xprocedit, A Browser-Based Open-Source XProc Editor

Marco Geue, Hochschule Merseburg <`marco.geue@stud.hs-merseburg.de`>

Gerrit Imsieke, le-tex publishing services GmbH <`gerrit.imsieke`>

Abstract

A visual XProc editor can serve at least two purposes: Communicating the process flow to non-programmers and easing the notoriously steep learning curve for programmers.

An implementation using the Javascript framework JointJS and the in-browser XSLT 3 processor Saxon-JS is demonstrated, along with the challenges of supporting XProc's peculiarities in a generic graph editing framework.

Table of Contents

Introduction

Visual programming languages have been around since the 1960s (see, for example, [Boshernitsan1998]). Visual programming often uses the dataflow programming paradigm, whose description XProc matches exemplarily:

> Applications are represented as a set of nodes (also called blocks) with input and/or output ports in them. These nodes can either be sources, sinks or processing blocks to the information flowing in the system. Nodes are connected by directed edges that define the flow of information between them. [BoldtSousa2012]

XProc is an "XML Pipeline Language" that was first specified in 2010 [XProc1]. Its main purpose is the orchestration of XML validation and transformation tasks that traditionally is done with scripting languages and tools such as Ant, Make, shell scripts, or custom Java programs. Two advantages of XProc are: 1. Processing happens in main memory, which means documents need not be serialized or even stored to disk between processing steps, and there are no penalties in JVM startup times that are often associated with shell script or Makefile orchestration. 2. There is no global state in an XProc pipeline, a thing that is responsible for unexpected side effects in Ant, Make or Shell scripts and that makes these programming languages unsuited for encapsulating possibly complex, re-usable functionality. XProc provides this encapsulation and composability in its processing blocks, the so-called "steps."

It has been argued that, given the advances in XPath 3.1 and XSLT 3.0, together with extension modules for dealing with binary data, archives, HTTP APIs, etc., XSLT itself can now be used as a replacement for XProc pipelines [Quin2019]. While this is possible, XSLT does not offer or enforce the degree of processing block encapsulation that XProc offers and that is a prerequisite for creating a visual editor that lets users assemble predefined building blocks.

While XProc version 1.0 is primarily focused on processing XML documents, XProc 3.0 [XProc3] knows other document types, namely text, HTML, and JSON, but also arbitrary binary files, as first-class citizens. XProc's main applications so far are in publishing, and publishing these days requires handling of non-XML HTML and JSON, in particular. XProc 3.0's embrace of XPath 3.1 as its expression language and the XSLT and XQuery serialization 3.1 specification [Serialization31] are key enablers of its more "webby" capabilities.

XProc 3 programs continue to be written in XML syntax, at least this is the only serialization that is specified for 3.0. When writing a browser-based editor, there probably needs to be some translation between the editor's representation of a pipeline, be it HTML, SVG, or JSON, and XProc's XML syntax. As it will be shown, the solution presented in this paper uses JSON data structures as the internal representation. Translation from XProc XML to this internal model and back to XProc XML will be done with XSLT 3.0 in the browser and the XPath 3.1 functions that convert between JSON and XML.

While the concepts of XProc seem straightforward at first glance – processing steps whose outputs connect with inputs of other steps – many users have reported difficulties in writing actual pipelines. This is partly due to the syntax. The concept of primary inputs and outputs and default readable ports (that is supposed to make pipeline documents less verbose) contributes to this reportedly steep learning curve.

Having experienced graph editors that also know the concept of multiple ports per processing unit, namely the Blender node editor [Blender] and the Lego Mindstorms visual programming environment, we at le-tex thought that connecting existing XProc steps graphically will help overcome the initial learning difficulties. It will also be useful to visualize complex XProc pipelines since the XML representation offers almost no visual clues (apart from adjacent primary output/input ports) which steps are connected and how data flows through the pipeline.

Why is XProc Special?

XProc pipelines are basically directed acyclic graphs where the processing steps are the nodes and the input/output connections are the edges. However, when we at le-tex first tried to use generic browser-based graph editing frameworks for XProc in 2014, we discovered that some fundamental XProc properties are not well supported, to wit: Multiple docking ports per node, the distinction between input ports and options, the distinction between parameter and document inputs, encapsulation/sub-graphs, and default readable ports.

Encapsulation, for example, is a powerful feature of XProc that allows to expose a potentially complex pipeline as an apparently monolithic building block with a well-defined interface and opaque innards. Some graph editing frameworks are able to fold a sub-graph so that it occupies less screen real estate. This is no genuine encapsulation though since it requires folded sub-graphs to be copied and pasted rather than re-used. It does not allow the "write once, use many times" approach that XProc's language design supports so well.

This kind of encapsulation can be seen in other functional languages, too, and there are graphical editors for other programming languages. What sets XProc apart is the ability of a processing step to produce many different outputs that don't need to be consumed at once (or at all). This is useful for example when an encapsulated multi-step conversion pipeline produces the conversion result on one port and intermediate results and validation reports for the input and output on other ports. But this multi-valued, non-simultaneously consumed outputs deviate sufficiently enough from common programming paradigms as to render visual editors for these languages unsuited for editing XProc.

