Bacatá: A Language Parametric Notebook Generator (Tool Demo)

Mauricio Verano Merino
Eindhoven University of Technology
Eindhoven, The Netherlands
m.verano.merino@tue.nl

Jurgen Vinju
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
Eindhoven University of Technology
Eindhoven, The Netherlands
jurgen.vinju@cwi.nl

Tijs van der Storm
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
University of Groningen
Groningen, The Netherlands
storm@cwi.nl

Abstract

Interactive notebooks allow people to communicate and collaborate through a single rich document that might include live code, multimedia, computed results, and documentation, which is persisted as a whole for reproducibility. Notebooks are currently being used extensively in domains such as data science, data journalism, and machine learning. However, constructing a notebook interface for a new language requires a lot of effort. In this tool paper, we present Bacatá, a language parametric notebook generator for domain-specific languages (DSL) based on the Jupyter framework. Bacatá is designed so that language engineers may reuse existing language components (such as parsers, code generators, interpreters, etc.) as much as possible. Moreover, we explain the design of Bacatá and how DSL notebooks can be generated with minimum effort in the context of the Rascal meta programming system and language workbench.

CCS Concepts • **Software and its engineering** → *Application specific development environments*; *Domain specific languages*;

Keywords Interactive computing, language workbenches, domain-specific languages, literate programming

ACM Reference Format:
Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. 2018. Bacatá: A Language Parametric Notebook Generator (Tool Demo). In *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering (SLE '18), November 5–6, 2018, Boston, MA, USA*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3276604.3276981

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SLE '18, November 5–6, 2018, Boston, MA, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6029-6/18/11…$15.00
https://doi.org/10.1145/3276604.3276981

1 Introduction

Interactive notebooks have received much attention in recent years due to the benefits they provide regarding immediate feedback, reproducibility, and collaborative features. Notebooks capture a *computational narrative* interleaving code, computed results, interactive visualizations, and documentation, in a single persisted document. Notebooks have become immensely popular in fields such as mathematics, data science, data journalism, and machine learning.

The Jupyter notebook framework [11] is a popular platform for writing and sharing computational narratives. This platform comes with built-in support for Python (IPython), but it provides an API for extending the framework to other languages, called *language kernels*. These kernels capture language specific aspects, such as how to highlight syntax elements, how to call the interpreter or compiler, and how to visualize computed results.

Developing a language kernel from scratch requires a lot of effort and communication with Jupyter's low-level wire protocol. Nevertheless, interactive notebooks would provide a valuable addition to the toolbox of generic language services offered by language workbenches [4]. This would open up the interactive notebook metaphor for DSLs developed using these language workbenches.

In this tool paper, we present an extended view of Bacatá [14], a language parametric notebook generator, based on the Jupyter platform. Bacatá hides the low-level complexity of Jupyter's wire protocol, providing generic hooks for registering language services. Bacatá has been integrated in the Rascal language workbench [10], which allows extensive reuse of language components defined with Rascal. As a result, obtaining a notebook interface for a DSL becomes a matter of writing a few lines of code. In addition, we present Bacatá's support for fully interactive computed results through Rascal's web UI framework (Salix). DSLs that use this library can thus be run from within a Bacatá notebook, with virtually no additional effort.

2 Bacatá

Bacatá is a language parametric interface between the Jupyter platform and the Rascal language workbench. This interface

generates Jupyter language kernels that reuse language components such as grammars, parsers, and Read-Eval-Print Loops (REPLs). In this section, we describe Bacatá's general architecture and Bacatá-Core.

2.1 Architecture

Figure 1 depicts a general overview of Bacatá's architecture, which highlights its most essential components. Two primary actors interact with Bacatá, language engineers and end-users. Language engineers use Bacatá to generate Jupyter language kernels. Whereas end-users utilize a language kernel, previously generated by a language engineer, to interact with the language through a notebook front-end.

Bacatá consists of two main components, Bacatá-Core and Bacatá-Rascal. On the one hand, Bacatá-Core abstracts away the communication layer between Jupyter and the language. It provides a generic language protocol interface (similar to Microsoft's Language Server Protocol [15]), that could be implemented for language workbenches other than Rascal. This component is responsible for the interaction between the executable code written in a notebook and its execution.

