

Open Piping: Towards an Open Visual Workflow Environment

BOISVERT, Charles <<http://orcid.org/0000-0002-3069-5726>>, ROAST, Chris <<http://orcid.org/0000-0002-6931-6252>> and URUCHURTU, Elizabeth <<http://orcid.org/0000-0003-1385-9060>>

Available from Sheffield Hallam University Research Archive (SHURA) at:

<https://shura.shu.ac.uk/24549/>

This document is the Accepted Version [AM]

Citation:

BOISVERT, Charles, ROAST, Chris and URUCHURTU, Elizabeth (2019). Open Piping: Towards an Open Visual Workflow Environment. In: MALIZIA, Alessio, VALTOLINA, Stefano, MORCH, Anders, SERRANO, Alan and STRATTON, Andrew, (eds.) End-User Development. Lecture Notes in Computer Science book (11553). Springer, 183-190. [Book Section]

Copyright and re-use policy

See <http://shura.shu.ac.uk/information.html>

1 Open Piping: Towards an Open Visual 2 Workflow Environment

3 *Charles Boisvert* *Chris Roast* *Elizabeth Uruchurtu*

4 Department of Computing. Communication and Computing Research Centre.
5 Sheffield Hallam University, City Campus, Howard Street, Sheffield S1 1WB, UK
6 *initial.surname@shu.ac.uk*

7 **Abstract**

8 The most popular visual programming tools focus on procedural, object-oriented and
9 event-based programming. This paper describes a boxes-and-wires functional programming
10 tool, aimed to be accessible to novice programmers, while also supporting open access to the
11 specified processes, executable programs and results for study and deployment.

12 **Keywords** Computer science education, data science, functional programming, end-user
13 programming

14 **1 Introduction**

15 Visual, block-based environments such as ALICE [4] or Scratch [17] have recently
16 transformed the teaching of computing [9, 1].

17 Yet this development in procedural and object-oriented programming tools has not
18 disseminated to analysing and processing data. For example, the nifty assignments
19 repository of computing assessment ideas [15, 14] contains 107 assignments,
20 collected for their quality, but only eight of these incorporate work with a real data set.

21 Of particular concern to us, at Sheffield Hallam University, is adapting our tools,
22 teaching methods and resources in order to facilitate access to and process of data by
23 students at any level. Specific interest areas have been working with open data
24 advocacy groups [12] and making data analytic tools more available [19].

25 The Open Piping project pursues this idea with an open-source functional
26 programming environment and visual data flow interface for data processing¹.

¹ <http://boisvert.me.uk/openpiping>

2 Project motivations

Open Piping is a visual functional programming environment, based on a boxes and wires model, intended for data processing applications.

Visual boxes and wires environments are common [11, 13], including some in commercial [8] and scientific [7] use. But in many cases, the value of the tools is limited due to the poor transparency of the processes and technology they implement.

Take the case of the popular - until its end in 2015 - Yahoo pipes [13]. To execute pipes on systems of their choice, users had to go through a complex export process. This was their only option when Yahoo support ended.

Open piping aims to propose an ease of use comparable to commercial tools, in an open architecture to facilitate development flexibility, reuse and allow richer exchanges between users.

2.1 Open by design

Our ambition is to propose a graphical tool for user-defined data processes, which would include, by design, the transparency and flexibility needed to apply user-defined processes in a range of languages and environments. Open piping aims to be at once:

Open. That is, Open Source; the system's source code is available under the GNU licence. But so is the notation used to define processes. Any user process can then be transformed from this notation into executable code in a target programming language.

Interoperable. The process specification format is openly available, and uses a human-readable, JSON formatted S-expression. This is needed to ensure the interoperability of the system with any manner of services, such as alternative end-user interfaces, new languages or process hosting and remote execution tools.

Easy to use. The user interface makes it easy to define data flows and shows clearly the relation between data flow, resulting S-expression, and executable functionality.

With resulting processes easy to deploy. The ability to choose from multiple languages and standards for services and content integration, would facilitate the re-use of user-defined processes in different environments, such as within content-management systems, as web or application widgets, or within a service-oriented architecture.