Another peculiarity, XProc's concept of "primary" and "default readable ports" is meant to make pipeline authoring less verbose: The primary port of adjacent (in document order) steps connect implicitly, without the need to establish explicit connections. On the two-dimensional canvas of a visual pipeline editor, however, there is no canonical document order. Turning the 2-D representation into a linear XProc XML document, a task that the visual XProc editor needs to perform, will become an optimization problem where a score needs to be attached to multiple possible serializations, rewarding implicit connections via default readable ports. Alternatively, users of the graphical editor could be forced to make XML document order explicit, a thing that we wanted to avoid for usability reasons.

This means that although XProc seems to be a perfect candidate for visual or dataflow programming, its reliance on XML serialization is an extra challenge when converting the natural graph to an XML representation that actually helps people writing their first pipelines.

One should acknowledge that a visual XProc programming environment will probably never replace actual coding. To a large extent this is due to the amount of XSLT that many pipelines orchestrate. At least this is what our experience as developers of the transpect framework [transpect] tells us. In most of our pipelines and libraries, the core tasks will be performed by XSLT stylesheets. This is not a shortcoming of XProc but rather a feature. Apart from ripping apart XSLT micropipelines, we don't strive at replacing what we do in XSLT with XProc steps. We do see a huge benefit in continuing to use XSLT's template matching and import mechanisms while encapsulating multi-step XSLT, zip/unzip, HTTP request, validation, etc. pipelines in well-defined XProc step signatures with possibly multiple outputs.

Selecting a Graph Editing Framework

The criteria for selecting a graph editing framework were partly XProc-related, partly informed by current software development trends and user expectations:

- browser-based
- interactive
- customizable
- provides ports
- supports encapsulation (hierarchical graphs)
- open source

A market research of graph editing frameworks, excluding those that don't run in browsers, led to the following score matrix:

	GoJS	**JointJS**	**NoFlo/FloHub**	**vis.js**	**yFiles**
interactive	yes	yes	no/yes	yes	yes
customizable	yes	yes	yes	yes	yes
encapsulation	no	yes	yes	no	no
ports	no	yes	yes	no	yes
open source	no	yes	yes/partly	yes	no

The Javascript framework that was selected for xprocedit is JointJS [JointJS]. Although it needs to be said that XProc pipelines are, by far, not a graph type that is supported by JointJS out of the box, its basic support for graph editing, for storing a graph model and for rendering a model as SVG has been very helpful.

NoFlo was a contender, but JointJS was chosen after initial experiments were promising. NoFlo didn't enter the experimental phase, therefore we cannot say whether it would have required less customizing.

Auto-layout was not an initial requirement, but many of the frameworks support it and it will be put to use in particular after loading an existing XProc pipeline.

The choice of XSLT 3 and Saxon-JS for converting back and forth between the browser representation (be it SVG or Javascript/JSON) and XProc XML was never really disputed. This is in spite of Saxon-JS not meeting the open-source requirement that we imposed on the graph editing framework. The rationale for this is that we regard XSLT 3 in the browser as a strategic choice of technology, and we want to support the only existing implementation. Graph libraries for the Web, on the other hand, do too little for XProc-style graphs out of the box to justify an investment in a commercial graph library.

Solution

The solution, called xprocedit [xprocedit], consists of an extension to the JointJS shape model in order to accommodate different kinds of XProc steps (atomic, compound, multicontainer; standard/user-defined), primary/non-primary ports, and options that act like input ports but are displayed differently and can have default values.

More details are given in one of the author's master's thesis [GeueMT].

XProc 3.0 is currently in the later stages of being specified. One XProc 1.0 concept that 3.0 does away with is parameter ports. It was decided that parameter ports will not be supported by xprocedit. However, for pipelines that don't use parameter ports, an XProc 1.0 serialization is still available. This is because the generated pipelines need to be tested but XProc 3.0 processors are not widespread yet. Support for XDM 3.1 maps, which serve as a replacement for parameter ports when supplied as a parameter option, is currently being implemented in xprocedit. It should be possible, if users are interested and if funding is available, to use xprocedit as an XProc 1.0→3.0 migration tool, converting parameter ports to map options during import.

An issue where an interactive editor may help is validation, or rather, forcing the user to only create valid pipelines. Ideally, a graphical pipeline editor will not, for example, let users connect two steps with a directed edge that points from input to output, from input to input, or from a step to itself. Another check that is built into the tooling might prevent users from connecting an output port that may emit multiple documents to an input port that only accepts a single document. To that end, a certain amount of custom Javascript application logic had to be employed, and this is not finished yet. Verifying that the declared content types of an output port match the content types that an input port accepts is another future custom Javascript validation that is interesting in designing XProc 3.0 pipelines with their support for non-XML documents.

