Vate: Runtime Adaptable Probabilistic Programming for Java

Daniel Goodman
Oracle Labs
Daniel.Goodman@oracle.com

Adam Pocock
Oracle Labs

Jason Peck
Oracle Labs

Guy Steele
Oracle Labs

Abstract

Inspired by earlier work on Augur, Vate is a probabilistic programming language for the construction of JVM based probabilistic models with an Object-Oriented interface. As a compiled language it is able to examine the dependency graph of the model to produce optimised code that can be dynamically targeted to different platforms. Using Gibbs Sampling, Metropolis–Hastings and variable marginalisation it can handle a range of model types and is able to efficiently infer values, estimate probabilities, and execute models.

Keywords: Probabilistic Programming, Java, Augur, MCMC, Parallel Programming

ACM Reference Format:
Daniel Goodman, Adam Pocock, Jason Peck, and Guy Steele. 2021. Vate: Runtime Adaptable Probabilistic Programming for Java. In *The 1st Workshop on Machine Learning and Systems (EuroMLSys'21), April 26, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3437984.3458835

1 Introduction

In this paper we introduce the design and implementation of Vate, a probabilistic programming language [4] for creating models for JVM-based applications. Inspired by earlier work on Augur [18], models are constructed to allow dynamic switching of execution environment. Currently this is restricted to single threaded and Fork-Join [10] JVM execution environments, but we plan to extend to GPUs, clusters, and runtime environments such as Callisto [8].

While deep learning models have advanced many tasks, they typically require large amounts of data, and it is hard to encode domain specific knowledge into them, or explain how they reach their results. For these reasons, domain experts may choose to construct explicit probabilistic models. Typically these models can be easily described, but implementing them requires the user to construct custom inference code. This is labour intensive and potentially error prone. The considerable effort acts as a deterrent to modifying the models for either general development or domain specific targeting. Probabilistic programming addresses this problem by providing a high-level means of describing the model, typically through a language [2, 5, 11, 15, 18] or through an API embedded in another language [1, 13, 17, 19]. Models described in probabilistic programs have the operations by which the values of the model are inferred provided by the compiler or API, saving the user from this error prone work, and reducing the burden of adapting models.

Vate compiles models to JVM classes that encapsulate the complexity of the different model operations, providing the user with a clean Object-Oriented interface. The language is single assignment, but designed to be familiar to Java programmers, making models more accessible to the developer community. Models constructed in Vate perform three classes of operation: *Value inference*, where the value of any variables that are not fixed can be inferred; *Probability inference*, where the probabilities of individual parameters and the whole model can be estimated; *Conventional execution*, which can be used to generate outputs from a trained model in cases such as linear regression or to produce synthetic data sets. The values of variables within the model can be fixed at compile time or at run time.

Value inference is performed primarily through Gibbs sampling [3] using conjugate priors when possible and reverting to marginalising out sampled variables for finite discrete distributions, and localised Metropolis–Hastings [9] inference for unbounded or continuous distributions. The results of this inference can be: all the sampled values modulo any requested burning in and thinning; the value when the model was in its most probable state (Maximum a Posteriori, MAP); or no values saved. These policies are set for the whole model and can be modified on a per variable basis.

EuroMLSys'21, April 26, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8298-4/21/04…$15.00
https://doi.org/10.1145/3437984.3458835

1.1 Probabilistic Models

At a high level probabilistic models consist of parameterised random variables, constant parameters supplied by the user, and relationships between the variables, parameters and data. Random variables encapsulate simpler probability distributions which can be easily sampled from producing values. The values of the random variables and the values sampled from them can be inferred in a variety of ways.

For example a random variable representing a Uniform distribution parameterised with the values 0 and 1 will when sampled provide values in the range $[0 \ldots 1]$ with uniform probability, and will not provide any values outside of this range. This sampled value can then be used as an input to another random variable. If we use it to parameterise a Bernoulli distribution, it could represent the bias of a coin. Repeated sampling from the Bernoulli distribution provides values that represent repeated flipping of the coin. If we then provide a set of observed results of flipping a real coin we can use *value inference* to determine the most like bias of the coin and the variance of this value. For this simple model a closed form solution exists rendering the probabilistic model unnecessary, but for more complex models such as the one used as an example in Section 2 this is not the case.