Altogether, these characteristics aim to ensure that users can easily define the processes they want to operate on data, while also retaining control of these processes to use them in new environments.

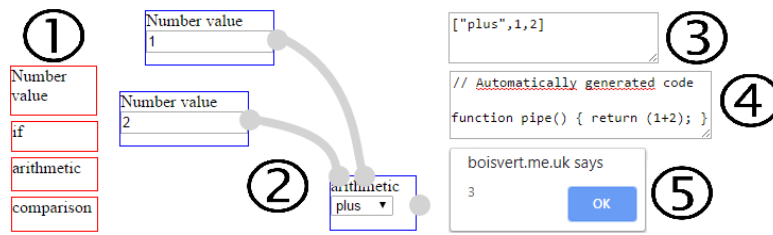


Fig. 1: Open piping main interface elements

3 Open piping Operation

3.1 System architecture

The boxes-and-wires model describes the directed acyclic graph for a function, with the boxes representing functions and the wires, the data to which they apply.

Configuration data defines base functions available to the end-user. This information at once determines primary graphical blocks, provides access to basic processing capabilities, and limits that access, for security, to a chosen set with defined functionality.

The end-user defines a function by wiring elementary blocks. This function is translated into an S-expression in JSON, which can be compiled into an executable function in any number of languages, provided that calls to the primitive functions can be defined.

The interface elements presented fig. 1 sum up the use of Open Piping. The end-user chooses elementary blocks (1) to define a flow (2) which is translated to a symbolic expression (3) encoded in JSON to use the many existing tools for this format. The expression is then interpreted (4) and executed (5).

3.2 Defining and encoding a data flow

The block description and interface configuration also uses the JSON format. For instance, fig. 2 shows the configuration lines to define the box representing arithmetic operations. The user can choose add, subtract, divide, or multiply from a single 'arithmetic' box.

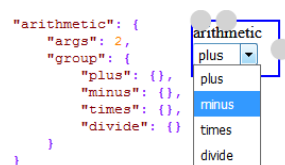


Fig. 2: Defining and representing graphically a box of arithmetic functions

An example data flow is presented fig. 3. The web interface uses the JSPlumb library [10] to manipulate and represent the screen objects. Traversing the graph recursively provides a symbolic expression. An advantage of symbolic expressions is that code

85 remains close to existing languages such as LISP or Scheme. For instance, in a
 86 LISP-like language, the workflow figure 3 results in the structure:

87 `(if (isNumber 15) (plus 1 15) "not a number")`

[1]

88

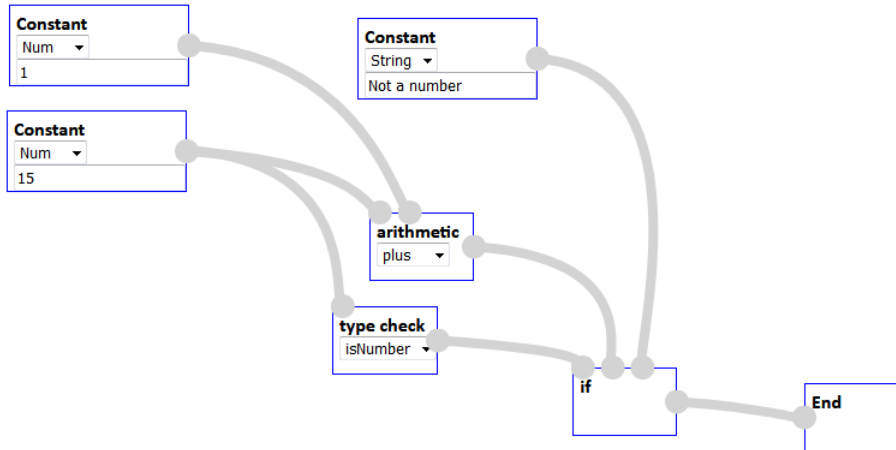


Fig. 3: An example workflow

89 Another benefit of S-expressions is the original argument for this notation: executable
 90 code and data follow the same conventions. This facilitates the processing of an
 91 expression like line [1] in multiple environments.

