ProB and Jupyter for Logic, Set Theory, Theoretical Computer Science and Formal Methods

David Geleßus and Michael Leuschel(✉)

Institut für Informatik, Universität Düsseldorf, Universitätsstr. 1, 40225 Düsseldorf, Germany
{dagel101,michael.leuschel}@hhu.de

Abstract. We present a tool for using the B language in computational notebooks, based on the Jupyter Notebook interface and the ProB tool. Applications of B notebooks include executable documentation of formal models, interactive manuals, validation reports but also teaching of formal methods, logic, set theory and theoretical computer science. In addition to B and Event-B, the tool supports Z, TLA$^+$ and Alloy.

1 Introduction and Motivation

The computational notebook concept has recently become popular in teaching and research, as it allows mixing executable code with rich text descriptions and graphical visualizations. We present a tool which enables B and other formal methods to be used in computational notebooks. Such notebooks have many applications, from teaching formal methods to documenting formal models or generating executable reference documents. Given the foundations of B in set theory and logic, and given that the Unicode syntax of B is identical to or very close to standard mathematical notation, our tool can also be used to produce notebooks for teaching mathematical foundations in general or theoretical computer science in particular. Given that our tool is based on the ProB tool, the notebooks also provide convenient access to its constraint solver.

2 Jupyter Kernel for B

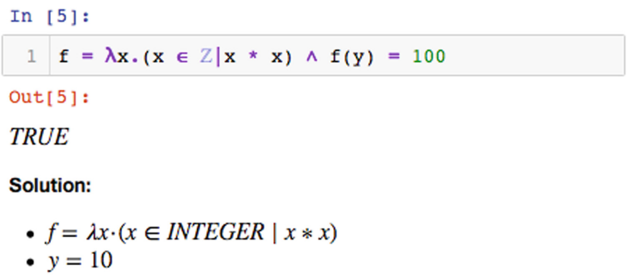
Architecture. Jupyter Notebook [4] is a cross-platform computational notebook interface implemented in Python with a web-based frontend. Originally it was developed under the name IPython Notebook and only supported Python-based notebooks, but it has since been extended to allow using languages other than Python. Support for each language is provided by a Jupyter *kernel*: a language-specific backend that receives input from Jupyter Notebook, processes it using the target language, and returns the results to Jupyter. Jupyter communicates with kernels using a language-agnostic protocol, which allows implementing kernels in languages other than Python. In the case of ProB, the kernel was implemented in Java, as ProB provides a high-level Java API [1], and there is an

A. Raschke et al. (Eds.): ABZ 2020, LNCS 12071, pp. 248–254, 2020.
https://doi.org/10.1007/978-3-030-48077-6_19

existing Java implementation of the Jupyter kernel protocol by the jupyter-jvm-basekernel project [7].

The Jupyter Notebook web interface can also be extended using JavaScript-based plugins. This capability was used to implement syntax highlighting for B.

Interacting with B. At its core, the ProB Jupyter kernel is a simple REPL. It accepts standalone B expressions and predicates as input, which are evaluated or solved using ProB. The results are output as LaTeX formulas and rendered by Jupyter Notebook, as shown in the following screenshot:



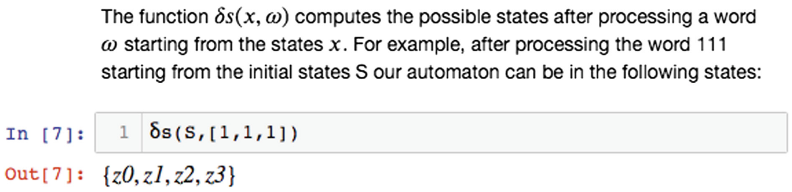
```
In [5]:
f = λx.(x ∈ Z|x * x) ∧ f(y) = 100
Out[5]:
```

TRUE

Solution:

- $f = \lambda x \cdot (x \in INTEGER \mid x * x)$
- $y = 10$

Markup cells can be used to provide documentation for the evaluation cells:



The function $\delta s(x, \omega)$ computes the possible states after processing a word ω starting from the states x. For example, after processing the word 111 starting from the initial states S our automaton can be in the following states:

```
In [7]: δs(S,[1,1,1])
Out[7]:
```

$\{z0, z1, z2, z3\}$

Many ProB features can be accessed using notebook *commands*. For example, prefixing a B expression with the command `:table` displays the result as a table, which is useful for viewing complex values, such as sets of tuples. Additional commands include `:prettyprint` to pretty-print a predicate without evaluating it, `:type` to display an expression's static type, and `:solve` to solve a predicate using ProB's various solver backends (such as Kodkod or Z3).