1.2 Contributions

The contributions of this paper are as follows:

- A language for writing probabilistic programming models that is familiar to Java developers.
- A compiler and runtime system for compiling models to Java class files, allowing more efficient execution that models built out of runtime API calls.
- An Object-Oriented design for compiled models that enforces the separation of the model implementation and the system that uses the model.
- An Object-Oriented design allowing the execution environment to be dynamically changed so models written once can take best advantage of the system they are used on.

The rest of this paper first describes the language and resulting models from a user's perspective before examining some of the implementation details. It briefly looks at performance before considering how Vate differs from other probabilistic programming frameworks.

2 Model Presentation

In this section we describe the construction and use of a model. As an example we use a simple Hidden Markov Model (HMM). This models flipping a number of potentially biased coins. After each flip, the next coin will be selected with probabilities drawn from a Dirichlet distribution. The hidden part of the model is which coin is being flipped. The code for the model is in Figure 1, and when combined with the code in Figure 3 takes a sequence of observed coin flips and determine the most likely coin biases and transition probabilities.

2.1 Model Code

Models are declared via the keyword `model` followed by the model name, any arguments, and a block containing the body. The arguments and the body are valid java syntax, with the exception of the syntactic sugar for initialising arrays of constants and setting loop bounds. In Java the body describes a sequence of operations to perform, in a probabilistic model it describes the relationship between variables so the value on the left of an assignment can effect values on the right. Variables named in the model will become fields in the compiled model allowing properties of these variables to be queried or set.

Variables within the model consist of base types such as `int` and `double`, arrays, and random variables representing instances of the distributions supported by Vate. Random variables are constructed either via a call to a constructor, for example `new Beta(1.0, 1.0)`, or via statically imported factory methods as in the example. The method `observe` is used to tie the values generated by the model to data provided by the user for inference. One or more values can be drawn from a random variable by the `sample` method, if sampled values are not observed, they can be changed during the inference steps.

To simplify the compilation and understandability of the model, all values are single assignment. This means that to sum the values in an array some additional functionality is required. To achieve this the method `reduce` is included. This takes as its arguments an array of values of type X, a unit value of type X, and an associative function taking two values of type X and returning one value of type X. These are used to build a binary tree with a lambda at each node and either an array element or a unit value at each leaf. This will be executed to provide the return value.

Models can be split into reusable methods, which are not required to exist in the same files, but can be formed into libraries, allowing families of models with a similar structure to be created without having to recreate the entire model. The syntax of methods is a subset of that used in Java. An example can be seen in Figure 2. Because the model will ultimately be represented by a Directed Acyclic Graph (DAG) before the inference code is constructed, recursive models are not currently supported.

2.2 Compiled Model

The compilation of the model generates a collection of classes in the specified package location, in this case `examples.hmm`, either in a directory or a jar file. These work in conjunction with the Vate runtime libraries. The runtime libraries are used to separate out common code, preventing duplication if multiple models are used, and improving maintainability

```
package examples.hmm;

model HMM(boolean[] measured, int nCoins) {
  //Construct a transition matrix m.
  double[] v = new double[nCoins] <~ 0.1;
  double[][] m = dirichlet(v).sample(nCoins);

 //Construct a weighting for the first coin to flip.
  double[] initialCoin = dirichlet(v).sample();

  //Construct biases for each coin
  double[] bias = beta(1.0, 1.0).sample(nCoins);

 //Allocate space to record which coin is flipped
  int nFlips = measured.length;
  int[] st = new int[nFlips];

  //Calculate the movements between coins
  st[0] = categorical(initialCoin).sample();
  for (int i: [1..nFlips) )
    st[i] = categorical(m[st[i - 1]]).sample();

  //Flip the coins
  boolean[] flips = new int[nFlips];
  for (int j: [0..nFlips) )
    flips[j] = bernoulli(bias[st[j]]).sample();

 //Assert that the flips match the measured data.
  flips.observe(measured);
}
```