92 The expression is encoded in JSON, to provide to the interpreter. JSON's wide use
 93 and readability make it particularly suitable to this purpose. The encoding follows
 94 these simple rules:

- 95 • JSON notation defines *objects*, *arrays*, *strings*, *numbers* and the values `true`,
 96 `false`, and `null`. Our encoding relies on all but *objects*.
- 97 • Atomic values are *strings*, *numbers* and the values `true`, `false`, and `null`.
- 98 • Lists are represented by a JSON array. Each element of the list can be an
 99 atomic value or a list, and so on recursively.

100 Respecting this convention, the process shown fig. 3 is written:

101 `["if", ["isNumber",15], ["plus",1,15], "Not a number"]`

[2]

102

103 3.3 Interpret a symbolic expression in executable language

104 To allow the execution of the same expression in diverse environments, we rely on
 105 characteristics present in most programming languages - use of variables, of a means
 106 of conditional execution, of functions - but we must provide elementary information to

107 support the interpretation in each language. These data are themselves written in
108 JSON.

109 To illustrate the interpretation process, let us study the case of interpreting expression
110 [2] above in JavaScript and JQuery.

111 The interpretation relies on a list of predefined functions and string substitutions for
112 the language:

```
113     "plus": {"args": "a,b", "sub": "@a+@b"}  
114     "if": {"args": "a,b,c", "sub": "@a?@b:@c"}  
115     "isNumber": {"args": "n", "body": "return $.isNumeric(n);"} [3]
```

115 Some operators are interpreted by substituting character chains to form the target
116 code. Arithmetic operators like + use this technique, but so do conditionals, which we
117 interpret in JavaScript with the ternary operator. Functions are identified and
118 composed from arguments and body information. So [4] contains all the information
119 needed to interpret the example completely.

120 Using this data, the expression is interpreted recursively. First the expression

```
121     ["isNumber", 15] [4]
```

122
123 results in the definition of function `isNumber`,

```
124     function isNumber(n) {return $.isNumeric(n);} [5]
```

125
126 and into one function call. The `plus` function is then interpreted by substituting strings,
127 and finally `if` to compose the overall result:

```
128     process(isNumber(15)?(1+15):"Not a number"); [6]
```

129
130 We can see that the interpretation of a user-defined function is simple; to be able to
131 execute a process in a given language, we simply need to define and execute safely
132 the primitive functions required.

133 3.4 Overcoming visual limitations

134 The graphical model shown above should support end-user's understanding and
135 programming of simple processes. However, based on our experience and prior
136 research such as [18, 3, 2], we speculate that several aspects of the visualisation are
137 not easily represented in ways that end-users spontaneously understand. Here, we
138 present a number of potential solutions to support end-users as programs become
139 more complex.

140 3.4.1 Coordinating visual code with results

141 Visual programming can support end-users with a number of displays - the results of
 142 a program, of its code, of its execution. The wires and boxes model is a form of visual
 143 code, but many systems show a visual representation of execution results.

144 Coordinated views can also apply to viewing code. Yahoo pipes [16] is an example of
 145 this approach: its visualisation showed code, in boxes and wires form, along with a
 146 sample of the data resulting from it. Users could also select subsets of the code to
 147 view its result. This supported end-users with a presentation of the code, of some
 148 results, and of execution information (as partial execution results), as well as
 149 debugging support by means of choosing code subsets to test.

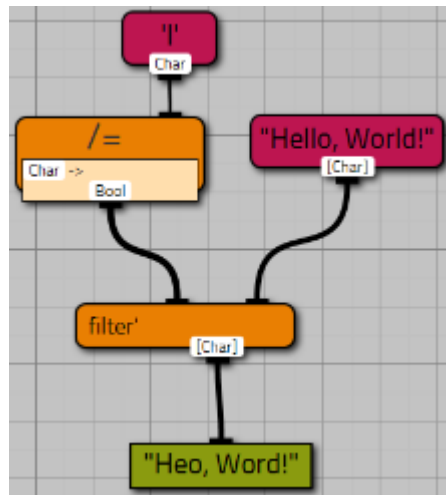


Fig. 4: Viskell shows data type explicitly

150 3.4.2 Data Typing

151 The boxes and wires model shown in our example fig. 3 does not show any type
 152 information. Typing has many advantages for novice programmers, in particular
 153 limiting errors by constraining the validity of constructs, ensuring security, and
 154 facilitating debugging.

