$.

Definition 13 (Stationary Property) The Stationary Property asserts that the distribution does not change with time. Consider an arbitrary vector of $m < N \cdot K$ variables from V: $\bar{V}' = (v_{i_1,j_1}, v_{i_2,j_2}, \ldots,\ v_{i_m,j_m})$. Then any well-formed (i.e., those remaining in V) forward shifts in time do not change the distribution:

$$\begin{aligned}
&\forall i_1 < \cdots < i_m \in \{1 \ldots N \cdot K\} : \forall j_1 < \cdots < j_m \in \{1 \ldots N \cdot K\} : \\
&\forall l \in \{1 \ldots N - \max(j_1 \ldots j_m)\} : \forall \bar{C} \in C(\bar{V}') \\
&\mathbb{P}_V((v_{i_1,j_1}, \ldots, v_{i_m,j_m}) = \bar{C}) = \mathbb{P}_V((V_{i_1,j_{1+l}}, \ldots, v_{i_m,j_{m+l}}) = \bar{C})
\end{aligned}$$

B PROOF OF LEMMAS

This part contains the lemmas from the original paper, their proofs, and the supplementary lemmas. All references to lemma in proofs are local, and do not correspond with the numbers in the original paper-those numbers are indicated at the end of their definitions.

B.1 Lemmas for Technical Supplement

Lemma 1 For any event $e = \{\bar{C}_1, \ldots, \bar{C}_k\}$ s.t. $e \subseteq \Omega(V)$ and $k \in \{1, \ldots, |\Omega(V)|\}$, $\mathbb{P}_V(e) = \sum_{i=1}^{k} \mathbb{P}_V(\bar{V} = \bar{C}_i)$.

Proof. By definition, all elementary events are mutually exclusive from one another. Because an event is a subset of the sample space $\Omega(V)$, any given event e can be written equivalently as a union of the corresponding elementary events: in this case, they would be $\bar{V} = \bar{C}_1, \ldots, \bar{V} = \bar{C}_k$. Finally, by Axiom 3 of Definition 3, we conclude that the probability of a union of mutually exclusive elementary events is equal to the sum of the probabilities of each individual elementary event, or $\mathbb{P}_V(e) = \sum_{i=1}^{k} \mathbb{P}_V(\bar{V} = \bar{C}_i)$. □

B.2 Lemmas for Data Generation Workflow

Lemma 2 In the static case, given an arbitrary $\mathbb{P}_{V_{*,i}}$ distribution, then $\mathbb{P}_{D_t}$ for all $t \in \{1, \ldots, N\}$ is known. (Lemma 1 in paper)

Proof. The Stationary Property ensures that all D_t are equally distributed:

$$\forall i, j > i \in \{1, \ldots, N\} : \forall \bar{C} \in C(\bar{D}_i) : \mathbb{P}_V(\bar{D}_i = \bar{C}) = \mathbb{P}_V(\bar{D}_j = \bar{C})$$

In the static case, $\bar{S}=\vec{0}$, meaning that $B_t=\emptyset$ and $D_t=V_{*,t}$ for all $t\in\{1,\ldots,N\}$. Thus, for all $i,j\in\{1,\ldots,N\}$, it follows that:

$$\mathbb{P}_{D_i}(\bar{V}_{*,i}\mid\bar{B}_i)=\mathbb{P}_{V_{*,i}}:C(\bar{V}_{*,i})=C(\bar{V}_{*,j}):\forall\bar{C}\in C(\bar{V}_{*,i}):\mathbb{P}_V(\bar{V}_{*,i}=\bar{C})=\mathbb{P}_V(\bar{V}_{*,j}=\bar{C})$$

By the law of total probability, we conclude that $\mathbb{P}_{V_{*,i}}=\mathbb{P}_{V_{*,j}}$. Therefore, given a single arbitrary $\mathbb{P}_{V_{*,i}}$ distribution for the static case, we know all of $\mathbb{P}_{D_t}$ for all $t\in\{1,\ldots,N\}$. □

Lemma 3 In the time-invariant case, given a $\mathbb{P}_{D_{t_j}}$ distribution where $|D_{t_j}|=|D_N|$, then $\mathbb{P}_{D_t}$ for all $t\in\{1,\ldots,N\}$ is known. (Lemma 2 in paper)

Proof. The Stationary Property ensures that all D_t are equally distributed:

$$\forall i,j>i\in\{1,\ldots,N\}:\forall\bar{C}\in C(\bar{D}_i):\mathbb{P}_V(\bar{D}_i=\bar{C})=\mathbb{P}_V(\bar{D}_j=\bar{C})$$