To load a B machine, the B code can be input directly into a notebook cell, which allows for quick testing and prototyping of short machines. When written this way, the entire B machine needs to be placed in a single notebook cell (it is currently not possible to insert text cells in the middle of the machine), and it cannot refine, extend, or otherwise reference other machines. It is also possible to load external machine files using the `:load` command, which is more convenient for larger machines, and also supports loading machines that reference other machine files.

The loaded machine can be animated, using the `:exec` command to execute operations or events. While a machine is loaded, the input is evaluated in the current state of the animator, meaning that the loaded machine's constants and variables can be used in expressions and predicates. Additional commands such as `:check` and `:browse` are provided to examine the current state of invariants, assertions and operations. It is also possible to exercise ProB's model checker.

ProB's state visualisation features can be called using the `:show` and `:dot` commands, which can for example be used to visualise the current machine state or the animator's state space. The visualisation results are displayed directly in the notebook as raster or SVG images.

Jupyter Notebook's advanced code editing features, such as syntax highlighting and code completion, are also supported by the ProB kernel. Both regular B syntax and custom commands are highlighted, and completion is provided for B keywords, variable names, command names and parameters, etc. The "inspect" feature (accessed using Shift+Tab) provides quick access to command help directly inside the notebook interface.

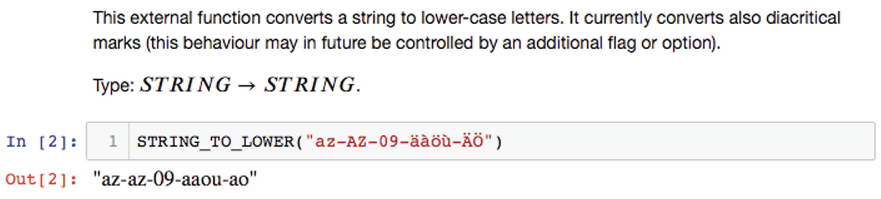
Working with Notebooks. The ProB Jupyter kernel only handles the actual evaluation of the code in the notebook. Jupyter Notebook provides all other parts of the system: including the web frontend responsible for editing notebooks and rendering the kernel's outputs, and the file format used when saving notebooks.

Using the `nbconvert` tool provided by Jupyter, B notebook files can be converted to a variety of standard formats, including HTML, LaTeX, and PDF. This allows distributing notebooks in a format that can be viewed without Jupyter Notebook, although the resulting files cannot be edited and re-executed like the original notebook.

3 Applications

Industrial. As B notebooks can load and animate external machine files, they can be used to document the behavior of existing models. This is conceptually similar to a trace file, with the advantage that notebooks can include not just operation execution steps, but also explanatory text and evaluation/visualisation calls to demonstrate specific aspects of the machine's state.

Some of ProB's own documentation is currently being converted from static documentation pages to B notebooks. The *Modelling Examples* section, e.g., contains pages which start with an introductory text, usually describing a short logic puzzle or part of a real-world use case of ProB, followed by B code fragments modelling the problem in B and explanations of how ProB can be used to visualize, verify or solve the model. The notebook format is well-suited for this kind of documentation: the code in notebooks can be directly executed by the user and the respective visualisation features can be called directly from the notebook. Below is part of the documentation of ProB's external functions. This documentation is automatically up-to-date and users can experiment themselves with the various external functions before integrating them into their models:


This external function converts a string to lower-case letters. It currently converts also diacritical marks (this behaviour may in future be controlled by an additional flag or option).