On the other hand, Bacatá-Rascal implements the interface offered by Bacatá-Core, and provides the means for languages developed using Rascal to be connected to Bacatá-Core. To use those services, Bacatá-Rascal takes as input an Algebraic Data Type (ADT) called `Kernel`. A `Kernel` object is the entry-point for generating and re-using language-specific artifacts such as CodeMirror [6] modes, language interpreters, completion functions, and interactive visualizations. After a language engineer generates a language kernel using Bacatá, this language becomes part of the supported languages of the current Jupyter environment.

From the end-user perspective, Bacatá-Rascal and Bacatá-Core are hidden, since they simply choose their desired language kernel from the Jupyter notebook interface. After selecting the language kernel, Jupyter automatically instantiates the language REPL through Bacatá, which allows the user to execute code.

2.2 Bacatá-Core

Jupyter offers a protocol called the *wire protocol* [8], which is a communication protocol implemented using ZeroMQ sockets [1]. This protocol describes a set of sockets and messages that enable the interaction between third-party languages and the Jupyter platform. Similarly, it describes the structure of the messages and how to exchange those messages among different sockets used by Jupyter. To extend Jupyter's default set of languages, language engineers need to implement a *language kernel.* A language kernel is a program that runs user code. To create a language kernel from scratch, language engineers must follow the low-level wire protocol.

Bacatá-Core offers the `ILanguageProtocol` interface that enables the communication between Jupyter and a language

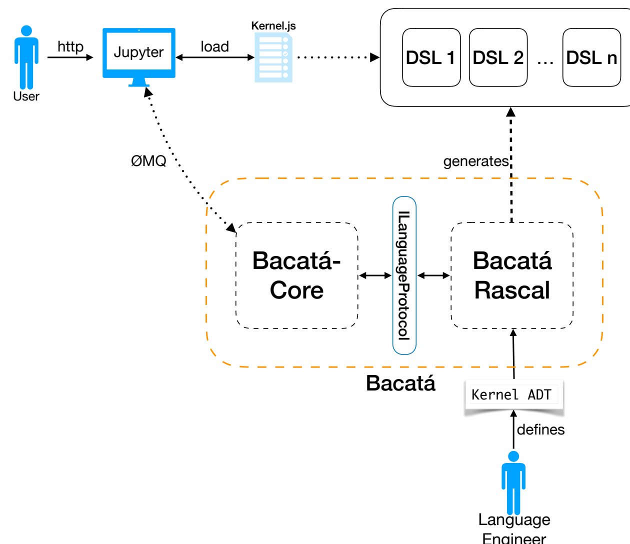

Figure 1. General overview of Bacatá's architecture.

```
data REPL
  = repl(Result(str) handler,
         Completion(str) completor);

alias Completion
  = tuple[int pos, list[str] suggestions];

data Result
  = text(str result, list[Message] messages);
```

Listing 1. REPL ADT.

in a generic way. The primary purpose of this layer is to abstract the implementation complexity of the wire protocol and its related socket management. Therefore, the language developer can focus on the language engineering layer. For DSLs developed within Rascal, we have implemented this interface in a language parametric way. In other words, it pretends to be a particular language kernel, but delegating all language specific service requests to a language implementation in Rascal.

3 Bacatá-Rascal

As explained before, to support new languages by Jupyter, developers have to implement a language kernel. Bacatá offers a Jupyter language kernel generator for DSLs written within the Rascal LWB.

To use Bacatá's kernel generator, a language engineer needs to define a function that produces a `REPL` ADT, which will be used as the language's interactive interpreter. The `REPL` ADT is defined as shown in Listing 1.

1. The language engineer calls the Bacatá function `bacata` which accepts one argument, a value of type `Kernel`.

```
data Kernel
  = kernel(str language, loc project,
      str replFunction, loc logo = |tmp:///|);
```

Listing 2. Kernel ADT.