In the time-invariant case, $\bar{S}\neq\vec{0}$, meaning that $B_t\neq\emptyset$ for all $t\in\{1,\ldots,N\}$. Let D_{t_j} be an arbitrary window set with the property $|D_{t_j}|=|D_N|$ By the Markov Property, for any window set $D_t=\{v_{i_1,j_1},v_{i_2,j_2},\ldots,v_{i_m,j_m}\}$, there exists a shifted set $D_{t_j,t}=\{v_{i_1,j_1+l},v_{i_2,j_2+l},\ldots,v_{i_m,j_m+l}\}\subseteq D_{t_j}$. Then:

$$C(\bar{D}_{t_j,t})=C(\bar{D}_t)\forall\bar{C}\in C(\bar{D}_t):\mathbb{P}_{D_t}(\bar{D}_t=\bar{C})=\mathbb{P}_{D_{t_j,t}}(\bar{D}_{t_j,t}=\bar{C})$$

By the law of total probability, the above equations lead to $\mathbb{P}_{D_{t_j,t}}=\mathbb{P}_{D_t}$. Then by the law of total probability, $\forall\bar{C}\in C(\bar{D}_{t_j,t}):\mathbb{P}_{D_{t_j,t}}(\bar{D}_{t_j,t}=\bar{C})=\mathbb{P}_{D_{t_j}}(\bar{D}_{t_j,t}=\bar{C})$. As a result, it is sufficient to know $\mathbb{P}_{D_{t_j}}$, because $\mathbb{P}_{D_{t_j,t}}$ and, hence, $\mathbb{P}_{D_t}$ are known then for all $t\in\{1,\ldots,N\}$. □

B.3 Lemmas for Inferring Distributions

Lemma 4 Given variables V and event $e\subseteq\Omega(V)$, $\mathbb{P}_V(e)$ can be expressed as a sum over $O(V)$. (Lemma 3 in paper)

Proof. An event e is defined such that $e=\{\bar{C}_1,\ldots,\bar{C}_k\}$, where $e\subseteq\Omega(V)$ and $k\in\{1,\ldots,|\Omega(V)|\}$. As shown in Lemma 1, $\mathbb{P}_V(e)=\sum_{i=1}^{k}\mathbb{P}_V(\bar{V}=\bar{C}_i)$, where $\bar{V}=\bar{C}_i$ represents one of the k elementary events included in e. This equality can then be rewritten in terms of O-parameters as follows:

$$\mathbb{P}_V(e)=\sum_{i=1}^{k}O_{\bar{C}_i}$$

□

Lemma 5 Given variables V and events $e_1,e_2\subseteq\Omega(V)$, $\mathbb{P}_V(e_1\mid e_2)$ can be expressed algebraically over $O(V)$. (Lemma 4 in paper)

Proof. For two events $e_1,e_2\subseteq\Omega(V)$,$\mathbb{P}_V(e_2)>0$, we can expand $\mathbb{P}_V(e_1\mid e_2)$ by definition:

$$\mathbb{P}_V(e_1\mid e_2)=\frac{\mathbb{P}_V(e_1\cap e_2)}{\mathbb{P}_V(e_2)}$$

By Lemma 4, any event in $\Omega(V)$ can have its probability written as a sum over O-parameters. Clearly $e_1\cap e_2,e_2\subseteq\Omega(V)$, so $\mathbb{P}_V(e_1\cap e_2)$ and $\mathbb{P}_V(e_2)$ can be rewritten in terms of these O-parameters, creating an algebraic representation of $\mathbb{P}_V(e_1\mid e_2)$ over $O(V)$. □

Lemma 6 Given variables V and its subset $V'\subseteq V$, an independence constraint $\perp V'$ can be equivalently translated into a finite set of algebraic constraints over parameters $O(V)$. (Lemma 5 in paper)

Proof. As observed in Definition 7, the definition of variable set independence, an independence constraint $\perp V'$ over $V' \subseteq V$ can be formally established through a finite number of probability lines, one for each combination of variable values in each subset of V'. These probability lines are composed of probability expressions $\mathbb{P}_V(e)$ where $e \subseteq \Omega(V)$. Lemma 4 established that any probability expression in $\mathbb{P}_V$ can have its probability written as a sum over O-parameters. Therefore, the independence constraint $\perp V'$ can be equivalently translated over parameters $O(V)$. □

Lemma 7 Given variables V and its subsets $V_1, V_2 \subseteq V$, an independence constraint $\perp V_1 \mid V_2$ can be equivalently translated into a finite set of algebraic constraints over parameters $O(V)$. (Lemma 6 in paper)

Proof. As observed in Definition 10, the definition of conditional variable set independence, a conditional independence constraint $\perp V_1 \mid V_2$ over $V_1, V_2 \subseteq V$ can be formally established through a finite amount of probability lines, one for each combination of variable values in each subset of V'. These probability lines are composed of conditional probability expressions $\mathbb{P}_V(e_1 \mid e_2)$ where $e_1, e_2 \subseteq \Omega(V)$. Lemma 5 established that any conditional probability expression in $\mathbb{P}_V$ can have its probability written as an algebraic expression over O-parameters. Therefore, the conditional independence constraint $\perp V_1 \mid V_2$ can be equivalently translated over parameters $O(V)$. □