Figure 1. Code for simple HMM model flipping a number of potentially biased coins. This uses the built in distributions *Bernoulli*, *Beta*, *Categorical*, and *Dirichlet*.

```
private double sum(double[] a) {
  return
    reduce (a, 0, (i, j) -> { return i + j; });
}
```

Figure 2. An example of a method using a reduction to sum the values of an array.

of the code because only the sections that are truly unique to the model appear in the generated code.

The compiled class that the user will interact with has the same name as the model, in this case `HMM`. Models have 3 constructors, an *empty* constructor allowing variables to be set later, a *full* constructor taking the values specified in the model signature, and a *shape* constructor with observed parameters absence for scalar values, and either an integer, or an $n-1$ dimensional integer array describing the shape of input arrays. This constructor is used when conventional

```
//Construct the model
int nCoins = 3;
boolean[] flips = loadObservedFlips(....);
HMM model = new HMM(flips, nCoins);

//Set the retention policies
model.setDefaultRetentionPolicy(
                             RetentionPolicy.MAP);
model.st.setRetentionPolicy(
                            RetentionPolicy.NONE);

//Run 2000 inference steps to infer model values
model.inferValues(2000);

//Gather the results.
double[] bias = model.bias.getMAP();
double[][] transitions = model.m.getMAP();
```

Figure 3. An example of inferring the parameters of the model in Figure 1 based on a series of coin flips.

execution of the model is required, because only the shape of observed arrays is required.

Named parameters in the model each have a field of the same name in the compiled model class. Each of these fields references an object that can be used to assign values to and query properties of the parameter.

An example of user interaction with the compiled example model is given in Figure 3.

2.2.1 Fixing and saving values.

Once the value of parameters in the model have been set by a user or inferred from training data, it is possible to fix them on a per value basis. Use cases for this include: Fitting a topic model, then fixing the topics and running inference against new documents to determine their topic; And running conventional execution on a trained model to generate a synthetic dataset. Inferred data can be saved to and reloaded later, allowing trained models to be easily distributed.

2.2.2 Documentation.

To make models easier to share JavaDoc comments can be attached to the whole model, individual variables, and methods. These are then propagated to the compiled model, providing JavaDoc that can be browsed, or used by IDEs. To allow model identification all compiled model classes have a method to return the original models source code.

3 Compilation

Having illustrated how models are written and used, this section provides an overview of how the models are constructed internally, the compilation process, and how we overcame some issues.

3.1 Model Structure

Compiling Vate models produces a number of classes that extend elements of the Vate runtime system. These classes are split into two categories: The classes that provide the object-oriented aspects of the model; And core classes implementing a common interface, providing the current state of the model, and the numeric methods to manipulate this state. Only one of these core classes is instantiated at any given time, with each targeting a different hardware/software backend. This separation of concerns allows the user interface and the numeric calculations to be developed separately. This is particularly important in allowing models to target different hardware or runtime systems by constructing a different core class while maintaining the stability of the user code.

3.2 Compilation Process

The compilation of a model takes the following steps:

1. Translate the model to Java API calls which are then compiled to intermediate Java class files.
2. Execute the intermediate code to construct a DAG representing the model code. This is a direct representation of the code, not a Bayesian Network.
3. Explore the DAG to determine relationships between variables.
4. Construct methods for inferring new values and calculating probabilities for each random variable.
5. Construct methods for combining these operations.
6. Apply optimisations.
7. Output constructed methods to target languages.
8. Compile this code to the final model implementation.

The first step of compiling a model is source-to-source translation to generate Java code for classes which contains static methods constructed from API calls. These methods are a root method representing the model, and a method representing each method defined in the model. The Java code is then compiled producing classes to be executed in the next step. If methods are included in the model the compiler will output the translated and compiled intermediate classes so they can be included in the compilation of later models by adding them to the class path for step 2.

