A Live, Multiple-Representation Probabilistic Programming Environment for Novices

Maria I. Gorinova[1], **Advait Sarkar**[1], **Alan F. Blackwell**[1], **Don Syme**[2]

[1]Computer Laboratory, University of Cambridge
15 JJ Thomson Avenue, Cambridge, United Kingdom
{mig30,advait.sarkar,alan.blackwell}@cl.cam.ac.uk

[2]Microsoft Research Cambridge
Cambridge, UK
dsyme@microsoft.com

ABSTRACT

We present a live, multiple-representation novice environment for probabilistic programming based on the Infer.NET language. When compared to a text-only editor in a controlled experiment on 16 participants, our system showed a significant reduction in keystrokes during introductory probabilistic programming exercises, and subsequently, a significant improvement in program description and debugging tasks as measured by task time, keystrokes and deletions.

Author Keywords

Probabilistic programming; visual languages; multiple representation; computational thinking; development environment

ACM Classification Keywords

H.5.m Information Interfaces and Presentation (e.g. HCI): Miscellaneous; D.2.6 Software Engineering: Programming Environments

INTRODUCTION

A new generation of probabilistic programming languages, such as Infer.NET [12], manipulates variables which are associated not with single values, but rather entire probability distributions. This is becoming a popular approach for data analytics, as many statistical techniques are more naturally expressed as probabilistic programs. In this paper, we report the design and evaluation of a system that makes these capabilities more accessible to students and end-user developers.

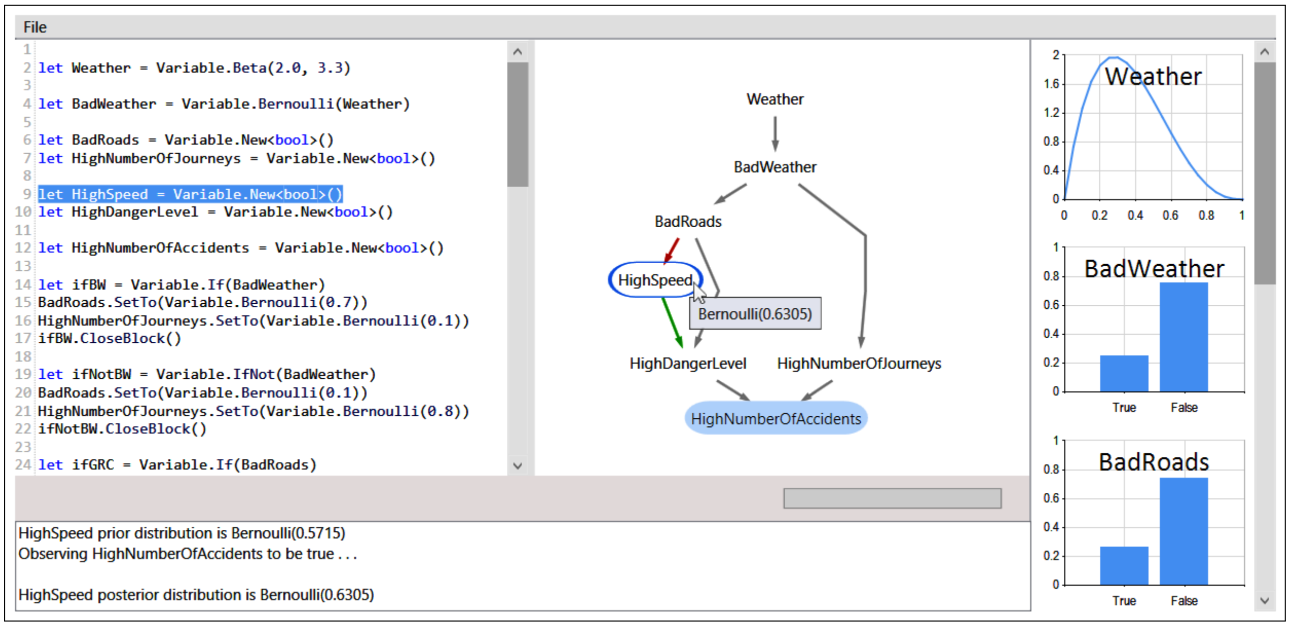
The conventional understanding that a variable holds only one value has been suggested as the single most important concept determining whether someone is able to learn programming [6]. The conceptual model of probabilistic programming, where variables embody distributions, involves a substantial shift from this convention. For example, consider a random variable *B* whose distribution depends upon the value of a random variable *A*. When *B* takes on a particular value, our beliefs regarding the distribution of *A* are updated. Probabilistic programming is a challenge not only because the notion of a variable and its value is fundamentally different, but also because changes are not always explicit; changing the value of a variable affects those *upon which it depends*, in addition to affecting those which depend upon it (as in conventional languages).