Lemma 8 Given a set of variables V in the time-invariant case, a Stationary Property over V can be equivalently translated into a finite set of algebraic constraints over $O(V)$. (Lemma 7 in paper)

Proof. We choose two unique subsets $V_1', V_2' \subseteq V' \subseteq V$ as the largest subsets whose elementary events adhere to the Stationary Property in V':

$$\exists l \in \{1, \ldots, N\} : \forall v_{i,j} \in V_1' : \exists v_{i,j+l} \in V_2'$$

$$\forall \bar{C} \in C(\bar{V}_1') : \mathbb{P}_V(\bar{V}_1' = \bar{C}) = \mathbb{P}_V(\bar{V}_2' = \bar{C})$$

Then, by the law of total probability:

$$\forall \bar{C} \in C(\bar{V}_1') : \mathbb{P}_{V_1'}(\bar{V}_1' = \bar{C}) = \mathbb{P}_{V'}(\bar{V}_1' = \bar{C})$$

$$\forall \bar{C} \in C(\bar{V}_2') : \mathbb{P}_{V_2'}(\bar{V}_2' = \bar{C}) = \mathbb{P}_{V'}(\bar{V}_2' = \bar{C})$$

This establishes that $\forall \bar{C} \in C(\bar{V}_1')$, $\mathbb{P}_{V'}(\bar{V}_1' = \bar{C}) = \mathbb{P}_{V'}(\bar{V}_2' = \bar{C})$, which satisfies the Stationary Property in $\mathbb{P}_{V'}$. This set of probability equalities is bounded in size by $|\Omega(V')|$, and is therefore finite. Lemma 4 established that any probability expression can have its probability written as a sum over O-parameters. Therefore, the above set of probability equalities can be equivalently translated over parameters $O(V')$.

Suppose for contradiction that specifying $\forall \bar{C} \in C(\bar{V}_1')$, $\mathbb{P}_{V'}(\bar{V}_1' = \bar{C}) = \mathbb{P}_{V'}(\bar{V}_2' = \bar{C})$ was not sufficient to ensure the Stationary Property in V'. That is, there exists some $V_1'' \subseteq V'$ such that there was a corresponding $V_2'' \subseteq V'$ where $\forall \bar{C}' \in C(\bar{V}_1'')$, $\mathbb{P}_{V'}(\bar{V}_1'' = \bar{C}') = \mathbb{P}_{V'}(\bar{V}_2'' = \bar{C}')$ needed to be specified for the Stationary Property to hold — and it wasn't established by V_1' and V_2'. This implies that there exist variables in V_1'' and V_2'' not in V_1' and V_2' respectively: if $V_1'' \subseteq V_1'$ and $V_2'' \subseteq V_2'$, then any $\mathbb{P}_{V'}(\bar{V}_1'' = \bar{C}') = \mathbb{P}_{V'}(\bar{V}_2'' = \bar{C}')$ statement would have been been established by the original specifications. However, this is a contradiction, because V_1' and V_2' were defined to be the largest possible subset that could establish the Stationary Property, and the existence of V_1'' and V_2'' would mean that more variables could have been added to those sets. Thus, the specifications with V_1' and V_2' are enough to translate the Stationary Property into a finite set of constraints over parameters $O(V')$. □

Lemma 9 Given a window set D_t and its subset $B_t \neq \emptyset$ in the time-variant case, each $q \in Q(B_t)$ is equivalent to a unique polynomial over O-parameters in $O(D_t)$. (Lemma 8 in paper)

Proof. By the law of total probability, we know that the following relationship holds between B_t and D_t:

$$\forall \bar{C} \in C(B_t) : \mathbb{P}_{B_t}(\bar{B}_t = \bar{C}) = \mathbb{P}_{D_t}(\bar{D}_t = \bar{C})$$

Whereas $\bar{B}_t = \bar{C}$ is an elementary event in $\mathbb{P}_{B_t}$, as it maps to only one outcome, that is not necessarily the case in $\mathbb{P}_{\bar{D}_t}$. Therefore, let $\{\bar{C}_1, \ldots, \bar{C}_k\}$, such that $k \in \{1, \ldots, |\Omega(V)|\}$be the set of outcomes that correspond to $D_t = \bar{C}$ in $\mathbb{P}_V$. By Lemma 4, $\mathbb{P}_{D_t}(\bar{D}_t = \bar{C})$ can be represented as a sum of its corresponding elementary events: in this case, that would be $\bar{D}_t = \bar{C}_1, \ldots, \bar{D}_t = \bar{C}_k$. With this, $\mathbb{P}_{B_t}(\bar{B}_t = \bar{C}) = \mathbb{P}_{D_t}(\bar{D}_t = \bar{C})$ can be put into terms of $O(D_t)$ and $Q(B_t)$:

$$q_{\bar{C}} = \sum_{i=1}^{k} o_{\bar{C}_i}$$

This justification holds for all $|C(\bar{B}_t)|$ elementary events in $\mathbb{P}_{V'}$. □

