Programming Reactive Probabilistic Applications

GUILLAUME BAUDART, MIT-IBM Watson AI Lab, IBM Research
LOUIS MANDEL, MIT-IBM Watson AI Lab, IBM Research
MARC POUZET, École Normale Supérieure
ERIC ATKINSON, MIT
BENJAMIN SHERMAN, MIT
MICHAEL CARBIN, MIT

1 INTRODUCTION

Synchronous languages were introduced 30 years ago for the design of real-time embedded systems. They are based on the *synchronous parallelism* model [2]. Several synchronous languages have been proposed, most notably Scade [7] an industrial language used to design and implement critical real-time software (flight control, engine control, for example).

Scade is a *data-flow* language à la Lustre: input/output signals are infinite sequences (or *streams*), a system (or *node*) is a function on streams, and all streams progress together, step by step, in a *synchronous* manner. This programming style is well-suited to express classical control blocks (relays, filters, PID controllers, etc.), a discrete model of the environment, and interaction loops between these two components. The following code implements a PID controller (proportional, integral, derived) in Zelus, an academic language close to Scade [4].[1]

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

The node `pid` defines a stream of commands `u` from a stream of setpoints `r` and a stream of measures `y`. The command is the weighted sum of three expressions (proportional, integral, and derivative) applied to the error between the setpoint and the measurement. The weights `p`, `i`, and `d` are constants and the calls to the node `integr(0., e)` and `deriv(e)` compute respectively the integral (initialized to `0.`) and the derivative of `e`.

Synchronous languages offer limited support for modeling of non-determinism and uncertainty of the environment, which are ubiquitous in a real system. A controller often only has a partial, noisy view of its surroundings, and the behavior of the system itself is often subject to disruption.

Probabilistic programming languages can describe probabilistic models and automatically *infer* the distribution of *latent* (i.e., unobserved) parameters from *observations* (i.e., inputs). A common approach [3, 8, 14, 18–20] is to extend a general-purpose programming language with three new constructs: (1) `x = sample(d)` introduce a *latent* random variable `x` of distribution `d`; (2) `observe(d, y)` measures the *likelihood* of an *observation* `y` with respect to a distribution `d`; (3) `infer m obs` calculates the distribution of output values of a program or *model* `m` *knowing* the observations `obs` given as input. Probabilistic programming languages offer a variety of automatic inference techniques ranging from exact symbolic computation, to sampling approximations (Monte Carlo methods). But none of these languages offers the support and the associated guarantees given by synchronous languages to design embedded reactive systems (data-flow programming, execution with bounded resources, and absence of deadlocks).

In this article, we present ProbZelus [1], a probabilistic extension of Zelus. ProbZelus allows developers to combine the constructions of a synchronous reactive language and the constructions of

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

a probabilistic language — `sample`, `observe` and `infer` — to develop *probabilistic reactive applications*. We illustrate with examples the advantages offered by ProbZelus:

(1) Programming reactive models: a trajectory detector from noisy observations (Section 2).
(2) Inference in the loop: a robot controller guided by the result of a probabilistic trajectory detector (Section 3).
(3) Semi-Symbolic Inference: a more complex robot model, able to infer both its position and the map of its environment (Section 4).

2 PROBABILISTIC REACTIVE PROGRAMMING

In ProbZelus, probabilistic models are special nodes introduced by the keyword `proba`. The constructs `sample` and `observe` can only be invoked within a probabilistic node. The operator `infer` takes a probabilistic node and produces a deterministic result the *posterior* distribution defined by the model. A type checker statically checks these constraints. The signatures associated with probabilistic constructs are as follows:

```
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
```

The arrows indicate the nature of the functions: `~D~>` indicates a probabilistic node, `-D->` a deterministic node, and `-S->` indicates a static argument (a constant known at compilation). The first argument of `infer` is static because ProbZelus limits higher-order operations to stream functions (not flows of stream functions). The second argument of `infer`, and the arguments of `sample` and `observe`, are streams of values.