Our concern was therefore to design a practical tool that successfully communicates this different conceptual basis, while also allowing users to successfully manipulate probabilistic programs. Tutorials on graphical models and Bayesian networks [13] typically explain the structure of the model with a node-and-link diagram, where nodes represent random variables, and links show conditional dependencies. When Bayes' theorem is introduced through visualisations, students learn faster and report higher temporal stability [14]. Despite the term "graphical," many probabilistic programming languages describe models in purely textual terms, rather than in diagrammatic form. This is no doubt due in part to the relative ease of building text parsers and compilers, but may also have resulted from the relative lack of success for tools such as UML, which become relatively inflexible through separate phases of diagrammatic specification and code generation.

Programming environments exist that maintain multiple representations on screen simultaneously, with modification of one resulting in simultaneous modification of the others [15]. This approach has also been characterised in terms of "liveness" [17], and has recently been popularised in projects such as Light Table [3]. Such research provides evidence for the educational and practical potential of this strategy. In the remainder of this paper we describe the first application of this strategy to the design and evaluation of an integrated development environment for live probabilistic programming.

This work makes the following contributions:

- We motivate probabilistic programming as an emerging domain which can benefit from established diagrammatic convention using live, multiple-representation editors.
- We present the design and implementation for a multiple-representation environment (MRE) for probabilistic programming aimed towards novices.
- We report a real-world comparative study of the effects of an MRE on the effort of novice programmers during initial practice and simple application tasks.



```

let Weather = Variable.Beta(2.0, 3.3)

let BadWeather = Variable.Bernoulli(Weather)

let BadRoads = Variable.New<bool>()
let HighNumberOfJourneys = Variable.New<bool>()

let HighSpeed = Variable.New<bool>()
let HighDangerLevel = Variable.New<bool>()

let HighNumberOfAccidents = Variable.New<bool>()

let ifBW = Variable.If(BadWeather)
BadRoads.SetTo(Variable.Bernoulli(0.7))
HighNumberOfJourneys.SetTo(Variable.Bernoulli(0.1))
ifBW.CloseBlock()

let ifNotBW = Variable.IfNot(BadWeather)
BadRoads.SetTo(Variable.Bernoulli(0.1))
HighNumberOfJourneys.SetTo(Variable.Bernoulli(0.8))
ifNotBW.CloseBlock()

let ifGRC = Variable.If(BadRoads)
```

```
HighSpeed prior distribution is Bernoulli(0.5715)
Observing HighNumberOfAccidents to be true . . .

HighSpeed posterior distribution is Bernoulli(0.6305)
```

Figure 1: Our multiple-representation editor. Left to right: source code, Bayesian network diagram, posterior distribution charts.

PREVIOUS WORK

Previous work has largely focused on visualising graphical models in a single-representation system. Bayesia [1] and Netica [4] are both commercial tools that allow users to create Bayesian networks interactively. However, these single-representation systems do not expose the underlying code and have restricted functionality due to their commercial focus. ViSMod [18] is a similar system aimed towards students. VIBES [5] and WinBUGS [11] both allow the user to describe a probabilistic model and perform inference on it. The model in both cases is described by either XML script/BUGS language or pictorially, via a graphical interface. However, neither implements liveness or multiple representations. Tabular [9] is a live probabilistic language which allows programs to be written as annotated relational schemas within spreadsheets. While achieving liveness, its interface is purely spreadsheet-based and does not provide visualisations.

Bayesian networks are useful visualisations of dependence between variables, but are incomplete representations of the problem, as they don't represent the nature of dependencies or priors. While the design for a complete visual language for probabilistic programming is an intriguing avenue, our aim was to help students better understand how the fully-expressive code representation corresponds to concepts of probability theory as learnt in the classroom. Thus, our solution is rooted in multiple representation theory [15], through which existing novice knowledge scaffolds new concepts.

OUR MULTIPLE-REPRESENTATION ENVIRONMENT

Visual representation

Our interface (Figure 1) implements 3 panes: the first is a standard source code editor. The second is a node-link diagram representing the graphical model according to standard convention. The third is a series of charts showing the distributions of each random variable in the program. The node-link diagram is laid out initially using the Sugiyama algorithm [16]. Nodes can be freely moved after the visualisation is rendered. Nodes corresponding to observed variables are coloured blue. Array variables are shown as rectangles, as a compact alternative to the standard plate notation [7]. Tooltips on each node show the exact parameterised distribution. Hovering over a node highlights the corresponding variable definition in the source pane (Figure 1). Future work may introduce bidirectional navigation [8]. Posterior distributions for inferred variables are visualised in the rightmost pane. Note that only the code is editable; the graph/distributions are not. This is potential future work and our graph-to-code navigation is a step in this direction.