This first step allows Java to type check the model, any type errors returned are mapped back to the locations where they occur in the model. There is also the potential to construct translators for other languages. The use of class files to store the models in their partially compiled state allows the Java package system, jar files, etc. to be used when handling the compiled methods.

When the root method is executed, the API calls construct a DAG consisting of task nodes and variable nodes representing the model. Task nodes represent operations within the model such as arithmetic, sampling, constructing random variables, array operations, and control constructs such as *for* loops. Variable nodes represent the data generated by each task, with each task producing a single variable. An example can be seen in Figure 4. This DAG is then traversed to identify producer-consumer relationships between random variables, input values, observed values, and internal values. To construct the complete DAG for a given model would require runtime information describing how many iterations of each *for* loop are performed, this could also make the DAG prohibitively large. To overcome this, "for loop" nodes in the DAG represent iterations, with each iteration outputting a different index value. In conjunction with the granularity of arrays it is possible for dependencies to exist between loops iterations. For example, in the HMM code, the value of `st` depends on earlier values of `st`. How this is handled is described in Section 3.5.

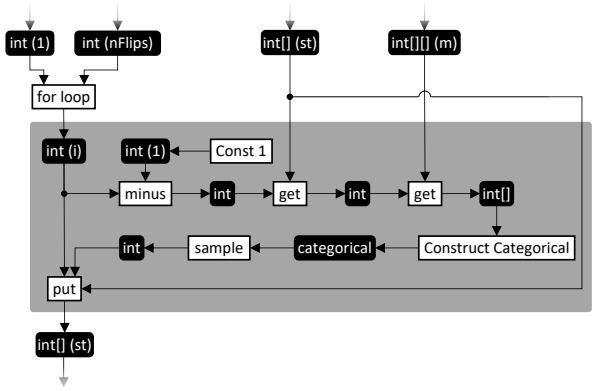

Figure 4. A representation of the DAG for the part of the model from Figure 1 that sets the values of `st` from 1 to `nFlips`. Tasks are white, and variables are black with their name or value if known in brackets. The tasks and variables in the grey area are in the scope of the *for* loop. The remaining tasks and variables are in the Global scope.

Once the DAG and the traces are constructed, trees representing code for conventional execution of parts of the model are generated from the DAG. For example, the construction of the input `v`. This is done via calls to the nodes in the DAG that will then recursively construct the required trees. Code for value inference via Gibbs Sampling and probability calculations requires the ability to invert the relationships between consuming and producing tasks. This is done via a combination of traces recording sequences of tasks, and inverse functions included in each task.

The data structures representing all the constructed methods and fields are collected in a single data structure, and an aggressive set of optimisations is applied to them. This allows the earlier steps to produce simple code that is easier to check for correctness. Many of the optimisations cannot be applied later by the target language compiler because domain specific information is lost.

3.2.1 Scopes. Each task and variable in the DAG has an associated scope describing which construct of the model,

(*for* loop, reduction, etc.) encompasses it. With the exception of the outermost scope (Global), each scope has a parent scope creating a tree structure of all the scopes in the model. This provides an easy way of determining the relationship between elements of the DAG, as demonstrated in Section 3.4. It also provides a framework into which fragments of generated code can be embedded, greatly simplifying the code generation methods within the DAG nodes.

3.3 Value inference techniques

As described in the introduction the primary inference technique is Gibbs Sampling using conjugate priors to calculate values generated by a source random variable and consumed by other random variables. To do this the values sampled from the consuming random variables are calculated by back tracking through the DAG, and the inputs to the supplied to the source random variable are calculated by conventional execution. We use heuristics to mark points in the DAG where results will be stored, in all other locations the values are recomputed on demand. Using these values, for each conjugate pair there is a formula which allows the sampling of a new value.

