Probabilistic Reactive Programming

Louis Mandel Guillaume Baudart Avraham Shinnar Kiran Kate Martin Hirzel
IBM Research

Abstract

Modeling reactive systems with uncertainty is challenging because reactive systems typically run without terminating, interact with an external environment, and evolve during execution. To facilitate the modeling of such systems, we propose a programming language that mixes features from both probabilistic and reactive languages. It can express probabilistic reactive models with complex dynamics.

1 Probabilistic Reactive Models

A reactive system is a system in continuous interaction with its environment. Reactive programming languages are domain specific languages designed to facilitate the implementation of reactive systems, offering intuitive formalisms, dedicated analyses, and optimized code generation.

We propose to extend a reactive programming language to allow probabilistic programming of reactive models. Examples of probabilistic reactive systems include online time-series prediction, agent-based systems, or infrastructure self-tuning. This paper proposes a probabilistic extension to ReactiveML,[1] which is itself a reactive extension of OCaml[2] that supports parallel composition of processes, communication through signals, preemption, and suspension. ReactiveML is based on the synchronous model [1] that provides a notion of global logical time defined as a succession of instants. Following [7, 8], we add probabilistic constructs to ReactiveML: `sample`, to draw a value from a distribution, `factor`, to penalize execution paths based on observations, and `infer`, to compute the distribution defined by a process. We also add the construct `propose` to expose the state of a model during the inference.

Consider the example of a Hidden Markov Model (HMM) that continuously tracks the positions p_t of a moving object from noisy observations o_t (Figure 1).

```
1 let rec process hmm obs p_prev =
2   await obs([o_t]) in
3   let p_t = sample (sph_gaussian p_prev speed) in
4   factor (score (sph_gaussian p_t noise) o_t);
5   propose p_t;
6   run hmm obs p_t
```

A *process* is a function that runs over multiple time instants. Line 1 defines a recursive process `hmm` with two arguments `obs` and `p_prev`. A *signal* is a communication channel between processes. Line 2 waits to receive a new observation `o_t` (o_t in Figure 1) from input signal `obs`. After reception, Line 3 samples the current position `p_t` (p_t) from a Gaussian centered on the previous position `p_prev` (p_{t-1}), and Line 4 conditions on the current observation. The `score` function returns the log-probability of a sample, which is used here as a measure of how far the sample is from the observation. The values `speed` and `noise` are parameters of the `hmm` defined as global constants. Line 5 emits the position on an implicit output signal for use as a distribution outside of the `hmm` process. Line 6 re-launches the process for the next step, creating an infinite loop.

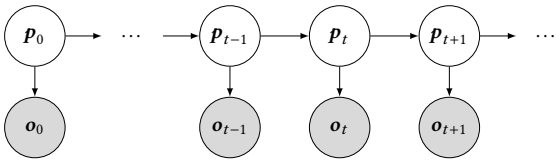
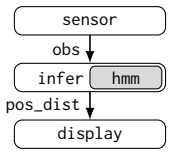


Figure 1. A simple Hidden Markov Model (HMM)

The `main` process drives the inference and manages communication with the environment.

```
1 let process main =
2   signal obs in
3   signal pos_dist in
4   run sensor obs ||
5   infer pos_dist (hmm obs [0.; 0.]) ||
6   run display pos_dist
```



Lines 2–3 declare the communication channels and Lines 4–6 is the parallel composition of three processes. Line 5 launches probabilistic inference on the `hmm` process. The first argument of `infer` is a signal on which the inferred distribution of positions is emitted after each observation. Lines 4 and 6 realize the interface with the environment via two processes `sensor` and `display` running in parallel with the inference. The process `sensor` emits noisy positions observed from the environment on signal `obs`, and the process `display` visualizes the distribution of inferred positions.

2 Design Choices

In the previous example probabilistic constructs are used inside arbitrary ReactiveML code. Compared to other probabilistic languages where inference is executed on terminating functions without side effects, the main challenge is to run the inference on non-terminating probabilistic processes that communicate with the environment.

Instantaneous models. We initially designed a less expressive version of probabilistic ReactiveML, but determined that it was insufficient. The previous version limited inference to *instantaneous* functions, that is, pure OCaml code without reactive constructs. Adding this feature to ReactiveML

[1] http://reactiveml.org
[2] https://ocaml.org

is straightforward using inference techniques from classic probabilistic languages [11].

```
1 let rec process hmm' obs pos_dist last_dist =
2   await obs(o_t) in
3   let dist =
4     infer (fun () ->
5       let p_prev = sample (last_dist) in
6       let p_t = sample (sph_gaussian p_prev speed) in
7       factor (score (sph_gaussian p_t noise) o_t);
8       p_t)
9   in
10  emit post_dist dist
11  run hmm' obs pos_dist dist
```

This version explicitly triggered inference inside the process (Line 4), putting additional burden on the programmer. Each step computes the current position by first sampling the previous position from the distribution inferred at the previous step (Line 5).

Hybrid applications. In our latest design, we focus instead on hybrid applications where probabilistic reactive models can be inferred in parallel with deterministic processes. We thus make the following design choices to allow communication between the two parts of the application.

First, a probabilistic model must be able to receive inputs from an evolving environment. We support that by letting models receive values from a signal during inference (e.g., `await obs([o_t])`, in `hmm` Line 2).