A progress bar appears during compilation and rendering. The visualisation is dimmed through a translucent overlay during recompilation to indicate staleness. Finally, a console at the bottom displays compilation/type-checking error messages, as well as the output of print statements.

Liveness

We use the F# Compiler Services package [2] to parse the user's program and walk the typed abstract syntax tree. Our implementation supports visualisation of top-level declarations in single-sourcefile programs. We inject `infer` statements to obtain the distribution of each random variable, which causes Infer.NET to serialise a *factor graph* [10] describing the model, from which we extract the corresponding Bayesian network. Concurrently-threaded edit-triggered recompilation provides 'level 3 liveness' [17]. Each edit spawns a separate type-checking and compilation thread on an in-memory copy of the program. Subsequent edits terminate old threads. This enables concurrent background compilation while the user continues editing.

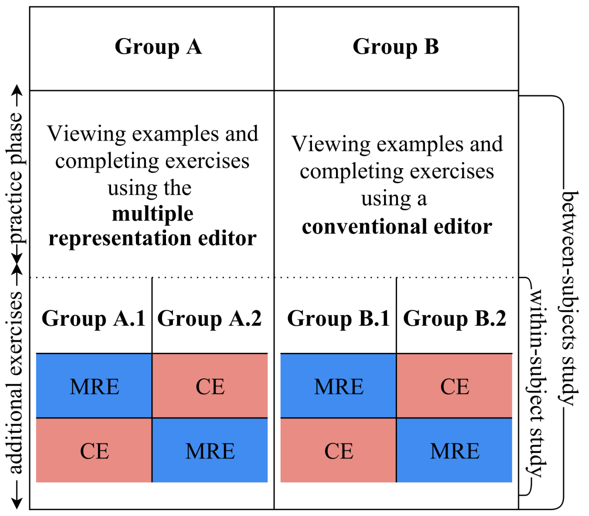

Group A
Group B
Viewing examples and completing exercises using the multiple representation editor
Viewing examples and completing exercises using a conventional editor
Group A.1
Group A.2
Group B.1
Group B.2
MRE
CE
CE
MRE
MRE
CE
CE
MRE
practice phase
additional exercises
between-subjects study
within-subject study

Figure 2: Overall design of the experiment.

Our system scales to complex programs provided Infer.NET compiles fast enough for liveness. The graph can become unwieldy for vast numbers of variables and interdependencies (e.g., 1000s). However, in the context of novice programming, problems are typically much smaller and meant to be illustrative, and our system handles these well.

EXPERIMENTAL EVALUATION

We designed an experiment to evaluate the following: **Practice**: does the use of multiple representations facilitate initial practice exercises in probabilistic programming, as measured by task time, keystrokes and deletions relative to a traditional editor? **Application**: does the use of multiple representations reduce effort in simple end-user development (EUD) tasks, using the same measures?

Experiment design and procedure

We recruited 16 University of Cambridge undergraduates studying computer science or mathematics. All students had basic prior knowledge of probability, but had never encountered probabilistic programming.

To present the experimental tasks in a familiar manner, we designed a workbook in the same format as programming assessments from the Cambridge undergraduate computer science course. The workbook consisted of two parts:

Part 1 introduced the participant to F#, probabilistic programming and Infer.NET, by interleaving text explanations, four code examples and four practice exercises.

Part 2 consisted of 8 exercises in 4 pairs designed to be of equal difficulty. The exercises were equally divided into two types: *Debugging* exercises, where participants were given a program and a short English specification and asked to indicate the bugs which caused the program to deviate from the specification; and *Description* exercises, where participants were given a program and asked to describe its purpose. Equal difficulty was based on number of random variables in the model, equal numbers of discrete vs. continuous, numbers

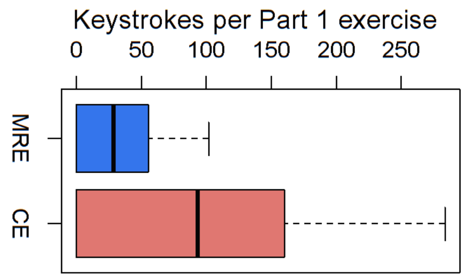


Figure 3: Lower keystrokes in phase 1 with MRE.

of dependencies, and numbers of modification steps needed to complete the debugging exercises.