When it is not possible to use conjugate priors Vate reverts to marginalising out sampled variables for finite discrete distributions, and localised Metropolis–Hastings inference for unbounded or continuous distributions. The implementation of these two techniques has a similar structure. For marginalisation each possible value of the sampled value is set in turn, the state of the model is updated, and the probability of the model is calculated. This probability calculation is constrained to the source and consuming random variables. These probabilities are then used to parameterize a categorical random variable from which the new value is drawn. For Metropilis-Hastings the probability of the current value is calculated, a difference is drawn from a normal distribution, this is added to the current value, and the model is updated. The new probability is calculated. The ratio of the two probabilities is then compared with a value drawn from a random variable in the range $[0 \ldots 1]$ to determine if the new value should be kept, or the model should be reverted.

3.4 Working with Arrays

We now consider some of the issues related to arrays.

3.4.1 Assignment. While all variables are single assignment, in the case of arrays, assignment is at the granularity of individual elements within the array, not the whole array. This means that the value of an array can change. To encode these changes into the DAG, a *put* task takes as inputs the array, the index, and the value to assign, and outputs a new variable representing the new value of the array. Because iterations are included into the DAG but not executed, the number of *put* tasks, and therein variable nodes representing new values of the array is equal to the number of *put* operations in the model, not the number of *put* operations that would occur if the model was executed on a given data set. For example, in the code in Figure 1, there would be 2 *put* tasks in the DAG for the array `st`: One to assign the value of element 0; And one for all the assignments that appear in the *for* loop. In the case of multi-dimensional arrays, a *put* to an inner array will trigger an implicit *put* to the outer arrays to ensure that operations on the array are a total order. Because each variable stores the task that constructed it, this creates a linked list of array values which can be explored to generate a set of possible other traces that could exist beyond the ones that can be read from the DAG. This is used primarily in the detection of dependencies between iterations; For example, the construction of values of `st` that depend on earlier values of `st`. These dependencies are considered in Section 3.5.

3.4.2 Multiple Traces. Values can be placed into arrays, read back out of the arrays, transformed, placed into different arrays, etc. Because these reading and writing operations apply to individual elements it is possible to create a number of different paths between producers and consumers. All paths must be considered when constructing code that reverses a function to ensure that the correct transformations are applied to the correct values. An example can be seen in Figure 5, where two possible paths exist between the variable `a`, and the variable `c`.

To achieve this, the code for performing the inverse function for the tasks in each trace is placed inside a set of *for* loops that represent all the *for* loops that each trace has been through. Within all these loops guards are placed to ensure that the code is executed only if all the indexes for *puts* and *gets* to the array match. An example of this can be seen in Figure 6. This is clearly not very efficient with large numbers of loop iterations being executed without the inner loop body being executed. To address this, an aggressive set of optimisations are applied. These take advantage of knowledge of the bounds and step size of loops to: Move guard conditions to the earliest point they can be applied to allow early pruning; Rearrange the conditions to allow the loop indexes to be replaced with functions based on other variables thereby removing the loop. The effect of this can also be seen in Figure 6. In this example we have left out conditions to prevent double counting for brevity.

3.5 Iterations

The presence of arrays and loops can lead to dependencies between iterations. For example the Categorical random variable that generates the later values of `st` is also a consumer of the output from earlier iterations. This creates two issues.

First, random variables that consume the output of a given random variable can be missed because exploring the DAG may not identify all potential traces. In the example, this

```
double[] a = new double[nSamples];
for (int i: [0..nSamples) )
  a[i] = beta(1,1).sample();

double[] b = new double[nSamples];
for (int j: [0..nSamples/2) )
  b[j] = 1 - a[j];
for (int k: [noSample/2..nSamples) )
  b[k] = a[k];

double[] c = new double[nSamples];
for (int m: [0..nSamples) )
  c[m] = b[nSamples - m];
```

Figure 5. Code with two paths between arrays a and c.

```
for (int i = 0; i < nSamples; i++) {
  for (int j = 0; j < nSamples/2; j++)
    for (int m = 0; m < nSamples; m++)
      if (i == j && j == nSamples - m)
        f(1 - c[m], i);

  for (int k = nSamples/2; k < nSamples; k++)
    for (int m = 0; m < nSamples; m++)
      if (i == k && k == nSamples - m)
        f(c[m], i);
}
```

a) Unoptimised code.

```
for (int i = 0; i < nSamples; i++) {
  if (i < nSamples/2)
    f(1 - c[nSamples - i], i);

  if (i >= nSamples/2)
    f(c[nSamples - i], i);
}
```

b) Code after optimisation.