155 Typing can be presented in textual form, a solution adopted by Viskell as shown in fig.
 156 4 [20]. An alternative is visual clues, such as colour, shape, or icons: languages like
 157 MIT Scratch [17] adopt this approach, and use the added advantage of shape as a
 158 metaphor for syntactic validity. Type can also be implemented in the language and
 159 enforced in the interaction, yet not presented visually: that is the solution adopted by
 160 Yahoo pipes, which enforces type checking with the impossibility of connecting a wire
 161 to a box if types do not match, but give no visual typing clue.

162 3.4.3 Representing Conditionals

163 Conditional execution is one of the basic elements of programming. A three-argument
 164 function, for the Boolean that determines which branch is executed, and each of the

165 two branches, is a suitable technical answer, but as the prototype workflow shown
 166 earlier in fig. 3, visual clues in support of the user are clearly lacking.

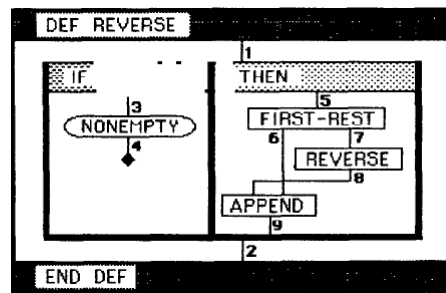


Fig. 5: Prograph shows the conditional branches within two frames for clarity.

167 Prograph [5] solves this problem by adding to boxes and wires a third construct,
 168 *frames*, for sections of code that are end-users should consider separately.

169 3.4.4 First class functions

170 First-class functions are a fundamental benefit of functional programming, but also a
 171 difficult concept to represent in ways that users can understand and control. The
 172 earlier illustration of Viskell (fig. 4), shows a lambda-expression within the model,
 173 supported by textual type annotation: not every end-user will find it clear.

174 An alternative relies on the same notion of frames as for conditionals: a function that
 175 accepts another as a parameter, represents that parameter within a frame. Yahoo
 176 pipes adopts that solution, albeit for a limited use of first-class functions: it implements
 177 user actions to drop a box into a functional parameter slot. [6] have investigated the
 178 primitives needed to represent completely the power of first-class functions within
 179 frames, but the solution is not an easy visualisation of the notion.

180 4 Conclusion and future work

181 The structure of our system lets users retain control of their processes. In particular:

182 **Limits to processing capabilities** are not inherent to the system, but instead to the
 183 environment in which the process is deployed, for example by setting a processing
 184 time limit.

185 **The visual language** is loosely coupled to the execution environment, by producing a
 186 function definition in an open intermediate representation; this ensures that changes
 187 to the visual interface, to the target language, and to the execution environment are
 188 independent.

189 **Risks of code injection** are limited by transmitting the symbolic expression to an
 190 interpretation environment hosted with the execution environment, rather than
 191 communicate executable code, as well as by defining in the interpreter what primitive
 192 functions are allowable.

193 We believe that these characteristics can support adoption and self-learning through
194 greater open access to computation.

195 Currently our prototype ensures that end-users can define processes, and
196 demonstrates the compilation from the S-expression to JavaScript and execution.
197 Multiple environments common on web servers and clients are considered - e.g.
198 JQuery, PHP, node.js, etc, as well as deployment of executable results in new
199 systems.

200 Developing this prototype's capabilities to support users further, will require a balance
201 of technical feasibility, theoretical clarity and empirical evidence to identify the most
202 appropriate solutions.