We did not include authoring tasks, as these are harder to introduce in a setting where the learning time of participants is limited. Use of debugging tasks allowed us to use considerably more complex programs than participants could be expected to author from scratch, and also reflects contextual evidence that novice end-user developers generally modify programs created by others.

For phase 1, participants were divided into 2 groups of size 8. Both groups completed Part 1 of the workbook, the first group using the multiple representation environment (MRE) and the second group using a more conventional environment (CE), created by hiding the two visualisation panes from our prototype, leaving only the source code pane. Both MRE and CE included the console which shows parse/type errors and acts as standard output. Participants who used the MRE in phase 1 were not specifically instructed about the MRE system. They explored the same tutorial examples but with the MRE rather than the CE.

For phase 2, each group was then further split into four subgroups of size 2, using a 2×2 counterbalanced design to facilitate a within-subject study on programming effort. Each participant completed four exercises using the MRE and the other 4 (their equal difficulty counterparts) with the conventional editor. The order of condition (with-visualisations or without), and the order of exercises was counterbalanced.

For each participant, we recorded the number of keystrokes, backspaces and time needed to complete each exercise. Participants were asked to vocalise their answers and were given an indication of whether they were correct or incorrect. We did not record the number of correct answers; instead, participants were gently prompted to continue if they had made a mistake until a correct solution was reached. The study took 92 minutes on average.

RESULTS

Practice phase (between-subjects)

A one-way MANOVA treating the three measurements (task time, keystrokes, and deletions) as different dependent variables indicated that one or more of these variables is dependent on the type of practice ($p = 2.88 \cdot 10^{-3}$).

	Comment	Context
P04	*I completely missed [that] dependency.*	Description exercise, CE.
P05	*Found that quite fun.*	Finishing with the MRE.
P05	*Argh, no graph, nooo!*	Change from MRE to CE.
P06	*'Correct the errors' – how am I supposed to – the arrows were so useful for errors!*	Debugging exercise, CE.
P06	*I would really like the thing with the arrows.*	Debugging exercise, CE.
P06	*Looking at the code is horrific... but looking at the graph is not that bad.*	Description exercise, MRE.
P11	*You see straight from the IDE...*	Description exercise, MRE.
P16	*I'm not going to look at the code anymore!*	Change from CE to MRE.

Table 1: Participant comments.

	V	p	MRE median	CE median
Time	611	0.004162	288s	359s
Keystrokes	302	0.002044	15.5	79
Backspaces	367.5	0.02397	3.5	9

Table 2: Results of paired Wilcoxon tests on Part 2 exercises.

A follow-up ANOVA on each dependent variable separately (applying a Bonferroni correction to yield $\alpha = 0.0167$) revealed no significant effect on either task time or deletions. However, practice with the MRE required significantly fewer keystrokes ($F = 23, p = 2.85 \cdot 10^{-4}$). The effect size was a difference of medians of 64.5 keystrokes, down to 28.5 from 93 with the CE (Figure 3).

Application phase (within-subjects)

To conduct a within-subjects comparison between the MRE and CE in phase 2, we use the paired Wilcoxon signed-rank test as the data were not normally distributed (Shapiro-Wilk test, $p < 0.05$ in all cases).

Table 2 summarises the Wilcoxon tests carried out on phase 2 tasks. We observe that the MRE provided significant advantages in completing probabilistic programming exercises over the conventional editor. With the MRE, participants gained a median task time improvement of 71s, a median reduction of 63.5 keystrokes, and a median reduction of 5.5 deletions.

DISCUSSION

We did not observe a significant difference in practice time when using a multiple representation editor, likely because a larger proportion of time was spent reading and internalising the exercises than coding. During the practice phase, participants with the MRE used significantly fewer keystrokes, suggesting that providing multiple representations improved their ability to understand and apply Infer.NET syntax.

Moreover, during the second (application) phase, significant improvement in time, keystrokes and deletions was observed when using the MRE. Comments from the participants show qualitatively that users greatly preferred the visualisations and found them beneficial (Table 1).

The advantages of multiple representations were particularly evident during the description exercises. Four participants in the MRE-practice condition stated while using the conventional editor that they were trying to mentally visualise the Bayesian network to better understand the dependencies between variables. Moreover, when asked to describe a model aloud using the conventional editor, participants were more likely to start reading the code line by line (e.g. "when `Cloudy` is `true`, `Rain` is `Bernoulli(0.7)`, otherwise it is `Bernoulli(0.1)`."). In comparison, participants using the MRE spoke in terms of variable dependencies and uncovered connections otherwise not obvious from the source code alone (e.g "It's more likely to rain when it is Cloudy, and consequently it is more likely to be wet when it is Cloudy").

