Programmation d'Applications Réactives Probabilistes

Guillaume Baudart[1], Louis Mandel[1], Marc Pouzet[2],
Eric Atkinson[3], Benjamin Sherman[3], Michael Carbin[3]

[1] MIT-IBM Watson AI Lab, IBM Research
[2] École normale supérieure, PSL University
[3] Massachusetts Institute of Technology

Abstract

Les langages synchrones ont été introduits pour concevoir des systèmes embarqués temps-réel. Ces langages dédiés permettent d'écrire une spécification précise du système, de la simuler, la valider par du test ou de la vérification formelle puis de la compiler vers du code exécutable. Cependant, ils offrent un support limité pour modéliser les comportements non-déterministes qui sont omniprésents dans les systèmes embarqués.

Dans cet article, nous présentons ProbZélus, une extension probabiliste d'un langage synchrone descendant de Lustre. ProbZélus permet de décrire des modèles probabilistes réactifs en interaction avec un environnement observable. Lors de l'exécution, un ensemble de techniques d'inférence peut être utilisé pour apprendre les distributions de paramètres du modèle à partir de données observées. Nous illustrons l'expressivité de ProbZélus avec des exemples comme un détecteur de trajectoire à partir d'observations bruitées, ou un contrôleur de robot capable d'inférer à la fois sa position et une carte de son environnement.

1 Introduction

Les langages synchrones ont été introduits il y a 30 ans pour la conception de systèmes embarqués temps-réel. Ils sont fondés sur le modèle de *parallélisme synchrone* [3]. De nombreux langages ont été proposés, dont Scade [8] utilisé pour concevoir et implémenter les logiciels temps-réel critiques (commandes de vol, freinage, contrôle moteur, par exemple).

Scade est un langage *flot de données* descendant de Lustre : les signaux d'entrées/sorties sont des suites infinies (ou *flots*), un système (ou *nœud*) est une fonction de suites, et toutes les suites progressent ensemble, instant après instant, de manière *synchrone*. Ce style de programmation est bien adapté pour exprimer les blocs de contrôle classiques (relais, filtres, contrôleurs PID, etc.), un modèle discret de l'environnement et une boucle d'interaction entre ces deux composants. Le code suivant implémente un contrôleur PID (proportionnel, intégral, dérivé) en Zélus, un langage académique proche de Scade [5].[1]

```
let node pid (r, y) = u where
  rec e = r -. y
  and u = p *. e +. i *. integr(0., e) +. d *. deriv(e)
```

Le nœud `pid` définit un flot de commandes `u` à partir d'un flot de consignes `r` et d'un flot de mesures `y`. La commande est la somme pondérée de trois actions (proportionnelle, intégrale, et dérivée) appliquée à l'erreur entre la consigne et la mesure. Les poids `p`, `i`, et `d` sont des constantes et les appels de nœuds `integr(0., e)` et `deriv(e)` calculent respectivement l'intégrale (initialisée à `0.`) et la dérivée de `e`.

Les langages synchrones offrent un support limité pour modéliser le non-déterminisme et les incertitudes de l'environment, pourtant omniprésents dans un système réel. Un contrôleur n'a souvent qu'une vision partielle et bruitée de son environnement, et le comportement du

[1] www.zelus.di.ens.fr

système lui-même est souvent sujet à des perturbations. Il est bien sûr possible de programmer des contrôleurs arbitrairement compliqués, capable d'anticiper et de compenser ces incertitudes, mais leur programmation est notoirement difficile et source d'erreurs [1].