The `Kernel` type (shown in Listing 2), defines the configuration parameters for Bacatá-Core to obtain language specific information (e.g., name and location of the logo of the language) and find relevant resources, such as the fully qualified name of the REPL implementation to be used.

2. The generated kernel assumes that there is a `replFunction` which returns a `REPL` value. The `REPL` data type is shown in Listing 1. It encapsulates two functions, the `handler` for interpreting code, and a `completor` for code completion. The respective result types of each function are also shown in Listing 1.
3. Optionally, language engineers can generate CodeMirror syntax-highlighting modes. This is achieved by providing a value of the data type `Mode` (Listing 4), which can be automatically derived from the language's grammar.

The function `bacata` takes a `Kernel` object to generate a JSON file called *kernel.json* (Listing 3). This file contains different data such as Jupyter's connection details (e.g., ZMQ socket types and ports), language REPL execution instructions, and language-specific information (e.g., name and logo). When an end-user requests to generate a notebook for a specific language, all this data is being forwarded to Bacatá. Then, after generating the JSON file, Bacatá automatically registers the language as part of the Jupyter supported languages.

```
{
 "argv": [
    "java", "-jar",
    "/Mauricio/bacata/bacata-dsl.jar",
    "{connection_file}",
    "home:///projects/Calc",
    "Repl::myRepl",
    "Calc"
   ],
   "display_name": "Calc",
   "language": "Calc"
}
```

Listing 3. Generated Jupyter kernel for Calc

3.1 Syntax Highlighting

Jupyter's input cells highlighting is based on the CodeMirror editor[1], which supports easily customizable syntax highlighting using *modes*. Modes are like so-called "Textmate grammars"[2], which are used by editors such as Textmate, VS Code, SublimeText, and many others.

[1] https://codemirror.net

```
data Mode
  = mode(str name, list[State] states);

data State = state(str name, list[Rule] rules);

data Rule
  = rule(str regex, list[str] tokens,
         str next = "", bool indent = false,
         bool dedent = false);
```

Listing 4. Syntax Mode ADT

The `Mode` data type shown in Listing 4 models such modes. A mode has a name and contains several state definitions. Each `State` then defines a few rules that are applicable in that state. A `Rule` defines a regular expression to match a particular substring and assigns a list of token types to it that will determine its visual appearance. After a rule has matched, it may transit to another state via the `next` property. The optional booleans `indent` and `dedent` control auto indentation in block constructs.

To support syntax highlighting in Bacatá-generated notebooks, the `bacata` function supports an optional additional argument for the mode:

```
Notebook bacata(Kernel k, Mode mode=mode("", [])) {...}
```

Language engineers can define such modes manually. However, Bacatá also features a function to generate simple modes for keyword highlighting from a Rascal grammar using reflection.

3.2 Interactive Visualizations

Jupyter notebooks run in the browser, so this allows output cells to contain almost arbitrary interactive visualizations, beyond plain text output. Bacatá supports fully interactive, stateful graphical user interfaces in output cells through integration with Rascal's web UI framework Salix[3], which emulates Elm's[4] architecture. Salix supports all the standard HTML and SVG elements, and features integration with graph rendering libraries[5], and chart frameworks[6].

A Salix application is encapsulated as a value of type `App[&T]` where the type parameter `&T` indicates the type of the application data model. Under the hood, an `App` encapsulates a view to draw UIs using HTML and SVG elements, and an update function to update the model when a user event is triggered. Bacatá makes use of such Salix applications by

[2] https://manual.macromates.com/en/language_grammars
[3] https://github.com/cwi-swat/salix
[4] http://www.elm-lang.org
[5] https://github.com/dagrejs
[6] https://developers.google.com/chart/

```
In [1]:  1  x = 2
Out[1]:  2

In [2]:  1  y = 1 + x
Out[2]:  3

In [3]:  1  show 2 * y
Out[3]:  x: 2
         y: 6
         2 * y: 12
```

Figure 2. Interactive debugging of a Calc expression.

allowing Salix `Apps` as output of the REPL. This is achieved by extending the `Result` data type of Listing 1:

```
data Result
  = ...
  | app(App[&T] app, list[Message] messages);
```

This kind of result can be used to produce fully functional stateful output cells, leveraging all UI features of Salix.