C THE PROSPECT IMPLEMENTATION

Algorithm 1: The PROSPECT algorithm

Data: Text file with specification $\mathsf{Spec} = (\mathsf{Decl}, \mathsf{Indep}, \mathsf{Prob})$
Result: Value map for variables V in Spec
if Spec *is not a valid specification* **then return** $\emptyset$;
Parse Decl for $V, N, K, C, \bar{S}$, casetype ;
$F_o \leftarrow \emptyset$, $F_q \leftarrow \emptyset$, $A_o \leftarrow \emptyset$, $A_q \leftarrow \emptyset$;
if casetype = *'timevariant'* **then**
  $F_q \leftarrow$\{ 'basecase' in Prob translated to $Q(B_t)$\};
  $A_q \leftarrow$ all solutions $A(F_q) \subset [0,1]^{|Q(B_t)|}$;
  if $|A_q| \neq 1$ **then return** $\emptyset$;
  $F_o \leftarrow$ \{ $Q(B_t)$ translated to $Q(B_t)$ and $O(D_t)$\};
else if casetype = *'timeinvariant'* **then**
  $F_o \leftarrow$ \{ Stationary Assumption. translated to $O(D_t)$\};
$F_o \leftarrow F_o \cup$ \{ 'main' in Prob, Indep, translated to $O(D_t)$\};
$A_o \leftarrow$ all solutions $A(F_o) \subset [0,1]^{|O(D_t)|}$;
if $|A_o| \neq 1$ **then return** $\emptyset$;
$\mathbb{P}_{D_t} \leftarrow A_o$ over $\Omega(D_t)$ for $t \in \{\max(\bar{S}), \ldots, N\}$;
if casetype = *'timevariant'* **then** $\mathbb{P}_{B_t} \leftarrow A_q$ over $\Omega(B_t)$, for $t \in \{\max(\bar{S}), \ldots, N\}$;
CondProbs $\leftarrow$ Set of $\mathbb{P}_{D_i}(\bar{V}_{*,i} \mid \bar{B}_i) : i \in \{1, \ldots, N\}$ calculated with $\mathbb{P}_{D_t}$ and $\mathbb{P}_{B_t}$ in time-variant case, or $\mathbb{P}_{D_t}$ otherwise ;
if CondProbs *not well-defined* **then return** $\emptyset$;
return N samples from the DTMC built for V,N,K,AllProbs ;

Here we present the PROSPECT software tool (https://github.com/bisc/prospect), which converts user specifications into a system of equations and solves it. If the result is a unique distribution, then it sample all of it. PROSPECT is implemented in the Wolfram language, based on Mathematica 12.1. It reads a text file with the specification and prints and/or saves the generated data to a CSV file.

The control flow of PROSPECT is summarized in Algorithm 1. First, the declarations Decl are parsed. If the case is time-variant, then the base case is solved into A_q and added as a prior to define O-parameters through constraints F_o. In all cases, F_o gets the constraints from the main probability and independence specifications; the time-invariant case also gets constraints from the Stationary Assumption. This system is solved, providing a set of probabilities $\mathbb{P}_{D_t}(\bar{V}_{*,t} \mid \bar{B}_t)$ necessary for sampling. Finally, an appropriate DTMC is sampled, providing the generated data.

D PROBABILISTIC PROGRAMS FOR EVALUATION SCENARIOS

In this section, we present author implementations of the three evaluation scenarios from the original paper. The code is written in the probabilistic programming language Pyro v1.5.1 (based on Python v3.8.5). All of this code can be found and run in the PROSPECT repository at http://github.com/bisc/prospect.

D.1 Scenario 1: Lane Keeping, Static Case

The probabilistic programs below implement the Scenario 1: Lane Keeping case.