Les langages de programmation probabilistes permettent de décrire des modèles probabilistes et d'*inférer* automatiquement les distributions de paramètres *latents* (i.e., non-observés) à partir d'*observations* (i.e., des entrées). Une approche populaire [4, 10, 16, 20–22] consiste à étendre un langage de programmation généraliste avec trois nouvelles constructions: (1) `x = sample(d)` introduit une variable aléatoire *latente* `x` de distribution `d` ; (2) `observe(d, y)` mesure la *vraisemblance* d'une *observation* `y` par rapport à une distribution `d` (i.e., la densité de `y` par rapport à `d`) ; (3) `infer m obs` calcule la distribution des valeurs de sortie d'un programme ou *modèle* `m` *sachant* les observations données en entrée `obs`. Les langages de programmation probabilistes offrent une variété de techniques d'inférence automatique qui vont du calcul symbolique exact, aux approximations par échantillonnage (méthodes de Monte-Carlo). Mais aucun de ces langages n'offre le support et les garanties associées données par les langages synchrones pour concevoir des systèmes réactifs embarqués (programmation flot-de-données, exécution avec des ressources bornées, absence d'interbloquage).

Dans cet article, nous présentons ProbZélus [2], une extension probabiliste de Zélus. ProbZélus permet de combiner les constructions d'un langage réactif synchrone et les constructions d'un langage probabiliste — `sample`, `observe` et `infer` — pour développer des *applications réactives probabilistes*. Nous illustrons par des exemples les avantages offerts par ProbZélus :

1. Programmation de modèles probabilistes réactifs : un détecteur de trajectoire à partir d'observations bruitées (section 2).
2. Inférence dans la boucle : un contrôleur de robot guidé par le résultat d'un détecteur de trajectoire probabiliste (section 3).
3. Inférence semi-symbolique : un modèle de robot plus complexe, capable d'inférer à la fois sa position et une carte de son environnement (section 4).

2 Programmation Probabiliste Réactive

Nous rappelons ici le principe de l'inférence bayésienne sur lequel se fonde les langages de programmation probabilistes. Nous présentons ensuite un premier modèle probabiliste réactif : un détecteur de trajectoire à partir d'observations bruitées.

L'inférence bayésienne permet de calculer la probabilité d'une hypothèse (distribution *a posteriori*) à partir de croyances antérieures (distribution *a priori*), et d'observations. L'inférence s'appuie sur le théorème de Bayes :

$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{x})p(\mathbf{y}|\mathbf{x})}{p(\mathbf{y})} \propto p(\mathbf{x})p(\mathbf{y}|\mathbf{x})$$

La distribution *a posteriori* $p(\mathbf{x}|\mathbf{y})$ des variables *latentes* $\mathbf{x}$ après avoir observé les données $\mathbf{y}$ dépend essentiellement de deux termes : $p(\mathbf{x})$ la distribution supposée de $\mathbf{x}$ *a priori* (construction `sample`), et $p(\mathbf{y}|\mathbf{x})$ la *vraisemblance* des données $\mathbf{y}$ sachant $\mathbf{x}$ (construction `observe`).

En ProbZélus, les modèles probabilistes sont des nœuds particuliers introduits par le mot-clef `proba`. Les constructions `sample` et `observe` ne peuvent être invoquées qu'à l'intérieur d'un nœud probabiliste. L'opérateur `infer` prend en argument un nœud probabiliste et produit un resultat déterministe : la distribution *a posteriori* définie par le modèle. Ces contraintes sont vérifiées par typage lors de la compilation. Les signatures associées aux constructions probabilistes sont les suivantes :

```
val sample  : 'a Distribution.t ~D~> 'a
val observe : 'a Distribution.t * 'a ~D~> unit
val infer   : ('a ~D~> 'b) -S-> 'a -D-> 'b Distribution.t
```

Les flèches indiquent la nature des fonctions : `~D~>` indique un nœud probabiliste, `-D->` un nœud déterministe, et `-S->` indique un argument statique (une constante connue à la compilation). Le premier argument de `infer` est statique car ProbZélus limite l'ordre superieur aux fonctions de flots et n'accepte pas les flots de fonctions de flots. Le deuxième argument de `infer`, ainsi que les arguments de `sample` et `observe`, sont des flots de valeurs.