A Salix application consists of three functions. The first one produces the initial model. The second one is the `view` function, which takes a model and draws the UI. Finally, the `update` function updates the model.

An example of a fully interactive output cell is illustrated in Figure 2. It shows an interactive debugger for a simple calculator language (Calc). The language consists of commands and expressions. Commands consist of assignments and expression evaluation. Expression forms are variables, numbers, multiplication, and addition. Commands are executed using a function, which returns a number and a (possibly updated) environment. Expressions simply evaluate to numbers. In Figure 2 the user has typed in two assignments to variables `x` and `y`, and then invokes the `show`-command to inspect the effect of the current variable bindings on the expression `2 * y`. The result is two slider widgets for variable `x` and `y`, together with current evaluation of `2 * y`. When changing the slider for `x` or `y` the new result will be live updated on the last line. We required 50 SLOCs to define the notebook for the Calc language, including the definition of the REPL and the Salix application for debugging expressions.

Additionally, we have generated notebooks for three other DSLs, namely Halide* [17], QL [4], and SweeterJS[7].

4 Related Work

Bacatá can be positioned in an extensive line of research in program environment generation [2, 4, 7, 9, 18, 20, 22]. Currently, this work is centered around the concept of language workbenches, a term popularized by Fowler [5]. In his essay, he explains a brief history of the language-oriented programming, their pros and cons, and how IDE tooling has become essential for the viability of language oriented programming, and learning and using DSLs.

Language workbenches provide language parametric tools, meta languages, and techniques to lower the cost of DSL engineering. Bacatá aims to do the same for notebooks. Specifically, interactive notebooks provide a different user interface for code and documentation. Orthogonal to, but not in conflict with more traditional IDE or editor styles.

Concerning interactive computing, Cook [3] and Nagar [16] have highlighted the importance of this paradigm of software development. Cook [3], shows the consequences of adopting this paradigm and how it affects the way we write code based on immediate responses. While Nagar [16] shows a Python way of working using interactive computing, and how it has reduced the learning curve of a programming language if the user can experiment with commands and expressions.

Notebooks integrate the use of narrative in software development, literate programming [12, 19], interactive computing, and collaboration. Turner et al. [21] found notebooks useful as a way of supporting cooperative work and sharing information with non-technical staff. This is aligned with the perspective of using notebooks for DSLs that have a non-programmer audience. However, they found it difficult to differentiate between formal an informal information. Similarly, Malony et al. [13] performed computational experiments using a notebook environment, called the Virtual Notebook Environment (ViNE).

5 Conclusions

Constructing interactive notebooks for new languages requires a lot of effort, especially in the context of DSLs, where the engineering trade-offs and design cycle is different from general purpose languages. In this tool paper, we have presented Bacatá, a language parametric notebook generator based on the Jupyter framework. Given existing language components, such as parsers, interpreters, type checkers, etc., Bacatá reduces the effort of obtaining an interactive notebook interface to writing a few lines of code that wires language components together.

We described the core architecture of Bacatá and presented how the interface is exposed within the Rascal language workbench. Next to the usual notebook features (executing code, code completion, and highlighting), we have shown how Bacatá supports fully interactive output cells using Rascal's web-based GUI framework Salix.

Acknowledgments

This material is based upon work supported by the Impuls II cooperation project between Océ and TU Eindhoven.

[7]https://github.com/cwi-swat/hack-your-javascript

References

[1] Faruk Akgul. 2013. *ZeroMQ.* Packt Publishing.

[2] Philippe Charles, Robert M Fuhrer, Stanley M Sutton Jr, Evelyn Duesterwald, and Jurgen Vinju. 2009. Accelerating the creation of customized, language-Specific IDEs in Eclipse. In *ACM Sigplan Notices*, Vol. 44. ACM, 191–206.