These results demonstrate that use of multiple simultaneous representations in a live environment allows users to engage with the novel programming model more efficiently and effectively. In this study, we focused on relatively straightforward practice and application tasks, typical of novice or end-user development engagement with a new programming environment. A natural next step would be to extend this investigation from interpretation and debugging tasks, to more exploratory development tasks (as carried out by professional data analysts) or educational exercises (as used when teaching machine learning with probabilistic languages).

CONCLUSION

We have motivated the use of multiple simultaneous representations in probabilistic programming tools. Our core design aims were *multiple representation*, i.e. showing multiple simultaneous views of the program; and *liveness*, where the program is not a static entity, executed only after compiling, but rather an interactive object.

We have presented a system which leverages existing diagrammatic convention to implement a live, multiple representation editor for probabilistic programs, incorporating a conventional text editor for source code, a Bayesian network node-link diagram, and chart visualisations to display the posterior distributions of random variables.

Finally, in a study of 16 users, we have shown that the use of multiple representations has significant qualitative and quantitative benefits, reducing the time, keystrokes, and deletions needed to carry out typical novice end-user tasks such as initial practice, program comprehension and simple debugging.

REFERENCES

1. 2015. Bayesia. (2015). http://www.bayesia.com/ Accessed: February 9, 2016.
2. 2015. F# Compiler Services. http://fsharp.github.io/FSharp.Compiler.Service/. (2015). Accessed: February 9, 2016.
3. 2015. Light Table. http://lighttable.com/. (2015). Accessed: February 9, 2016.
4. 2015. Netica. (2015). https://www.norsys.com/netica.html Accessed: February 9, 2016.
5. Christopher M Bishop, David Spiegelhalter, and John Winn. 2002. VIBES: A variational inference engine for Bayesian networks. In *Advances in neural information processing systems*. 777–784.
6. Richard Bornat, Saeed Dehnadi, and others. 2008. Mental models, consistency and programming aptitude. In *Proceedings of the tenth conference on Australasian computing education-Volume 78*. Australian Computer Society, Inc., 53–61.
7. Wray L. Buntine. 1994. Operations for learning with graphical models. *JAIR* 2 (1994), 159–225.
8. Sebastian Burckhardt, Manuel Fahndrich, Peli de Halleux, Jun Kato, Sean McDirmid, Michal Moskal, and Nikolai Tillmann. 2013. It's Alive! Continuous Feedback in UI Programming. In *PLDI*. ACM SIGPLAN. http://research.microsoft.com/apps/pubs/default.aspx?id=189242
9. Andrew D Gordon, Thore Graepel, Nicolas Rolland, Claudio Russo, Johannes Borgstrom, and John Guiver. 2013. *Tabular: A Schema-Driven Probabilistic Programming Language*. Technical Report MSR-TR-2013-118. http://research.microsoft.com/apps/pubs/default.aspx?id=204661
10. F.R. Kschischang, B.J. Frey, and H.-A. Loeliger. 2001. Factor graphs and the sum-product algorithm. *IEEE Transactions on Information Theory* 47, 2 (2001), 498–519. DOI: http://dx.doi.org/10.1109/18.910572
11. David J Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. 2000. WinBUGS-a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and computing* 10, 4 (2000), 325–337.
12. T Minka, J M Winn, J P Guiver, S Webster, Y Zaykov, B Yangel, A Spengler, and J Bronskill. 2014. Infer.NET 2.6. (2014).
13. Kevin Murphy. 1998. A brief introduction to graphical models and Bayesian networks. (1998).
14. P Sedlmeier and G Gigerenzer. 2001. Teaching Bayesian reasoning in less than two hours. *Journal of experimental psychology. General* 130, 3 (Sept. 2001), 380–400. http://www.ncbi.nlm.nih.gov/pubmed/11561916
15. Alistair Stead and Alan F. Blackwell. Learning Syntax as Notational Expertise when using DrawBridge. In *Psychology of Programming Interest Group Annual Conference 2014*. 41.
16. Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. 1981. Methods for visual understanding of hierarchical system structures. *Systems, Man and Cybernetics, IEEE Transactions on* 11, 2 (1981), 109–125.
17. Steven L Tanimoto. 1990. VIVA: A visual language for image processing. *Journal of Visual Languages & Computing* 1, 2 (1990), 127–139.
18. Uan-Diego Zapata-Rivera and Jim E. Greer. 2004. Interacting with Inspectable Bayesian Student Models. *International Journal of Artificial Intelligence in Education* Volume 14, Number 2/2004 - IOS Press (2004). http://iospress.metapress.com/content/rj0wwc454vvll8xn/