Méthodes d'inférence. Un programme probabiliste peut être compilé en un *échantillonneur* qui génère un échantillon aléatoire et un score qui mesure la qualité de cet échantillon (i.e., une exécution possible du programme et sa vraisemblance). Dans ce cadre, chaque `sample` tire aléatoirement une valeur dans la distribution associée, et chaque `observe` met à jour le score de l'échantillon. La méthode d'inférence la plus simple, *l'échantillonnage préférentiel*, lance N particules indépendantes. Chaque particule exécute l'échantillonneur pour obtenir une paire (valeur, score). Les résultats sont ensuite normalisés pour approximer la distribution *a posteriori*.

Malheureusement au cours de l'inférence, certaines particules empruntent des chemins d'exécution très improbables qui n'ont que peu d'influence et pénalisent donc l'estimation de la distribution *a posteriori*. Pour remédier à ce problème, les méthodes de *Monte-Carlo séquentielles* (MCS) (aussi appelé *filtres particulaires*) ré-échantillonnent périodiquement l'ensemble de particules au cours de l'exécution [9]. Les particules les plus improbables sont alors éliminées, et les plus probables sont dupliquées.

Enfin, la méthode d'*échantillonnage retardé* [15] permet de combiner des calculs symboliques exacts partiels avec des méthodes d'échantillonnage. En plus du score, les particules maintiennent un *réseau bayésien* qui capture symboliquement les distributions conditionnelles associées à un sous ensemble de variables aléatoires. Les observations peuvent être incorporées en conditionnant analytiquement le réseau. Les particules ne tirent un échantillon que si les calculs analytiques échouent, ou si une valeur concrète est nécessaire (e.g., condition d'un `if`).

Exemple. Considérons un robot qui cherche à estimer sa position courante à partir d'observations bruitées. La Figure 1 présente une modélisation possible de ce problème sous la forme d'un modèle de Markov caché (MMC). À chaque instant, la position courante x_t est une variable latente (cercle blanc) qui ne peut être observée directement. Le robot reçoit des observations y_t (cercle gris) produit par un capteur bruité, par exemple un radar. Chaque flèche indique une dépendance entre deux variables aléatoires. L'observation courante y_t dépend de la position courante x_t, et la position courante dépend de la position à l'instant précédent x_{t-1}. Le code ProbZélus correspondant est le suivant :

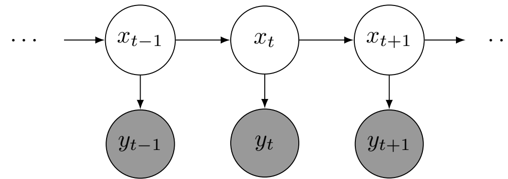


Figure 1: Un modèle de Markov caché. Les variables sont *latentes* (blanc) ou *observées* (gris).

```
let proba hmm (x0, y) = x where
  rec x =  sample (gaussian (x0 -> pre x, speed_x))
  and () = observe (gaussian (x, noise_x), y)

let node main x0 = display(y, x_dist) where
  rec y = sensor()
  and x_dist = infer hmm (x0, y)
```

Le nœud `main` illustre l'utilisation d'`infer` dans un nœud déterministe. Le corps de `main` définit deux flots: `y` les données bruitées envoyées par le `sensor`, et `x_dist` la suite de distributions de positions inférée à partir du modèle `hmm`, de la position initiale `x0` et des observations `y`. À chaque instant, la fonction `display` réalise l'affichage des flots `y` et `x_dist`.

Le nœud probabiliste `hmm`, introduit par le mot-clef `proba`, décrit le modèle de la Figure 1. La première équation indique que la position courante `x` est distribuée normalement autour de la position précédente (l'operateur d'initialisation `->` renvoie la valeur initiale `x0` au premier instant, et la position précédente `pre x` aux instants suivants). La seconde équation indique que l'observation courante `y` est distribuée normalement autour de la position courante `x`. Dans les deux cas, les variances des gaussiennes `speed_x` et `noise_x` sont des constantes.