203 References

- 204 [1] D. Adshead, C. Boisvert, D. Love, and P. Spencer. Changing culture: Educating
205 the next computer scientists. In *Proceedings of the 2015 ACM Conference on*
206 *Innovation and Technology in Computer Science Education*, pages 33–38. ACM,
207 2015.
- 208 [2] A. F. BLACKWELL. Pictorial representation and metaphor in visual language
209 design. *Journal of Visual Languages Computing*, 12(3):223 – 252, 2001.
- 210 [3] A. F. Blackwell. The reification of metaphor as a design tool. *ACM Trans.*
211 *Comput.-Hum. Interact.*, 13(4):490–530, Dec. 2006.
- 212 [4] S. Cooper, W. Dann, and R. Pausch. Alice: a 3-d tool for introductory
213 programming concepts. In *Journal of Computing Sciences in Colleges*,
214 volume 15, pages 107–116. Consortium for Computing Sciences in Colleges,
215 2000.
- 216 [5] P. Cox and T. Pietrzykowski. Advanced programming aids in prograph. In
217 *Proceedings of the 1985 ACM SIGSMALL symposium on Small systems*, pages
218 27–33. ACM, 1985.
- 219 [6] A. Fukunaga, W. Pree, and T. D. Kimura. Functions as objects in a data flow
220 based visual language. In *Proceedings of the 1993 ACM Conference on*
221 *Computer Science, CSC '93*, pages 215–220, New York, NY, USA, 1993. ACM.
- 222 [7] D. Hull, K. Wolstencroft, R. Stevens, C. Goble, M. R. Pocock, P. Li, and T. Oinn.
223 Taverna: a tool for building and running workflows of services. *Nucleic acids*
224 *research*, 34(suppl 2):W729–W732, 2006.
- 225 [8] N. Instruments. What is labview. <http://www.ni.com/en-gb/shop/labview.html>.
226 Accessed: 2019-30-04.
- 227 [9] S. P. Jones, T. Bell, Q. Cutts, S. Iyer, C. Schulte, J. Vahrenhold, and B. Han.
228 Computing at school. *International comparisons*. Retrieved May, 7:2013, 2011.

- 229 [10] I. JSPlumb. Jsplumb toolkit documentation. <https://jsplumbtoolkit.com/docs.html>.
230 Accessed: 2017-13-04.
- 231 [11] D. Le-Phuoc, A. Polleres, G. Tummarello, and C. Morbidoni. Deri pipes: visual
232 tool for wiring web data sources. *^(Eds.):'Book DERI pipes: visual tool for wiring
233 web data sources'(2008, edn.), 2008.*
- 234 [12] M. Love, C. Boisvert, E. Uruchurtu, and I. Ibbotson. Nifty with data: Can a
235 business intelligence analysis sourced from open data form a nifty assignment?
236 In *Proceedings of the 2016 ACM Conference on Innovation and Technology in
237 Computer Science Education, ITiCSE '16*, pages 344–349, New York, NY, USA,
238 2016. ACM.
- 239 [13] T. O'Reilly. Pipes and filters for the internet. [http://radar.oreilly.com/2007/02/
240 pipes-and-filters-for-the-inte.html](http://radar.oreilly.com/2007/02/pipes-and-filters-for-the-inte.html). Accessed: 2016-10-10.
- 241 [14] N. Parlante. Nifty assignments. <http://nifty.stanford.edu>. Accessed: 2016-01-12.
- 242 [15] N. Parlante, J. Popyack, S. Reges, S. Weiss, S. Dexter, C. Gurwitz, J. Zachary,
243 and G. Braught. Nifty assignments. In *ACM SIGCSE Bulletin*, volume 35, pages
244 353–354. ACM, 2003.
- 245 [16] M. Pruett. *Yahoo! pipes*. O'Reilly, 2007.
- 246 [17] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond,
247 K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch:
248 programming for all. *Communications of the ACM*, 52(11):60–67, 2009.
- 249 [18] C. Roast, R. Leitão, and M. Gunning. Visualising formula structures to support
250 exploratory modelling. In *Proceedings of the 8th International Conference on
251 Computer Supported Education, CSEDU 2016*, pages 383–390. SCITEPRESS -
252 Science and Technology Publications, Lda, Portugal, 2016.
- 253 [19] C. Roast, D. Patterson, and V. Hardman. Visualisation — it is not the data, it is
254 what you do with it. In P. Kommers and P. Isaías, editors, *e-Society 2018
255 Conference Proceedings*, pages 231–238. IADIS, 2018.
- 256 [20] F. Wibbelink. Interacting with conditionals in viskell. 2016.