Example. Consider a robot that seeks to estimate its current position from noisy observations. Figure 1 presents a possible model of this problem in the form of a hidden Markov model (HMM). At every moment, the current position x_t is a latent variable (white circle) that can not be observed directly. The robot receives observations y_t (gray circle) produced by a noisy sensor, for example a radar. Each arrow indicates a dependency between two random variables. The corresponding ProbZelus code is:

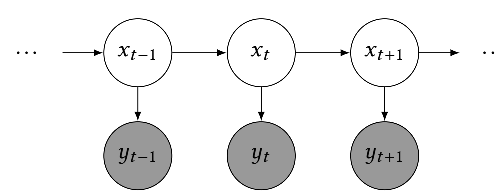


Fig. 1. A hidden Markov model. The variables are *latent* (white) or *observed* (gray).

```
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)
```

The `main` node illustrates the use of `infer` in a deterministic node. The body of `main` thus defines streams: `y`, the noisy data sent by the `sensor`; and `x_dist`, the series of inferred position distributions from the `hmm`, the initial position `x0` and the observations `y`. The expression `infer` returns the stream of distributions over the output values of a probabilistic node (here `x`). At each step, this yields the current distribution given past observations. At every moment, the `display` function displays the streams `y` and `x_dist`.

The probabilistic node `hmm`, introduced by the keyword `proba`, describes the model in Figure 1. The first equation indicates that the current position `x` is normally distributed around the previous position (the initialization operator `->` in this context returns the initial value `x0` at the first instant, and then the previous position `pre x`). The second equation indicates that the current observation `y` is normally distributed around the current position `x`. In both cases, the Gaussian variances `speed_x` and `noise_x` are constants.

Inference ProbZelus offers a set of black-box inference techniques to compute the posterior distribution of reactive models. These techniques are adapted to the reactive settings where computations never stop. The set of inference techniques includes classic sequential Monte-Carlo methods, or particle filters; and *delayed sampling*, a recently proposed implementation of Rao-Blackwellized particles filters [13] that combines partial exact symbolic computations with sampling methods.

Under delayed sampling, in addition to a score, each particle maintains a *Bayesian network* that symbolically captures the conditional distributions associated with a subset of random variables. Delayed sampling then exploits conjugacy relations between variables to analytically incorporate observations to the network whenever possible. Particles draw a sample only if analytical computations fail, or if a concrete value is needed (e.g., for the condition of a `if` statement).

3 INFERENCE IN THE LOOP

ProbZelus allows you to arbitrarily mix deterministic Zelus code with probabilistic code (provided that you follow the typing constraints above). The inference runs in parallel with the deterministic processes. At each step, the deterministic components can use the results computed by inference and vice versa. This is what we call *inference in the loop*.

To illustrate this approach, we use the example of a robot that can estimate its position from its previous commands and observations from a GPS. At each step, the command depends on the previous position estimation, that is, the result of the inference.

Modularity. The commands received by the robot are accelerations. To estimate the position of the robot from these accelerations, we define a node `tracker` which computes a stream of positions `p` and velocity `v` by integrating the acceleration stream `a` from the initial conditions `p0` and `v0`.

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

Due to factors such as engine friction, wheel adhesion or terrain inclination, the effect of the control on the position of the robot is not deterministic. We can therefore consider these commands noisy and use the probabilistic node `hmm` presented in Section 2 to take this noise into account. We can thus model the position `p`, the velocity `v` and the acceleration `a` from the command `u` by combining the probabilistic node `hmm` with the deterministic node `tracker`.

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

Sporadic activation. Integrating the acceleration to estimate the position accumulates errors. As time passes, the distribution over positions becomes increasingly spread out. To mitigate this issue, the robot can additionally use a GPS. Since GPS measurements are expensive, the robot only sporadically calls the GPS and relies on the acceleration to estimate its position between two measurements. The following node adds (noisy) GPS observations to the previous model.

```
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)
```

At every step, the `acc_tracker` node returns an estimate of the current position `p`. The entry `gps` is a *signal* that is emitted when the GPS measures a new position. When a value `p_obs` is emitted on the signal `gps`, the `present` construct executes its body and further conditions the model using this new observation. The variance `p_noise` is a global constant.

Feedback loop. Now that we have a model that estimates the robot's position given previous commands and the GPS, we can use the inferred position distribution to update the command.

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

The `geolocalizer` node generates the sporadic `gps` signal. The `controller` node computes the command `u` from the mean (`mean`) of the estimated position distribution `p_dist`. This distribution stream `p_dist` is inferred from the probabilistic node `gps_acc_tracker` defined above which takes as input the initial conditions, the command `u`, and the signal `gps`. We thus have a feedback loop between the controller and inference.