The accurate baseline:

```python
import torch
import pyro
import sys
import csv

# command line takes one argument for number of samples
if len(sys.argv) >= 2:
    num_samples = int(sys.argv[1])
else:
    print('In the command line, please specify number of samples\n')
    sys.exit()

def static_ex():
    # encode specifications
    # time and lane are independent given detection
    prob_detected_given_day = .75
    prob_detected_given_twilight = .4
    prob_detected_given_night = .2
    prob_day = .6
    prob_twilight = (1 - prob_day) / 2
    prob_night = prob_twilight
    prob_out = .2
    prob_detected_given_in = .6

    # create cat dist for time var, sample
    time = pyro.sample('time', pyro.distributions.Categorical(torch.tensor([prob_day, prob_twilight, prob_night])))

    # update probability of detected given observed time
    if(time.item() == 0.0):
        time = 'day'
        detection = pyro.sample('detection', pyro.distributions.Bernoulli(prob_detected_given_day))
    elif(time.item() == 1.0):
        time = 'twilight'
        detection = pyro.sample('detection', pyro.distributions.Bernoulli(prob_detected_given_twilight))
    else:
        time = 'night'
        detection = pyro.sample('detection', pyro.distributions.Bernoulli(prob_detected_given_night))

    # the time and lane variables are conditionally independent given any value of the detection variable; suffices to compute P[lane = 'in' | detection = 'detected'] and P[lane = 'in' | detection = 'not detected'] to generate data

    # compute P[detected] before invoking Bayes' Theorem
    prob_detected = prob_detected_given_day * prob_day + prob_detected_given_twilight * prob_twilight + prob_detected_given_night * prob_night

    if(detection.item() == 1.0):
        detection = 'detected'
        prob_in_given_detected = prob_detected_given_in * (1 - prob_out) / prob_detected
        lane = pyro.sample('lane', pyro.distributions.Bernoulli(prob_in_given_detected))
    else:
        detection = 'not detected'
        prob_in_given_notdetected = ((1 - prob_out) - (prob_detected_given_in * (1 - prob_out))) / (1 - prob_detected)
        lane = pyro.sample('lane', pyro.distributions.Bernoulli(prob_in_given_notdetected))

    if(lane.item() == 1.0):
        lane = 'in'
    else:
        lane = 'out'

    return [time, detection, lane]

with open('static_baseline_accurate.csv', 'w') as file:
    writer = csv.writer(file)
    # writer.writerow(['time', 'detection', 'lane'])
    for i in range(num_samples):
        writer.writerow(static_ex())
```

The naive baseline:

```python
import torch
import pyro
import sys
import csv

# command line takes one argument for number of samples
if len(sys.argv) >= 2:
    num_samples = int(sys.argv[1])
else:
    print('In the command line, please specify number of samples\n')
    sys.exit()

def static_ex():
    # encode specifications
    # time and lane are independent given detection
    prob_detected_given_day = .75
    prob_detected_given_twilight = .4
    prob_detected_given_night = .2
    prob_day = .6
    prob_twilight = (1 - prob_day) / 2
    prob_night = prob_twilight
    prob_out = .2
    prob_detected_given_in = .6

    # create cat dist for time var, sample
    time = pyro.sample('time', pyro.distributions.Categorical(torch.tensor([prob_day, prob_twilight, prob_night])))

    # update probability of detected given observed time
    if(time.item() == 0.0):
        time = 'day'
        detection = pyro.sample('detection', pyro.distributions.Bernoulli(prob_detected_given_day))
    elif(time.item() == 1.0):
        time = 'twilight'
        detection = pyro.sample('detection', pyro.distributions.Bernoulli(prob_detected_given_twilight))
    else:
        time = 'night'
        detection = pyro.sample('detection', pyro.distributions.Bernoulli(prob_detected_given_night))

    if(detection.item() == 1.0):
        detection = 'detected'
    else:
        detection = 'not detected'

    # if one isn't careful, it may be tempting to generate data for "lane" using just the marginal probability P[lane = "out"] =
      .2, as shown below
    lane = pyro.sample('lane', pyro.distributions.Bernoulli(1 - prob_out))
    if(lane.item() == 1.0):
        lane = 'in'
    else:
        lane = 'out'

    # This method uses the assumption that lane and detection are independent, but this is not necessarily true based on the
      specifications.
    # More calculation is needed to compute conditional probabilities as detailed in the static.py.

    return [time, detection, lane]

with open('static_baseline_naive.csv', 'w') as file:
    writer = csv.writer(file)
    # writer.writerow(['time', 'detection', 'lane'])
    for i in range(1, num_samples + 1):
        writer.writerow(static_ex())
```

D.2 Scenario 2: Network Latency, Time-Invariant Case

The probabilistic programs below implement the Scenario 2: Network Latency case.