3 Inférence dans la boucle

ProbZélus permet de mélanger arbitrairement du code Zélus déterministe avec du code probabiliste (à condition de respecter les contraintes de typage indiquées plus haut). L'inférence s'exécute en parallèle avec les processus déterministes. À chaque instant, les composants déterministes peuvent donc utiliser les résultats calculés par l'inférence. C'est ce que nous appelons *l'inférence dans la boucle*.

Pour illustrer cette approche, nous programmons un robot qui peut estimer sa position à partir des commandes qui lui sont données et de la lecture d'un GPS. Les commandes sont calculées en fonction de l'estimation de la position, c'est-à-dire du résultat de l'inférence.

Modularité. Les commandes reçues par le robot sont des accélérations. Pour estimer la position du robot à partir de ces accélérations, nous définissons un nœud `tracker` qui calcule un flot de position `p` et de vitesse `v` en intégrant un flot d'accélération `a` à partir des conditions initiales `p0` et `v0`.

```
let node tracker(p0, v0, a) = p where
  rec p = integr(p0, v)
  and v = integr(v0, a)
```

À cause de facteurs comme les frottements du moteur, l'adhérence des roues ou l'inclinaison du terrain, l'effet de la commande sur la position du robot n'est pas déterministe. Nous pouvons donc considérer ces commandes comme bruitées et utiliser le nœud probabiliste `hmm` présenté section 2 pour prendre en compte ce bruit. On peut ainsi estimer la position `p`, la vitesse `v` et l'accélération `a` à partir de la commande `u` en combinant le nœud probabiliste `hmm` avec le nœud déterministe `tracker`.

```
let proba acc_tracker(p0, v0, a0, u) = p where
  rec a = hmm(a0, u)
  and p = tracker(p0, v0, a)
```

Activation sporadique. Intégrer l'accélération pour estimer la position accumule les erreurs d'estimation. Ainsi, plus le temps passe, plus la position réelle du robot s'éloigne de la position estimée. Pour remédier à ce problème, le robot utilise également un GPS. Comme les mesures GPS sont coûteuses à réaliser, le robot appel le GPS de façon sporadique et s'appuie sur l'accélération pour estimer sa position entre deux mesures. Le nœud ProbZélus suivant intègre les mesures du GPS (également bruitées) au traqueur précédent.

```
let proba gps_acc_tracker(p0, v0, a0, u, gps) = p where
  rec p = acc_tracker(p0, v0, a0, u)
  and () = present gps(p_obs) -> observe(gaussian(p, p_noise), p_obs) else ()
```

À chaque instant, le nœud acc_tracker renvoie une estimation de la position courante p. L'entrée gps est un *signal* qui est émis quand le GPS mesure une nouvelle position. Quand la valeur p_obs est émise sur le signal gps, la construction present exécute son corps et conditionne alors le modèle avec l'observation de cette nouvelle donnée. La variance p_noise est une constante globale qui représente la variance des mesures du GPS.

Boucle de rétroaction. Maintenant que nous avons un modèle qui permet d'estimer la position courante du robot, nous pouvons utiliser la distribution de positions inférées pour mettre à jour la commande.

```
let node robot(p0, v0, a0, target) = (u, p_dist) where
  rec gps = geolocalizer ()
  and u = zero -> controller(pre (mean p_dist), target)
  and p_dist = infer gps_acc_tracker (p0, v0, a0, u, gps)
```

Le nœud geolocalizer génère le signal sporadique gps. Le nœud controller calcule la commande u à partir de la moyenne (mean) de la distribution de positions estimées p_dist. Ce flot de distribution p_dist est inféré à partir du nœud probabiliste gps_acc_tracker qui prend en entrée les conditions initiales, la commande u, et le signal gps. On observe donc bien une boucle de rétroaction entre le contrôleur et l'inférence. Les règles de causalité restent les même que pour Zélus: les cycles de dépendances doivent contenir un délai unitaire (pre). Cela n'empêche pas d'avoir dans un même instant du code probabiliste qui dépend de code déterministe et du code déterministe qui dépend de code probabiliste.