Figure 6. Unoptimized and optimised code to use the method f to consume transformed values associated with each sample variable i. Further optimisation is possible and is intended future work.

happens between the Categorical and itself. Traces are constructed by recursively exploring backwards through the DAG starting at observed values and random variables. If a point of interest such as another random variable is reached, the trace between these two points is saved. To overcome the issue of missing traces via arrays, when a *get* task is encountered during the exploration, in addition to the *put* task that constructed that array being explored, any subsequent *put* tasks on this array are also considered to determine if the value assigned by any of them could be output by the *get* task. If it could be, these tasks are included in the exploration. This takes advantage of the ordered list of *put* tasks for each array described in Section 3.4.1 and the scopes described in Section 3.2.1. A later *put* task can be part of a trace if it is inside the same iteration as the *get* task. This can be tested for by simply taking the outer most iterative scope of the *get* task and the *put* task. If the scopes are the same then it is possible that there is a dependency, so the trace should explore from this point too. The *put* in Figure 4 would be added to traces from the categorical by this technique. As it is not possible to leave and re-enter a scope, if a *put* task fails this test, all later *put* tasks will also fail the test, so no further *put* tasks need to be considered.

Second, when there are no dependencies *for* loops can be calculated in parallel, but with dependencies the execution of the iterations must be restricted. Detecting these dependencies takes two steps. First, the constructed traces are explored to determine if there are any cycles. This is quick because cycles can be constructed only by traces that appear in the same outermost iterating scope, and only the start and end points of each trace need to be considered when using them to construct a cycle. By construction cycles have only a single task that links to a task appearing later in the graph. While more complex cycles can exist, they are all constructed by concatenating these simpler cycles.

Having determined there is a cycle, the next step is to determine where the dependency that creates the cycle occurs. This is achieved by sorting the non-implicit *put* tasks and the *get* tasks in the cycle into the order they appear in the DAG. Because the cycle contains only a single task linking to a later task there is a total ordering of these tasks. The list is then traversed. For each *get* encountered, the array, the dimension being indexed, and the index value are stored. If a *put* task on the same array and array dimension is subsequently encountered then this is assumed to be a dependency, and the loops are examined to determine which loop the index depends on. These loops are then marked to be executed serially and the unmarked loops can be safely executed in parallel. This mechanism can be overzealous, and further examination of the dependency may allow for some additional parallel execution.

4 Performance

A full performance analysis is beyond the scope of this paper, but as a point of comparison we compare inference of the example HMM model with a PyMC3 [13] version.

Model & input length	Iterations				
	1000	2000	4000	8000	16000
PyMC3, 1k	201.8s	388.4s	769.4s	1523s	3021s
Vate, 1k	0.174s	0.367s	0.760s	1.450s	2.809s
Vate, 10k	1.764s	3.499s	7.500s	14.34s	25.77s

All models are single threaded. In addition to being in excess of 1000x faster, the Vate results were also more probable.

While not suitable for models with local optima such as HMM we also ran the PyMC3 gradient based `findMAP` method on the two datasets. This took 13.3s and 13.9s to execute and returned values that were significantly less probable than those reached by Vate in 1000 iterations, and Vate was 76x and 7.8x times faster respectively. The difference in speed of convergence is due to different sampling techniques, with the Vate techniques converging in fewer iterations, and each iteration being quicker to execute. We attribute the difference in execution speed to a combination of interpreted vs compiled code, more targeted code, and the optimisations to Vate which can produce 40x speedups vs the unoptimized code. It terms of code complexity while this is hard to quantify, the PyMC3 model required 49 lines of code including the extending of two classes describing random variables, and implementing a filter to find the most probable result. The Vate implementation consisted of the 16 lines of code in this paper. A fuller evaluation is future work.