Control structures. ProbZelus offers numerous control structures: activation signals, modular re-initialization, and hierarchical automata [6]. It is thus possible to program in a formalism close to block diagrams [9], a classic notation for embedded systems.

The node `gps_acc_tracker` illustrates that control structures like `present` can be used inside probabilistic nodes. These control structures can also be used externally to control inference. For example, our robot can be used to perform a task when it reaches a certain 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
```

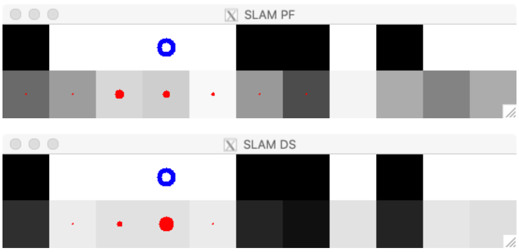
In the state `Go`, the command is the one computed by the controller `robot`, which also returns the distribution of current positions. When the probability that the robot is close to the target (between `target - espilon`, and `target + epsilon`) is greater than 0.9, the controller enters the state `Task` where the command is computed by the node `task_controller`.

4 SEMI-SYMBOLIC INFERENCE

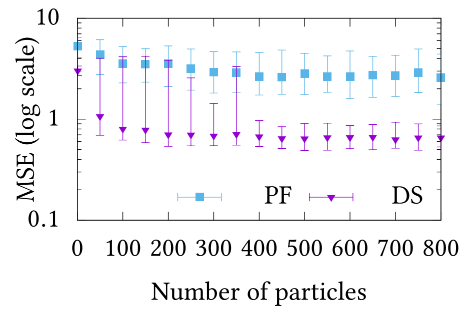
Sampling-based inference methods obtain good results for the preceding examples with a relatively low number of particles ($\leq$ 1000). Unfortunately, these methods may use prohibitively many particles on more complex models where the semi-symbolic approach of *delayed sampling* can give much better results.

In this section, we illustrate this situation with a robot controller that can infer both its current position and a map of its environment. This is a classic problem of *simultaneous location and mapping* (SLAM) [12].

SLAM. Consider the simple case where the robot evolves in a discrete one-dimensional world and each position corresponds to a black or white cell. A robot can move from left to right and



(a) For each screenshot, the top line shows the true map, and the blue circle the exact position of the robot. The lower line represents the inferred map where the gray level indicates the probability for the cell to be black and the size of the red dots indicates the probability of the robot's presence in the cell.



(b) Precision according to the number of particles. The dots represent the median of 100 executions, the error bars the 90% and 10% quantiles.

Fig. 2. Execution of SLAM with particle filter (PF) and delayed sampling (DS)

can observe the color of the cell on which it stands with a sensor. There are two sources of uncertainty: (1) The robot's wheels are slippery, so the robot can sometimes unknowingly stay at the same position. (2) The sensor is imperfect and can misread the colors. The controller tries to infer the map (cell color) and the current position of the robot (Figure 2a).

The robot maintains a map where each cell is a random variable that represents the probability of being black or white (gray level in the Figure 2a). The *prior* distribution of these random variables is a Beta(1,1) distribution:

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

The robot starts from the position `x0` and receives at each step a command `Right` or `Left`. It then moves to the left or right following the command with a 10% probability of staying at the same place (modeled by a Bernoulli distribution of parameter 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
```