Type: $STRING \rightarrow STRING$.

```
In [2]:  STRING_TO_LOWER("az-AZ-09-äàöù-ÄÖ")
Out[2]:  "az-az-09-aaou-ao"
```

Teaching. In the context of teaching the B language as well as theoretical computer science in general, B notebooks can be used as a format for writing lecture notes and worksheets.

Lecture notes involving B expressions or machines can be written and distributed as notebooks, allowing students to execute the code for themselves and experiment with modifications. Due to its foundations in set theory, the B language can also be used to express many general theoretical computer science concepts, such as finite automata. These concepts can be demonstrated using B notebooks, taking advantage of the LaTeX output and graph visualisation capabilities to display the results in a format familiar to students.

B notebooks can also be used as a format for exercise sheets. Students are provided with a notebook that contains the exercise text and possibly some initial code. They can solve the exercises directly in the notebook and turn in the finished notebook file with their solutions. The nbgrader [9] extension for Jupyter provides support for writing exercise sheet notebooks: it allows marking cells with exercise text as read-only, and cells with solutions so they are removed when the exercises are distributed to students. The extension also assists with grading and also enables automated verification of solutions.

4 Conclusion, Related and Future Work

A formal model is usually derived from a natural language requirements document. A big issue is that of keeping the formal model and natural language in sync. A related issue is that of traceability, tracing natural language requirements to the formal model. In that setting the idea of literate programming [5], mixing the natural language documentation with the program, is appealing. The Z language [8] has always allowed literate programming by interleaving LaTeX commands with Z constructs. A similar capability for the B language is provided by ProB's LaTeX mode [6]. In comparison, the ProB Jupyter kernel focuses more on interactivity. Individual cells of a B notebook can be quickly edited and re-rendered/evaluated, whereas the ProB LaTeX mode can only render the entire document at once. However, the ability to write LaTeX code directly offers more flexibility in terms of formatting and layout, compared to a B notebook converted to LaTeX or PDF using `nbconvert`.

An open-source Jupyter kernel for TLA$^+$ [3] is available. Its feature set is similar to the basic features of the ProB Jupyter kernel: it supports evaluation of standalone TLA$^+$ expressions, as well as loading and checking of TLA$^+$ models using the TLC model checker.

A previous attempt at implementing a notebook-like interface for B was the ProB worksheet interface [2]. Its design and goals were very similar to our work, but the implementation provided its own custom web UI, server, and file format, mainly because extensible notebook implementations like Jupyter were not available at the time (2012–2013). In comparison, using Jupyter as a base significantly reduces the required implementation and maintenance work, and allows B notebooks to benefit from existing tooling for Jupyter, such as `nbconvert` and `nbgrader`.

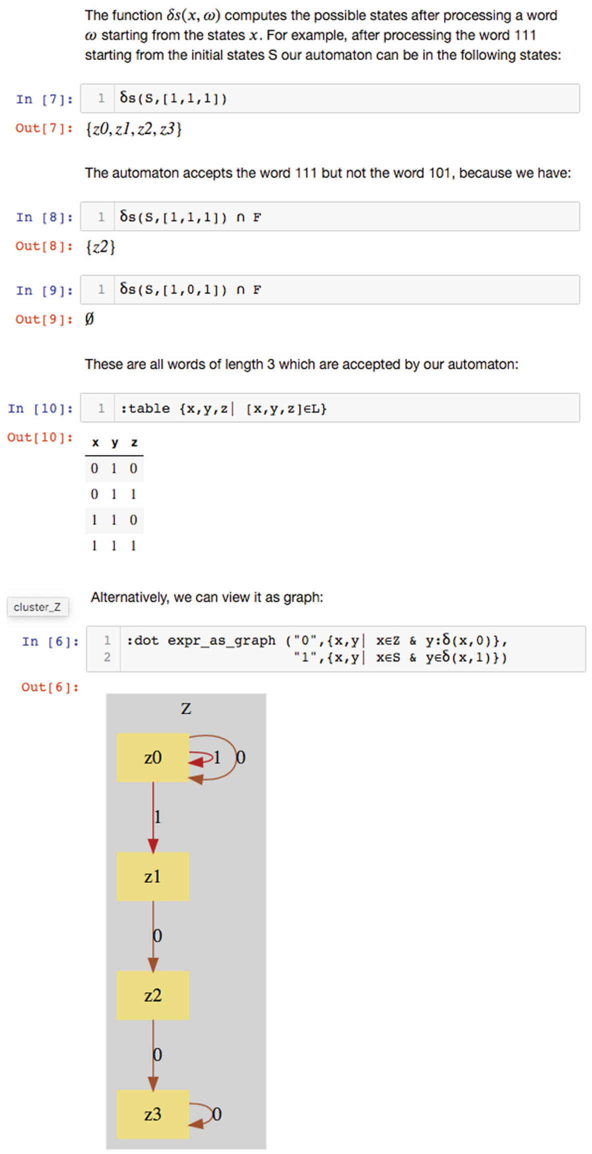
In summary, this new tool provides a notebook interface to a variety of state-based formal methods. Along with some extensions of ProB itself, such as allowing Greek letters or subscripts in identifiers, it is also of use for applications in teaching of discrete mathematics or theoretical computer science.