Second, instead of awaiting the final result of the inference, deterministic processes running in parallel need access to intermediate results. We thus introduce a new `propose` construct (e.g., `propose p_t`, in `hmm` Line 5) to output a probabilistic value. We forbid any other side effects (e.g. print statements) within models. Emitting a signal whose scope is outside of the inference is also a side effect. We can statically reject such programs using the type system presented in [9].

Third, since probabilistic models communicate with reactive processes during inference, they cannot rollback past actions. We make this choice explicit by relying on the synchronous model of execution on which ReactiveML is built. Predictions at a given instant cannot be updated at subsequent instants. Since it is the programmer who determines the granularity of instants, they specify when predictions are committed. This choice of no rollback implies that there is no need to store the unbounded input history across multiple instants to resample a trace. The application can thus be executed in bounded memory.

Implementation. Our experimental implementation relies on lightweight sampling inference methods à la WebPPL [7, 13]. Both the probabilistic and the reactive features are implemented using a rewrite-based semantics, which rewrites from a current state and a program into a new state and the remainder of the program to-be-executed. Making the remainder, or continuation, explicit enables the language runtime to explore different execution paths starting from `sample` statements, and to resume execution from `await` statements when a value is available on a signal.

3 Related Work

There are two relevant kinds of related work: programming languages with probabilistic and reactive features and probabilistic inference techniques that may be applicable to probabilistic reactive programming.

Lutin is a language for describing probabilistic reactive systems for testing and simulation [10], but while Lutin supports weighted sampling, it does not support inference. Wasserkrug et al. extend a pattern-based event language with probabilistic features [12], but the language cannot express general probabilistic models. Probabilistic programming has been used to implement agent-based models [6], but inference is only performed on terminating functions.

Sequential Monte Carlo methods [5] and streaming inference in Bayesian parametric as well as nonparametric models [3, 4] are suitable for reactive systems. But they are not directly applicable because they expect the inferred function to terminate. They could be adapted to our setting by using logical instants as "termination" points because we forbid resampling across multiple instants. Online learning handles non-terminating reactive systems, but, aside from naive Bayes, it does not use probabilistic models [2].

4 Conclusion

This paper introduces the first reactive probabilistic programming language, supporting online inference on streaming data. Our language can be used to write hybrid applications where the inference is executed on non-terminating functions and runs in parallel with deterministic processes.

References

[1] Albert Benveniste, Paul Caspi, Stephen A. Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert de Simone. 2003. The Synchronous Languages 12 Years Later. *Proc. IEEE* 91, 1 (Jan. 2003), 64–83. https://doi.org/10.1109/JPROC.2002.805826

[2] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. 2010. MOA: Massive Online Analysis. *Journal of Machine Learning Research (JMLR)* 11 (May 2010), 1601–1604. http://www.jmlr.org/papers/v11/bifet10a.html

[3] Tamara Broderick, Nicholas Boyd, Andre Wibisono, Ashia C. Wilson, and Michael I. Jordan. 2013. Streaming Variational Bayes. In *Conference on Neural Information Processing Systems (NIPS)*. 1727–1735. http://papers.nips.cc/paper/4980-streaming-variational-bayes

[4] Trevor Campbell, Julian Straub, John W. Fisher, III, and Jonathan P. How. 2015. Streaming, Distributed Variational Inference for Bayesian Nonparametrics. In *Conference on Neural Information Processing Systems (NIPS)*. 280–288. http://papers.nips.cc/paper/5876-streaming-distributed-variational-inference-for-bayesian-nonparametrics

[5] Arnaud Doucet, Simon J. Godsill, and Christophe Andrieu. 2000. On Sequential Monte Carlo Sampling Methods for Bayesian Filtering. *Statistics and Computing* 10, 3 (2000), 197–208. https://doi.org/10.1023/A:1008935410038

[6] Owain Evans, Andreas Stuhlmüller, John Salvatier, and Daniel Filan. 2017. Modeling Agents with Probabilistic Programs. http://agentmodels.org. (2017). Accessed: 2018-6-27.

[7] Noah D Goodman and Andreas Stuhlmüller. 2014. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org. (2014). Accessed: 2018-6-11.

[8] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 2014. Probabilistic Programming. In *ICSE track on Future of Software Engineering (FOSE)*. 167–181. https://doi.org/10.1145/2593882.2593900

[9] Louis Mandel, Cédric Pasteur, and Marc Pouzet. 2013. Time Refinement in a Functional Synchronous Language. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming (PPDP '13)*. 169–180. http://doi.acm.org/10.1145/2505879.2505904

[10] 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

[11] Daniel Ritchie, Andreas Stuhlmüller, and Noah D. Goodman. 2016. C3: Lightweight Incrementalized MCMC for Probabilistic Programs using Continuations and Callsite Caching. In *Conference on Artificial Intelligence and Statistics (AISTATS)*. 28–37. http://proceedings.mlr.press/v51/ritchie16.html

[12] Segev Wasserkrug, Avigdor Gal, Opher Etzion, and Yulia Turchin. 2008. Complex Event Processing over Uncertain Data. In *Conference on Distributed Event-Based Systems (DEBS)*. 253–264. https://doi.org/10.1145/1385989.1386022

[13] David Wingate, Andreas Stuhlmueller, and Noah Goodman. 2011. Lightweight Implementations of Probabilistic Programming Languages via Transformational Compilation. In *Conference on Artificial Intelligence and Statistics (AISTATS)*. 770–778. http://proceedings.mlr.press/v15/wingate11a.html