The accurate baseline:

```python
import pyro
import sys
import csv

# command line takes one argument for number of time steps in the time series
if len(sys.argv) >= 2:
    num_steps_arg = int(sys.argv[1])
else:
    print('In the command line, please specify number of time steps in the time series\n')
    sys.exit()

def invariant_ex(num_steps):
```

```
    if (num_steps < 1):
       return []
    # encode specifications
    # latency and ping_tmin1 are independent given ping_t
    # refer to ping as t TODO rename
    prob_t_hi_given_tmin1_hi = .7
    prob_t_lo_given_tmin1_lo = .65
    prob_lat_lo = .8
    prob_t_hi_given_lat_hi = .6

    # keep invariance assumption in mind: p(ping_t = hi) = p(ping_t-1 = hi)

    # derive by hand using invariance assumption
    prob_t_hi = (1 - prob_t_lo_given_tmin1_lo) / (1 - prob_t_hi_given_tmin1_hi + (1 - prob_t_lo_given_tmin1_lo))

    # solve for p(latency_t = hi | ping_t = hi) and p(latency_t = hi | ping_t = lo)
    prob_lat_hi_given_t_hi = prob_t_hi_given_lat_hi * (1 - prob_lat_lo) / prob_t_hi
    prob_lat_hi_given_t_lo = (1 - prob_t_hi_given_lat_hi) * (1 - prob_lat_lo) / (1 - prob_t_hi)

    # add results from each iteration to this list
    return_list = []

    # sample initial ping value from marginal prob
    ping_prev = pyro.sample('ping_prev', pyro.distributions.Bernoulli(prob_t_hi))

    # assign labels to ping_prev
    if (ping_prev.item() == 1.0):
        ping_prev = 'high'
    else:
        ping_prev = 'low'

    for x in range(num_steps):
        # sample ping_curr (current time step) given ping_t-1
        if (ping_prev == 'high'):
            ping_curr = pyro.sample('ping_curr', pyro.distributions.Bernoulli(prob_t_hi_given_tmin1_hi))
        else:
            ping_curr = pyro.sample('ping_curr', pyro.distributions.Bernoulli(1 - prob_t_lo_given_tmin1_lo))

        # assign label to ping_curr and sample lat_curr given ping_curr
        if (ping_curr.item() == 1.0):
            ping_curr = 'high'
            lat_curr = pyro.sample('lat_curr', pyro.distributions.Bernoulli(prob_lat_hi_given_t_hi))
        else:
            ping_curr = 'low'
            lat_curr = pyro.sample('lat_curr', pyro.distributions.Bernoulli(prob_lat_hi_given_t_lo))

        # assign label to lat_curr
        if (lat_curr.item() == 1.0):
            lat_curr = 'high'
        else:
            lat_curr = 'low'

        # append current var values to list of results
        return_list.append([lat_curr, ping_curr])

        # update ping_t-1 with ping_t
        ping_prev = ping_curr
    return return_list

with open('time_invariant_baseline_accurate.csv', 'w') as file:
    writer = csv.writer(file)
    writer.writerows(invariant_ex(num_steps_arg))
```

The naive baseline:

```
import pyro
import sys
import csv

# command line takes one argument for number of time steps in the time series
if len(sys.argv) >= 2:
    num_steps_arg = int(sys.argv[1])
else:
    print('In the command line, please specify number of time steps in the time series\n')
    sys.exit()

def invariant_ex(num_steps):
    if (num_steps < 1):
       return []
    # encode specifications
    # latency and ping_tmin1 are independent given ping_t
    # refer to ping as t TODO rename
    prob_t_hi_given_tmin1_hi = .7
    prob_t_lo_given_tmin1_lo = .65
    prob_lat_lo = .8
```

```
    prob_t_hi_given_lat_hi = .6

    # keep invariance assumption in mind: p(ping_t = hi) = p(ping_t-1 = hi)

    # derive by hand using invariance assumption
    prob_t_hi = (1 - prob_t_lo_given_tmin1_lo) / (1 - prob_t_hi_given_tmin1_hi + (1 - prob_t_lo_given_tmin1_lo))

    # solve for p(ping_t = hi | latency_t = lo)
    prob_t_hi_given_lat_lo = (prob_t_hi - (prob_t_hi_given_lat_hi * (1 - prob_lat_lo))) / prob_lat_lo

    # solve for p(latency_t = hi | ping_t = hi) and p(latency_t = hi | ping_t = lo)
    # prob_lat_hi_given_t_hi = prob_t_hi_given_lat_hi * (1 - prob_lat_lo) / prob_t_hi
    # prob_lat_hi_given_t_lo = (1 - prob_t_hi_given_lat_hi) * (1 - prob_lat_lo) / (1 - prob_t_hi)

    # add results from each iteration to this list
    return_list = []

    # sample initial ping value from marginal prob
    # ping_prev = pyro.sample('ping_prev', pyro.distributions.Bernoulli(prob_t_hi))

    # assign labels to ping_prev
    # if (ping_prev.item() == 1.0):
    #     ping_prev = 'high'
    # else:
    #     ping_prev = 'low'

    for x in range(num_steps):
        # sample ping_curr (current time step) given ping_t-1
        # if (ping_prev == 'high'):
        #     ping_curr = pyro.sample('ping_curr', pyro.distributions.Bernoulli(prob_t_hi_given_tmin1_hi))
        # else:
        #     ping_curr = pyro.sample('ping_curr', pyro.distributions.Bernoulli(1 - prob_t_lo_given_tmin1_lo))

        lat_curr = pyro.sample('lat_curr', pyro.distributions.Bernoulli(1 - prob_lat_lo))

        # assign label to lat_curr and sample ping_curr given lat_curr
        if (lat_curr.item() == 1.0):
            lat_curr = 'high'
            ping_curr = pyro.sample('ping_curr', pyro.distributions.Bernoulli(prob_t_hi_given_lat_hi))
        else:
            lat_curr = 'low'
            ping_curr = pyro.sample('ping_curr', pyro.distributions.Bernoulli(prob_t_hi_given_lat_lo))

        # assign label to ping_curr
        if (ping_curr.item() == 1.0):
            ping_curr = 'high'
        else:
            ping_curr = 'low'

        # append current var values to list of results
        return_list.append([lat_curr, ping_curr])

        # update ping_t-1 with ping_t
        ping_prev = ping_curr
    return return_list

with open('time_invariant_baseline_naive.csv', 'w') as file:
    writer = csv.writer(file)
    writer.writerows(invariant_ex(num_steps_arg))
```