At each instant, the robot computes its position `x` and retrieves the value of the map for this position `c`. We assume that the observation `o` follows a Bernoulli distribution parameterized by `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 c = Array.get map x
  and () = observe (bernoulli (c, obs))
```

Evaluation. As we can see from the top of Figure 2a, SLAM is a particularly difficult model for the particle filter. The results are much more accurate on the bottom part of Figure 2a which uses delayed sampling.

Quantitatively, Figure 2b presents the accuracy of the estimated position and the map after 2000 steps for a robot that moves from left to right on a map of size 11 (as in section 3, the controller uses the estimated position to compute the command). Precision is defined as the sum of the mean squared errors (MSE) of each of the random variables in the model.

Compared to the particle filter, delayed sampling exploits the conjugacy relation between the *prior* distribution of the map cells (Beta), and the observations (Bernoulli) to update the cell distribution using the observations. For instance, if $p(c) = \text{Beta}(\alpha, \beta)$ and $p(o|c) = \text{Bernoulli}(c)$, depending on the observation o we have: $p(c|o = \mathit{true}) = \text{Beta}(\alpha + 1, \beta)$, and $p(c|o = \mathit{false}) = \text{Beta}(\alpha, \beta + 1)$.

On the other hand, delayed sampling cannot exploit any conjugacy relation to compute the position distribution and falls back on a estimation based on sampling a set of particles. This is reflected in Figure 2b where the delayed sampling graph (DS) only reaches its maximum precision after 400 particles. The SLAM example thus illustrates the semi-symbolic approach combining exact calculations and sampling.

5 RELATED WORK

Probabilistic programming. In recent years, probabilistic programming languages have attracted increasing interest. Some languages like BUGS [11], Stan [5] or Augur [10] offer optimized inference techniques for a constrained subset of models. Others like WebPPL [8], Edward [19], Pyro [3] or Birch [14] allow to specify arbitrarily complex models. With respect to these languages, ProbZelus can be used to program *reactive parallel models* that do not terminate, and where inference is performed in interaction with deterministic components.

Non-determinism in Reactive Languages Lutin is a language for describing and simulating non-deterministic reactive systems [17] but does not allow to infer parameters from observations. ProPL [15] is a language to describe probabilistic processes that evolve over time. Compared to ProbZelus, ProPL focuses on a restricted class of *Bayesian dynamic networks* (DBN) models and uses standard inference techniques for DBNs. CTPPL [16] is a language for describing probabilistic processes in continuous time. The time required for a subprocess can be specified by a probabilistic model. These models can not be expressed in ProbZelus which is based on the discrete synchronous time model.

6 CONCLUSION

Modeling non-deterministic behaviors is a fundamental aspect of embedded systems that evolve in noisy and uncertain environments. Synchronous languages, introduced for the design of such systems, have until now offered little support to take into account probabilistic uncertainty.

In this article, we have illustrated the advantages offered by ProbZelus, the first probabilistic synchronous language. ProbZelus allows to write probabilistic reactive models able to infer latent parameters from observations. Inference runs in interaction with deterministic components, which makes it possible to program systems with *inference in the loop*. Finally, ProbZelus offers several methods of automatic inference that combine symbolic computations and particle filtering.

REFERENCES

[1] Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. 2019. Reactive Probabilistic Programming. *CoRR* abs/1908.07563 (2019). arXiv:1908.07563 http://arxiv.org/abs/1908.07563

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

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

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

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

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

[7] Jean-Louis Colaco, Bruno Pagano, and Marc Pouzet. 2017. Scade 6: A Formal Language for Embedded Critical Software Development. In *International Symposium on Theoretical Aspects of Software Engineering*.

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

[9] David Harel. 1987. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.* 8, 3 (1987), 231–274.

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

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

[12] Michael Montemerlo, Sebastian Thrun, Daphne Koller, and Ben Wegbreit. 2002. FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem. In *AAAI National Conference on Artificial Intelligence*.

[13] Lawrence M. Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B. Schön. 2018. Delayed Sampling and Automatic Rao-Blackwellization of Probabilistic Programs. In *AISTATS Proceedings of Machine Learning Research*.

[14] Lawrence M. Murray and Thomas B. Schön. 2018. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control* 46 (2018), 29–43. https://doi.org/10.1016/j.arcontrol.2018.10.013

[15] Avi Pfeffer. 2005. Functional Specification of Probabilistic Process Models. In *AAAI National Conference on Artificial Intelligence*.

[16] Avi Pfeffer. 2009. CTPPL: A Continuous Time Probabilistic Programming Language. In *International Jont Conference on Artifical Intelligence*.

[17] Pascal Raymond, Yvan Roux, and Erwan Jahier. 2008. Lutin: A Language for Specifying and Executing Reactive Scenarios. *EURASIP Journal of Embedded Sytems* (2008). https://doi.org/10.1155/2008/753821

[18] David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. In *Symposium on the Implementation and Application of Functional Programming Languages*.

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

[20] Yi Wu, Lei Li, Stuart J. Russell, and Rastislav Bodík. 2016. Swift: Compiled Inference for Probabilistic Programming Languages. In *International Joint Conference on Artificial Intelligence*.