Structures de contrôle. ProbZélus offre de nombreuse structures de contrôle : signaux d'activation, ré-initialisation modulaire, et automates hiérarchiques [7]. Il est ainsi possible de programmer dans un formalisme proche des diagrammes par blocs [11], une notation classique des systèmes embarqués.

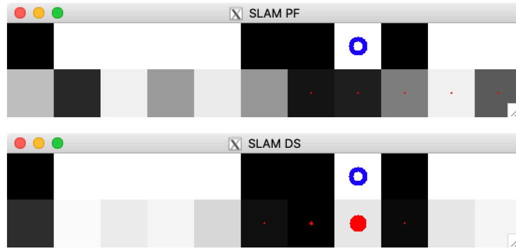
Nous avons vu avec le nœud gps_acc_tracker que les structures de contrôle comme present peuvent être utilisées à l'intérieur des nœuds probabilistes. Ces structures de contrôle peuvent également être utilisées à l'extérieur pour contrôler l'inférence. Par exemple, notre robot peut être utilisé pour effectuer une tâche lorsqu'il atteint une certaine position.

```
let node task_bot(p0, v0, a0, target) = cmd where
  rec automaton
  | Go -> do cmd, p_dist = robot(p0, v0, a0, target)
          until (probability p_dist (target - epsilon) (target + epsilon) > 0.9)
          then Task
  | Task -> do cmd = task_controller() done
```

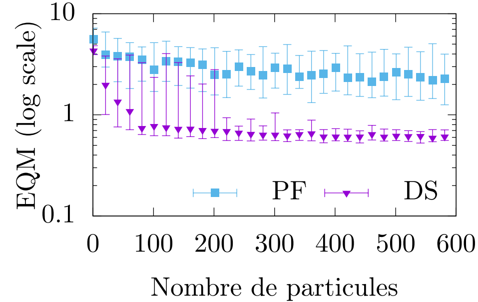
Dans l'état Go, la commande est celle calculée par le contrôleur robot, qui renvoie aussi la distribution de positions courantes. Lorsque la probabilité que le robot soit proche de la cible (entre target - espilon, and target + epsilon) est supérieure à 0.9, le contrôleur entre dans l'état Task où la commande est calculée par le nœud task_controller.

4 Inférence semi-symbolique

Les méthodes d'inférence par échantillonnage permettent d'obtenir des résultats satisfaisants pour les exemples précédents avec un nombre raisonnable de particules (≤ 1000). Malheureusement, ces méthodes peuvent échouer sur des modèles plus complexes où l'approche semi-symbolique de l'*échantillonnage retardé* peut donner de bons résultats.



(a) Pour chaque capture d'écran, la ligne supérieure montre la carte, et le cercle bleu la position exacte du robot. La ligne inférieure représente la carte inférée où le niveau de gris indique la probabilité pour la case d'être noire et les points rouges la probabilité de présence du robot sur la case.



(b) Précision en fonction du nombre de particules. Les points représentent la médiane de 40 exécutions, les barres d'erreur les 90% et 10% quantiles.

Figure 2: Exécution du SLAM avec filtre particulaire (PF) et échantillonnage retardé (DS).

Dans cette section, nous illustrons cette situation avec un contrôleur de robot capable d'inférer à la fois sa position courante et une carte de son environnement. Un problème classique de localisation et cartographie simultanées (*Simultaneous Location And Mapping*) [14].

SLAM. Considérons le cas simple où le robot évolue dans un monde discret à une dimension et chaque position correspond à une case noire ou blanche. Un robot peut se déplacer de gauche à droite et il peut observer la couleur de la case sur laquelle il se tient à l'aide d'un capteur. Il y a deux sources d'incertitude : (1) Les roues du robot sont glissantes, le robot peut donc parfois rester sur place en pensant se déplacer. (2) Le capteur fait des erreurs de lecture, et peut inverser les couleurs. Le contrôleur cherche à inférer la carte (couleur des cases) et la position courante du robot (Figure 2a).