D.3 Scenario 3: Tool Wearing, Time-Variant Case

The probabilistic programs below implement the Scenario 3: Tool Wearing case.

The accurate baseline:

```
import pyro
import sys
import csv

# command line takes two arguments for number of time steps in the time series and instances of time series
if len(sys.argv) >= 3:
    num_steps_arg = int(sys.argv[1])
    num_instances_arg = int(sys.argv[2])
else:
    print('In the command line, please specify number of time steps and number of time series (in that order) in the time series\n
    ')
    sys.exit()

def variant_ex(num_steps):
    if (num_steps < 1):
       return []
```

```
    # encode specifications

    prob_tool_broken_given_toolmin1_broken = 1
    prob_op_ok = .8

    # Keep below specifications in mind:
    # other specs: operation and tool_tmin1 are independent
    # P[tool_t = "func"] = P[tool_t-1 = "func"]-.1*P[tool_t-1 = "func"]
    # P[operation_t = "ok" && tool_t = "func"] = .95*P[operation_t = "ok" && tool_t-1 = "func"]

    prob_toolmin1_func = .9 # base case, step 0

    return_tool_list = [] # values of tool at each time step of a single time series instance
    return_op_list = [] # values of operation at each time step of an instance

    # sample initial tool value from marginal prob
    tool_prev = pyro.sample('tool_prev', pyro.distributions.Bernoulli(prob_toolmin1_func))

    # label initial tool value
    if (tool_prev.item() == 1.0):
        tool_prev = 'func'
    else:
        tool_prev = 'broken'

    for x in range(num_steps):
        # intermediate computations done each iteration
        prob_tool_func = prob_toolmin1_func - .1 * prob_toolmin1_func
        prob_tool_broken = 1 - prob_tool_func
        prob_toolmin1_broken = 1 - prob_toolmin1_func

        # compute p(tool_t = func | tool_t-1 = broken)
        prob_tool_func_given_toolmin1_broken = (prob_toolmin1_broken - (prob_tool_broken_given_toolmin1_broken *
      prob_toolmin1_broken)) / prob_toolmin1_broken

        # compute p(tool_t = func | tool_t-1 = func)
        prob_tool_func_given_toolmin1_func = (prob_tool_func - (prob_tool_func_given_toolmin1_broken * prob_toolmin1_broken)) /
      prob_toolmin1_func

        # compute current step probabilities
        prob_op_ok_and_tool_func = .95 * prob_op_ok * prob_toolmin1_func # use independence fact to split RHS
        prob_op_ok_given_tool_func = prob_op_ok_and_tool_func / prob_tool_func
        prob_op_ok_given_tool_broken = (prob_op_ok - prob_op_ok_and_tool_func) / prob_tool_broken

        # sample tool_curr given tool_prev
        if (tool_prev == 'func'):
            tool_curr = pyro.sample('tool_curr', pyro.distributions.Bernoulli(prob_tool_func_given_toolmin1_func))
        else:
            tool_curr = pyro.sample('tool_curr', pyro.distributions.Bernoulli(prob_tool_func_given_toolmin1_broken))

        # label tool_curr and sample op_curr given tool_curr
        if (tool_curr.item() == 1.0):
            tool_curr = 'func'
            op_curr = pyro.sample('op_curr', pyro.distributions.Bernoulli(prob_op_ok_given_tool_func))
        else:
            tool_curr = 'broken'
            op_curr = pyro.sample('op_curr', pyro.distributions.Bernoulli(prob_op_ok_given_tool_broken))

        # label op_curr
        if (op_curr.item() == 1.0):
            op_curr = 'ok'
        else:
            op_curr = 'fail'

        return_tool_list.append(tool_curr)
        return_op_list.append(op_curr)

        # update tool_t-1 with tool_t
        tool_prev = tool_curr
        prob_toolmin1_func = prob_tool_func

        return_lists = [return_tool_list, return_op_list]
    return return_lists

with open('time_variant_baseline_var_tool_accurate.csv', 'w') as tool_file, open('time_variant_baseline_var_op_accurate.csv', 'w')
      as op_file:
    tool_writer = csv.writer(tool_file)
    op_writer = csv.writer(op_file)
    if (num_instances_arg >= 1):
        for y in range(num_instances_arg):
            result = variant_ex(num_steps_arg)
            tool_writer.writerow(result[0])
            op_writer.writerow(result[1])
```