To examine parallel performance a more complex HMM model drawn from a production system is used. Here states are web pages, and events (Flips) are actions performed on those pages. When training the model will observe a large number of these traces to infer the models parameters. Figure 7 shows the speedup on a 6 core Intel i7-9750H, Coffee Lake, system is 4.5x with 6 threads rising to 5.5x with hyperthreading. We did not compare this model to an implementation in PyMC3 as the streams of events can be different lengths, requiring different numbers of random variables for each stream. As PyMC3 names each random variable implementing this would require code to generate and track a unique name for each random variable.

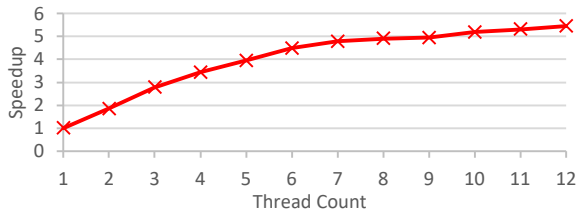

Figure 7. Speedup with increasing thread count for a more complex HMM model.

5 Related Work

Started in 1989 the BUGS [15] project is one of the first probabilistic programming languages. BUGS interprets declarative models written in a language that is syntactically similar to imperative languages but requires the user to separate out values sampled from distributions, and deterministic relationships between values. JAGS [12] was developed to make BUGS platform independent, and Multi-BUGS [7] is a Windows based implementation that performs parallel execution using techniques similar to those in Vate.

Later probabilistic programming systems can be roughly split into two forms: Languages that move away from imperative coding styles towards the statistical underpinnings of the models; And APIs that are called to construct models within an existing programming language such as Python or C#. Vate, like Augur [18], picks an alternative position where the language is familiar to developers, but the model is compiled providing separation of concerns, and performance benefits.

Augur uses Scala macros to compile models in Scala code to the JVM and to GPUs. The use of Scala macros ties Augur to specific versions of the Scala compiler, Vate's clean separation works with any Java 8+ compiler. In addition to the macros Augur also required further compilation of the byte code. These two steps with the Scala compiler in between made Augur very opaque and hard to debug, modify, and maintain. Unlike Vate's trace based approach Augur generates inference techniques via equation rewriting. Relative to Vate this restricts the complexity of the variable relationships and therein models that Augur is able to handle.

Examples of statistically inspired languages are the family of languages based on Lisp including Church [5] and Anglican [16], and languages such as Birch [11] and Stan [2]. Unlike the interpreted code of BUGS, Stan generates compiled C++, and uses gradient methods for value inference. This improves performance at the cost of excluding models with discrete distributions. Stan models are split into a large number of separate blocks based on their function. Some improvement of this has been made with SlicStan [6].

Examples of API based modelling include: PyMC3 [13], Infer.net [19], Edward [17], Pyro [1], and Bean Machine [14]. The use of APIs allows models to be constructed in languages that are familiar to developers, but at the cost of encapsulation. Models are no longer separated from the code that calls them making them more complex to read and modify, and removing guarantees that the model is not manipulated by control flow it is unaware of.

6 Conclusions

We introduced Vate, a probabilistic programming language for describing and compiling JVM based models that can be dynamically targeted to different runtime environments. These models are object-oriented, and perform a wide range of operations. Vate differs from existing work in the field by using a language familiar to developers to construct encapsulated models, separating the concerns of model use from model description.

Acknowledgments

We would like to thank Gavin Bierman, Mark Moir, and the reviewers for their constructive comments on this paper.

References

[1] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *J. Mach. Learn. Res.* 20, 1 (Jan. 2019), 973–978.

[2] 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).