[3] Joshua Cook. 2017. *Interactive Programming.* Apress, Berkeley, CA, 49–70. https://doi.org/10.1007/978-1-4842-3012-1_3

[4] Sebastian Erdweg, Tijs van der Storm, Markus Volter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2015. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures* 44 (2015), 24 – 47. https://doi.org/10.1016/j.cl.2015.08.007 Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014).

[5] Martin Fowler. 2015. Language Workbenches: The Killer-App for Domain Specific Languages? (2015). Retrieved June 18, 2018 from https://www.martinfowler.com/articles/languageWorkbench.html

[6] Marijn Haverbeke. 2007–2018. CodeMirror. (2007–2018). http://codemirror.net/

[7] Jan Heering and Paul Klint. 2000. Semantics of Programming Languages: A Tool-oriented Approach. *SIGPLAN Not.* 35, 3 (March 2000), 39–48. https://doi.org/10.1145/351159.351173

[8] Jupyter. 2015. The wire protocol. (2015). Retrieved July 24, 2017 from http://jupyter-client.readthedocs.io/en/latest/messaging.html#the-wire-protocol

[9] P. Klint. 1993. A Meta-environment for Generating Programming Environments. *ACM Trans. Softw. Eng. Methodol.* 2, 2 (April 1993), 176–201. https://doi.org/10.1145/151257.151260

[10] Paul Klint, Tijs van der Storm, and Jurgen Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM '09)*. IEEE Computer Society, Washington, DC, USA, 168–177. https://doi.org/10.1109/SCAM.2009.28

[11] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. 2016. Jupyter Notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt (Eds.). IOS Press, 87 – 90.

[12] Donald E. Knuth. 1984. Literate Programming. *Comput. J.* 27, 2 (May 1984), 97–111. https://doi.org/10.1093/comjnl/27.2.97

[13] Allen D. Malony, Jenifer L. Skidmore, and Matthew J. Sottile. 1999. Computational experiments using distributed tools in a web-based electronic notebook environment. In *High-Performance Computing and Networking*, Peter Sloot, Marian Bubak, Alfons Hoekstra, and Bob Hertzberger (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 381–390.

[14] Mauricio Verano Merino, Jurgen Vinju, and Tijs van der Storm. 2017. Bacatá: a generic notebook generator for DSLs *(Domain-Specific Language Design and Implementation workshop, DSLDI '17).*

[15] Microsoft. 2018. Language Server Protocol. (2018). https://microsoft.github.io/language-server-protocol

[16] Sandeep Nagar. 2018. *IPython.* Apress, Berkeley, CA, 31–45. https://doi.org/10.1007/978-1-4842-3204-0_3

[17] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. 2012. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. *ACM Trans. Graph.* 31, 4 (July 2012), 32:1–32:12.

[18] Thomas Reps and Tim Teitelbaum. 1984. The synthesizer generator. *ACM SIGSOFT Software Engineering Notes* 9, 3 (1984), 42–48.

[19] Johannes Sametinger. 1997. *Literate Programming.* Springer Berlin Heidelberg, Berlin, Heidelberg, 211–216. https://doi.org/10.1007/978-3-662-03345-6_18

[20] Emma Söderberg and Görel Hedin. 2011. Building semantic editors using JastAdd: tool demonstration. In *Proceedings of the Eleventh Workshop on Language Descriptions, Tools and Applications.* ACM, 11.

[21] Phil Turner and Susan Turner. 1997. *Supporting Cooperative Working Using Shared Notebooks.* Springer Netherlands, Dordrecht, 281–295. https://doi.org/10.1007/978-94-015-7372-6_19

[22] M.G.J. van den Brand, A. van Deursen, J. Heering, H.A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P.A. Olivier, J. Scheerder, J.J. Vinju, E. Visser, and J. Visser. 2001. The ASF+SDF Meta-Environment: A Component-Based Language Development Environment. *Electronic Notes in Theoretical Computer Science* 44, 2 (2001), 3 – 8. https://doi.org/10.1016/S1571-0661(04)80917-4 LDTA'01, First Workshop on Language Descriptions, Tools and Applications (a Satellite Event of ETAPS 2001).