The naive baseline:

```python
import pyro
import sys
import csv

# command line takes two arguments for number of time steps in the time series and instances of time series
if len(sys.argv) >= 3:
    num_steps_arg = int(sys.argv[1])
    num_instances_arg = int(sys.argv[2])
else:
    print('In the command line, please specify number of time steps and number of time series (in that order) in the time series\n')
    sys.exit()

def variant_ex(num_steps):
    if (num_steps < 1):
       return []

    # encode specifications

    prob_tool_broken_given_toolmin1_broken = 1
    prob_op_ok = .8

    # Keep below specifications in mind:
    # other specs: operation and tool_tmin1 are independent
    # P[tool_t = "func"] = P[tool_t-1 = "func"]-.1*P[tool_t-1 = "func"]
    # P[operation_t = "ok" && tool_t = "func"] = .95*P[operation_t = "ok" && tool_t-1 = "func"]

    prob_toolmin1_func = .9 # base case, step 0

    return_tool_list = [] # values of tool at each time step of a single time series instance
    return_op_list = [] # values of operation at each time step of an instance

    # sample initial tool value from marginal prob
    tool_prev = pyro.sample('tool_prev', pyro.distributions.Bernoulli(prob_toolmin1_func))

    # label initial tool value
    if (tool_prev.item() == 1.0):
        tool_prev = 'func'
    else:
        tool_prev = 'broken'

    for x in range(num_steps):
        # intermediate computations done each iteration
        prob_tool_func = prob_toolmin1_func - .1 * prob_toolmin1_func
        prob_tool_broken = 1 - prob_tool_func
        prob_toolmin1_broken = 1 - prob_toolmin1_func

        # compute p(tool_t = func | tool_t-1 = broken)
        prob_tool_func_given_toolmin1_broken = (prob_toolmin1_broken - (prob_tool_broken_given_toolmin1_broken * prob_toolmin1_broken)) / prob_toolmin1_broken

        # compute p(tool_t = func | tool_t-1 = func)
        prob_tool_func_given_toolmin1_func = (prob_tool_func - (prob_tool_func_given_toolmin1_broken * prob_toolmin1_broken)) / prob_toolmin1_func

        # compute current step probabilities
        # prob_op_ok_and_tool_func = .95 * prob_op_ok * prob_toolmin1_func # use independence fact to split RHS
        # prob_op_ok_given_tool_func = prob_op_ok_and_tool_func / prob_tool_func
        # prob_op_ok_given_tool_broken = (prob_op_ok - prob_op_ok_and_tool_func) / prob_tool_broken

        # sample tool_curr given tool_prev
        if (tool_prev == 'func'):
            tool_curr = pyro.sample('tool_curr', pyro.distributions.Bernoulli(prob_tool_func_given_toolmin1_func))
        else:
            tool_curr = pyro.sample('tool_curr', pyro.distributions.Bernoulli(prob_tool_func_given_toolmin1_broken))

        # label tool_curr
        if (tool_curr.item() == 1.0):
            tool_curr = 'func'
            # op_curr = pyro.sample('op_curr', pyro.distributions.Bernoulli(prob_op_ok_given_tool_func))
        else:
            tool_curr = 'broken'
            # op_curr = pyro.sample('op_curr', pyro.distributions.Bernoulli(prob_op_ok_given_tool_broken))

        # sample op_curr from marginal probability
        op_curr = pyro.sample('op_curr', pyro.distributions.Bernoulli(prob_op_ok))

        # label op_curr
        if (op_curr.item() == 1.0):
            op_curr = 'ok'
        else:
            op_curr = 'fail'

        return_tool_list.append(tool_curr)
        return_op_list.append(op_curr)

        # update tool_t-1 with tool_t
```

```
        tool_prev = tool_curr
        prob_toolmin1_func = prob_tool_func

        return_lists = [return_tool_list, return_op_list]
    return return_lists

with open('time_variant_baseline_var_tool_naive.csv', 'w') as tool_file, open('time_variant_baseline_var_op_naive.csv', 'w') as
     op_file:
    tool_writer = csv.writer(tool_file)
    op_writer = csv.writer(op_file)
    if (num_instances_arg >= 1):
        for y in range(num_instances_arg):
            result = variant_ex(num_steps_arg)
            tool_writer.writerow(result[0])
            op_writer.writerow(result[1])
```