[3] Stuart Geman and Donald Geman. 1984. Stochastic Relaxation, Gibbs Distributions, and the Bayesian Restoration of Images. *IEEE Trans. Pattern Anal. Mach. Intell.* 6, 6 (Nov. 1984), 721–741. https://doi.org/10.1109/TPAMI.1984.4767596

[4] Noah D. Goodman. 2013. The Principles and Practice of Probabilistic Programming. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 399–402. https://doi.org/10.1145/2429069.2429117

[5] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. 2008. Church: A Language for Generative Models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence* (Helsinki, Finland) *(UAI'08)*. AUAI Press, Arlington, Virginia, USA, 220–229.

[6] Maria I. Gorinova, Andrew D. Gordon, and Charles Sutton. 2019. Probabilistic Programming with Densities in SlicStan: Efficient, Flexible, and Deterministic. *Proc. ACM Program. Lang.* 3, POPL, Article 35 (Jan. 2019), 30 pages. https://doi.org/10.1145/3290348

[7] Robert J. B. Goudie, Rebecca M. Turner, Daniela De Angelis, and Andrew Thomas. 2020. MultiBUGS: A Parallel Implementation of the BUGS Modeling Framework for Faster Bayesian Inference. *Journal of Statistical Software* 95, 7 (2020). https://doi.org/10.18637/jss.v095.i07

[8] Tim Harris and Stefan Kaestle. 2015. Callisto-RTS: Fine-Grain Parallel Loops. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 45–56. https://www.usenix.org/conference/atc15/technical-session/presentation/harris

[9] W. K. Hastings. 1970. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika* 57, 1 (1970), 97–109. http://www.jstor.org/stable/2334940

[10] Doug Lea. 2000. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande* (San Francisco, California, USA) *(JAVA '00)*. Association for Computing Machinery, New York, NY, USA, 36–43. https://doi.org/10.1145/337449.337465

[11] Lawrence M. Murray and Thomas B. Schön. 2020. Automated learning with a probabilistic programming language: Birch. arXiv:1810.01539 [stat.ML]

[12] Martyn Plummer. 2003. JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling.

[13] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* 2 (apr 2016), e55. https://doi.org/10.7717/peerj-cs.55

[14] Nazanin Tehrani, Nimar S. Arora, Yucen Lily Li, Kinjal Divesh Shah, David Noursi, Michael Tingley, Narjes Torabi, Sepehr Masouleh, Eric Lippert, and Erik Meijer. 2020. Bean Machine: A Declarative Probabilistic Programming Language For Efficient Programmable Inference. In *Proceedings of the 10th International Conference on Probabilistic Graphical Models (Proceedings of Machine Learning Research, Vol. 138)*, Manfred Jaeger and Thomas Dyhre Nielsen (Eds.). PMLR, 485–496. http://proceedings.mlr.press/v138/tehrani20a.html

[15] Andrew Thomas, David J Spiegelhalter, and WR Gilks. 1992. BUGS: A program to perform Bayesian inference using Gibbs sampling. *Bayesian statistics* 4 (1992), 837–842.

[16] David Tolpin, Jan Willem van de Meent, Hongseok Yang, and Frank Wood. 2016. Design and Implementation of Probabilistic Programming Language Anglican. *arXiv preprint arXiv:1608.05263* (2016).

[17] Dustin Tran, Alp Kucukelbir, Adji B. Dieng, Maja Rudolph, Dawen Liang, and David M. Blei. 2016. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787* (2016).

[18] Jean-Baptiste Tristan, Daniel Huang, Joseph Tassarotti, Adam C Pocock, Stephen Green, and Guy L Steele. 2014. Augur: Data-Parallel Probabilistic Modeling. In *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger (Eds.), Vol. 27. Curran Associates, Inc., 2600–2608. https://proceedings.neurips.cc/paper/2014/file/cf9a242b70f45317ffd281241fa66502-Paper.pdf

[19] John Winn and Tom Minka. 2009. Probabilistic Programming with Infer.NET. https://www.microsoft.com/en-us/research/publication/probabilistic-programming-infer-net/