Our tool is available for download at:

https://gitlab.cs.uni-duesseldorf.de/general/stups/prob2-jupyter-kernel

A Appendix

Below we show two partial screenshots of a notebook for theoretical computer science. Observe that mathematical Unicode symbols, subscripts and Greek letters can be used in the B formulas and machines.



The function $\delta s(x,\omega)$ computes the possible states after processing a word ω starting from the states x. For example, after processing the word 111 starting from the initial states S our automaton can be in the following states:

```
In [7]:  δs(S,[1,1,1])
```

Out[7]: $\{z0, z1, z2, z3\}$

The automaton accepts the word 111 but not the word 101, because we have:

```
In [8]:  δs(S,[1,1,1]) ∩ F
```

Out[8]: $\{z2\}$

```
In [9]:  δs(S,[1,0,1]) ∩ F
```

Out[9]: $\emptyset$

These are all words of length 3 which are accepted by our automaton:

```
In [10]:  :table {x,y,z| [x,y,z]∈L}
```

Out[10]:

x	y	z
0	1	0
0	1	1
1	1	0
1	1	1

cluster_Z

Alternatively, we can view it as graph:

```
In [6]:  :dot expr_as_graph ("0",{x,y| x∈Z & y:δ(x,0)},
                              "1",{x,y| x∈S & y∈δ(x,1)})
```

Out[6]:

References

1. Bendisposto, J., Clark, J.: ProB Handbook. ProB 2.0. https://www3.hhu.de/stups/handbook/prob2/prob_handbook.html#prob2.0. Assessed 30 Jan 2020
2. Goebbels, R.: Worksheets für die Integration mit ProB. Bachelor's thesis, Heinrich-Heine-Universität Düsseldorf, 18 March 2013
3. Kelvich, S.: kelvich/tlaplus_jupyter: Jupyter kernel for TLA$^+$, 9 December 2019. https://github.com/kelvich/tlaplus_jupyter/. Accessed 17 December 2019

4. Kluyver, T., et al.: Jupyter notebooks – a publishing format for reproducible computational workflows. In: Loizides, F., Schmidt, B. (eds.) Positioning and Power in Academic Publishing: Players, Agents and Agendas, pp. 87–90. IOS Press (2016)
5. Knuth, D.E.: Literate programming. Comput. J. **27**(2), 97–111 (1984). https://doi.org/10.1093/comjnl/27.2.97
6. Leuschel, M.: Formal model-based constraint solving and document generation. In: Ribeiro, L., Lecomte, T. (eds.) SBMF 2016. LNCS, vol. 10090, pp. 3–20. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49815-7_1
7. Spencer Park. SpencerPark/jupyter-jvm-basekernel: an abstract kernel implementation for Jupyter kernels running on the Java virtual machine. Revision ccfb7bb1, 14 May 2018. https://github.com/SpencerPark/jupyter-jvm-basekernel/. Accessed 02 Aug 2018
8. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice-Hall, Upper Saddle River (1992)
9. Jupyter Development Team. nbgrader—nbgrader 0.5.4 documentation. Revision 808caf33, 18 July 2017. https://nbgrader.readthedocs.io/en/stable/. Accessed 20 Aug 2018