Le robot maintient une carte où chaque case est une variable aléatoire qui représente la probabilité d'être noire ou blanche (niveau de gris dans la Figure 2a). La distribution *a priori* de ces variables aléatoires est uniforme entre 0 et 1 (une distribution Beta(1,1)) :

```
let proba beta_priors _ = sample (beta (1., 1.))
```

Le robot démarre de la position `x0` et reçoit à chaque instant une commande `Right` ou `Left`. Il se déplace alors vers la gauche ou la droite suivant la commande avec une probabilité de 10% de rester sur place (modélisée par une loi de Bernoulli de paramètre 0.1).

```
let proba move (x0, cmd) = x where
  rec slip = sample (bernoulli 0.1)
  and xp = x0 -> pre x
  and x = match cmd with
      | Right -> min max_pos (if slip then xp else xp + 1)
      | Left  -> max min_pos (if slip then xp else xp - 1)
      end
```

On modélise de la même manière le capteur avec un probabilité d'erreur de lecture de 10% :

```
let proba read obs = if sample (bernoulli 0.1) then not obs else obs
```

À chaque instant, le robot calcule sa position `x` et récupére la valeur de la carte associée à cette position `c`. On suppose alors que l'observation `o` suit une loi de Bernoulli paramétrée par `c`.

```
let proba slam (obs, cmd) = (map, x) where
  rec init map = Array.init (max_pos + 1) beta_priors ()
  and x = move (0, cmd)
  and o = read (obs)
  and c = Array.get map x
  and () = observe (bernoulli (c, o))
```

Évaluation. On peut constater sur la partie haute de la Figure 2a, que le SLAM est un modèle particulièrement difficile pour le filtre particulaire. Les résultats sont beaucoup plus convaincants sur la partie basse de la Figure 2a qui utilise l'échantillonnage retardé.

Plus précisément, la Figure 2b présente la précision de l'estimation de la position et de la carte après 1500 instants pour un robot qui se déplace de gauche à droite sur une carte de taille 11. La précision est définie comme la somme des erreurs quadratiques moyennes (EQM) de chacune des variables aléatoires du modèle.

Comparé au filtre particulaire, l'échantillonnage retardé est capable d'exploiter la relation de conjugaison entre la distribution *a priori* des cases de la carte (Beta), et les observations bruitées (Bernoulli) pour mettre à jour la distribution des cases en incorporant les observations de manière analytique. Ainsi, si $p(c) = \mathrm{Beta}(\alpha, \beta)$ et $p(o|c) = \mathrm{Bernoulli(c)}$, on obtient selon l'observation o : $p(c|o = \mathit{true}) = \mathrm{Beta}(\alpha + 1, \beta)$, ou $p(c|o = \mathit{false}) = \mathrm{Beta}(\alpha, \beta + 1)$.

En revanche, le calcul symbolique échoue pour la position du robot qui ne peut être estimée qu'à partir d'un ensemble de particules (il n'y a pas de relation de conjugaison entre la position courante et la position précédente). La courbe DS n'atteint donc sa précision maximale que pour $100-200$ particules. L'exemple du SLAM illustre donc l'avantage d'une méthode semi-symbolique combinant calculs exacts et échantillonnage.

5 Travaux connexes

Programmation probabiliste. Ces dernières années, les langages de programmation probabilistes ont suscité un intérêt croissant. Certains langages comme BUGS [13], Stan [6] ou Augur [12] offrent des techniques d'inférence optimisées pour un sous-ensemble contraint de modèles. D'autres comme WebPPL [10], Edward [21], Pyro [4] ou Birch [16] permettent de spécifier des modèles arbitrairement complexes. Par rapport à ces langages, ProbZélus peut être utilisé pour programmer des *modèles parallèles réactifs* et qui ne terminent pas, et où l'inférence s'effectue en interaction avec des composants déterministes.

Non-déterminisme dans les Langages réactifs. Lutin est un langage pour décrire et simuler des systèmes réactifs non-déterministes [19] mais ne permet pas d'inférer des paramètres à partir d'observations. ProPL [17] est un langage pour décrire des processus probabilistes qui évoluent dans le temps. Comparé à ProbZélus, ProPL se concentre sur une classe restreinte de modèles : les *réseaux dynamiques bayésiens* (DBN) et s'appuie sur les techniques d'inférence standard pour les DBNs. CTPPL [18] est un langage pour décrire des processus probabilistes en temps continu. La durée nécessaire à un sous-processus peut être spécifiée par un modèle probabiliste. Ces modèles ne peuvent pas être exprimés en ProbZélus qui repose sur le modèle de temps synchrone discret.

6 Conclusion

Modéliser des comportements non-déterministes est un aspect fondamental des systèmes embarqués qui évoluent dans des environnements bruités et incertains. Les langages synchrones pourtant introduits pour la conception de tels systèmes n'offraient jusqu'alors que peu de support pour prendre en compte l'incertitude.

Dans cet article nous avons illustré les avantages offerts par ProbZélus, le premier langage synchrone probabiliste. ProbZélus permet d'écrire des modèles probabilistes réactifs capables d'inférer des paramètres latents à partir d'observations. L'inférence s'exécute en interaction avec des composants déterministes ce qui permet de programmer des systèmes avec *inférence dans la boucle*. Enfin, ProbZélus offre plusieurs méthodes d'inférence automatique qui combinent calculs symboliques et échantillonnage.

Bibliographie

[1] Karl J Åström. *Introduction to stochastic control theory.* Courier Corporation, 2012.

[2] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. Reactive probabilistic programming. *CoRR*, abs/1908.07563, 2019.

[3] Gérard Berry. Real time programming: Special purpose or general purpose languages. *Information Processing*, 89:11–17, 1989.

[4] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul A. Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research*, 20:28:1–28:6, 2019.

[5] Timothy Bourke and Marc Pouzet. Zélus, a Synchronous Language with ODEs. In *International Conference on Hybrid Systems: Computation and Control*, 2013.

[6] Bob Carpenter, Andrew Gelman, Matthew D Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1):1–37, 2017.

[7] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software*, 2005.

[8] Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet. Scade 6: A formal language for embedded critical software development. In *International Symposium on Theoretical Aspects of Software Engineering*, 2017.

[9] Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential Monte Carlo samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(3):411–436, 2006.

[10] Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org, 2014. Accessed: 2019-10-15.

[11] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[12] Daniel Huang, Jean-Baptiste Tristan, and Greg Morrisett. Compiling Markov chain Monte Carlo algorithms for probabilistic modeling. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[13] David Lunn, David Spiegelhalter, Andrew Thomas, and Nicky Best. The bugs project: Evolution, critique and future directions. *Statistics in medicine*, 28(25):3049–3067, 2009.

[14] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. FastSLAM: A factored solution to the simultaneous localization and mapping problem. In *AAAI National Conference on Artificial Intelligence*, 2002.

[15] Lawrence M. Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B. Schön. Delayed sampling and automatic rao-blackwellization of probabilistic programs. In *AISTATS Proceedings of Machine Learning Research*, 2018.

[16] Lawrence M. Murray and Thomas B. Schön. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control*, 46:29–43, 2018.

[17] Avi Pfeffer. Functional specification of probabilistic process models. In *AAAI National Conference on Artificial Intelligence*, 2005.

[18] Avi Pfeffer. CTPPL: A continuous time probabilistic programming language. In *International Jont Conference on Artifical Intelligence*, 2009.

[19] Pascal Raymond, Yvan Roux, and Erwan Jahier. Lutin: A language for specifying and executing reactive scenarios. *EURASIP Journal of Embedded Sytems*, 2008.

[20] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. Design and implementation of probabilistic programming language Anglican. In *Symposium on the Implementation and Application of Functional Programming Languages*, 2016.

[21] Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. Deep probabilistic programming. In *International Conference on Learning Representations*, 2017.

[22] Yi Wu, Lei Li, Stuart J. Russell, and Rastislav Bodík. Swift: Compiled inference for probabilistic programming languages. In *International Joint Conference on Artificial Intelligence*, 2016.