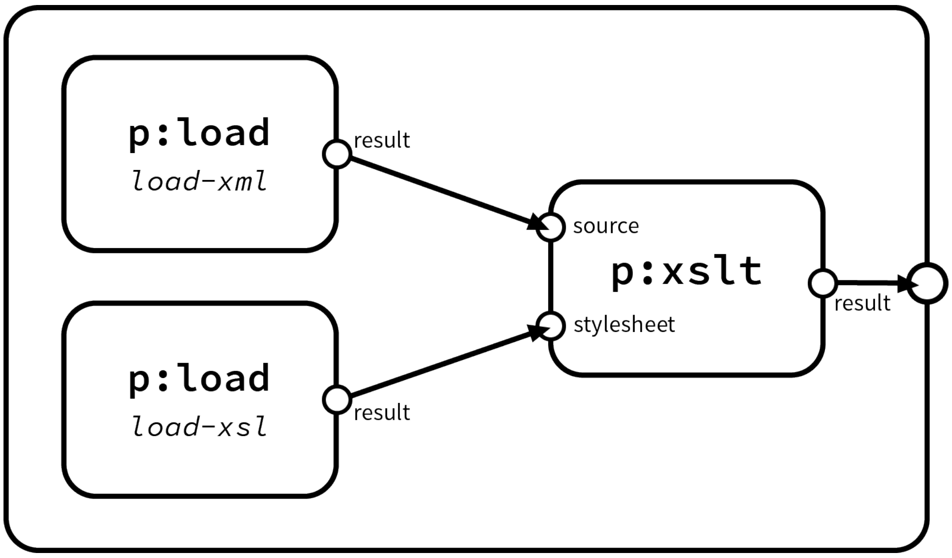
Some constraints, such as whether there are no loops in the pipeline or whether all required inputs are connected, are currently only checked upon export, via XSLT, and with poor error reporting.

As mentioned in the previous section, export from the internal Javascript model via its JSON serialization is performed by XSLT 3.0 transformations in the browser, starting from the result of applying `fn:json-to-xml()` to the JSON model.

The framework maintains an internal Javascript object representation of the graph. It was decided that the XProc XML document be generated from the JSON serialization of this model, rather than from the SVG rendering. The main reason is that, although the JSON representation also contains some layout information, it is considered as more stable than an SVG rendering. The tool that was chosen for generating the XProc XML document, Saxon-JS, supports XSLT 3 and XPath 3.1, and therefore it is equally capable of transforming JSON documents to XML as it is capable of transforming SVG to another XML vocabulary. What finally tipped the scale in favour of converting JSON rather than SVG to XProc was symmetry: There also needs to be an import process that imports pipelines and step libraries to the internal model, which is Javascript/JSON rather than SVG.

An important aspect when generating pipelines for future human editing in XML format is this: There can be multiple equivalent XML representations of a given graph. Consider the following pipeline (Figure 1, "A Simple XSLT Pipeline", taken from [GeueMT]):

Figure 1. A Simple XSLT Pipeline



When serializing this pipeline as an XProc XML document, two variants are possible:

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
  xmlns:c="http://www.w3.org/ns/xproc-step" version="1.0">
  <p:output port="result" primary="true"/>

  <p:load name="load-xsl" href="test.xsl"/>

  <p:load name="load-xml" href="test.xml"/>

  <p:xslt>
    <p:input port="stylesheet">
      <p:pipe port="result" step="load-xsl"/>
    </p:input>
    <p:input port="parameters"><p:empty/></p:input>
  </p:xslt>

</p:declare-step>
```

and

```
<p:declare-step xmlns:p="http://www.w3.org/ns/xproc"
  xmlns:c="http://www.w3.org/ns/xproc-step" version="1.0">
  <p:output port="result" primary="true"/>

  <p:load name="load-xml" href="test.xml"/>

  <p:load name="load-xsl" href="test.xsl"/>

  <p:sink/>

  <p:xslt>
    <p:input port="source">
      <p:pipe port="result" step="load-xml"/>
    </p:input>
    <p:input port="stylesheet">
      <p:pipe port="result" step="load-xsl"/>
    </p:input>
    <p:input port="parameters"><p:empty/></p:input>
  </p:xslt>

</p:declare-step>
```

The first variant uses a connection between the primary output of the step named `load-xml` with the primary input of the XSLT step, while the second variant serializes the `p:load` steps in reverse order and therefore needs to insert a `p:sink` step in between, and it needs to connect the primary input of the XSLT step, `source`, explicitly with the primary output, `result`, of `load-xml`.

Unless one wants to use the x-y-coordinates as an ordering hint, the graph editor does not provide clues about the preferred serialization order for the `p:load` steps. One possibility that was quickly rejected was to use another type of connectors in the 2-D graph that represent document order. The idea was rejected because it puts an additional burden onto the user that seems unnecessary.

Generating a serialization that makes maximum use of primary port connections is an optimization problem. The author addressed it by a graph traversal that favours serializing primary ports adjacently when there are multiple choices. This is a heuristics that does not guarantee optimal results but solves this issue well enough.

The import mechanism has to deal with a problem that is also related to default readable ports: It needs to make implicit connections explicit for ports and also for options. The core of this problem has already been addressed in an XSLT-based XProc documentation tool whose normalized output will be converted to JointJS's internal JSON graph model using Saxon-JS.

Sub-Graphs

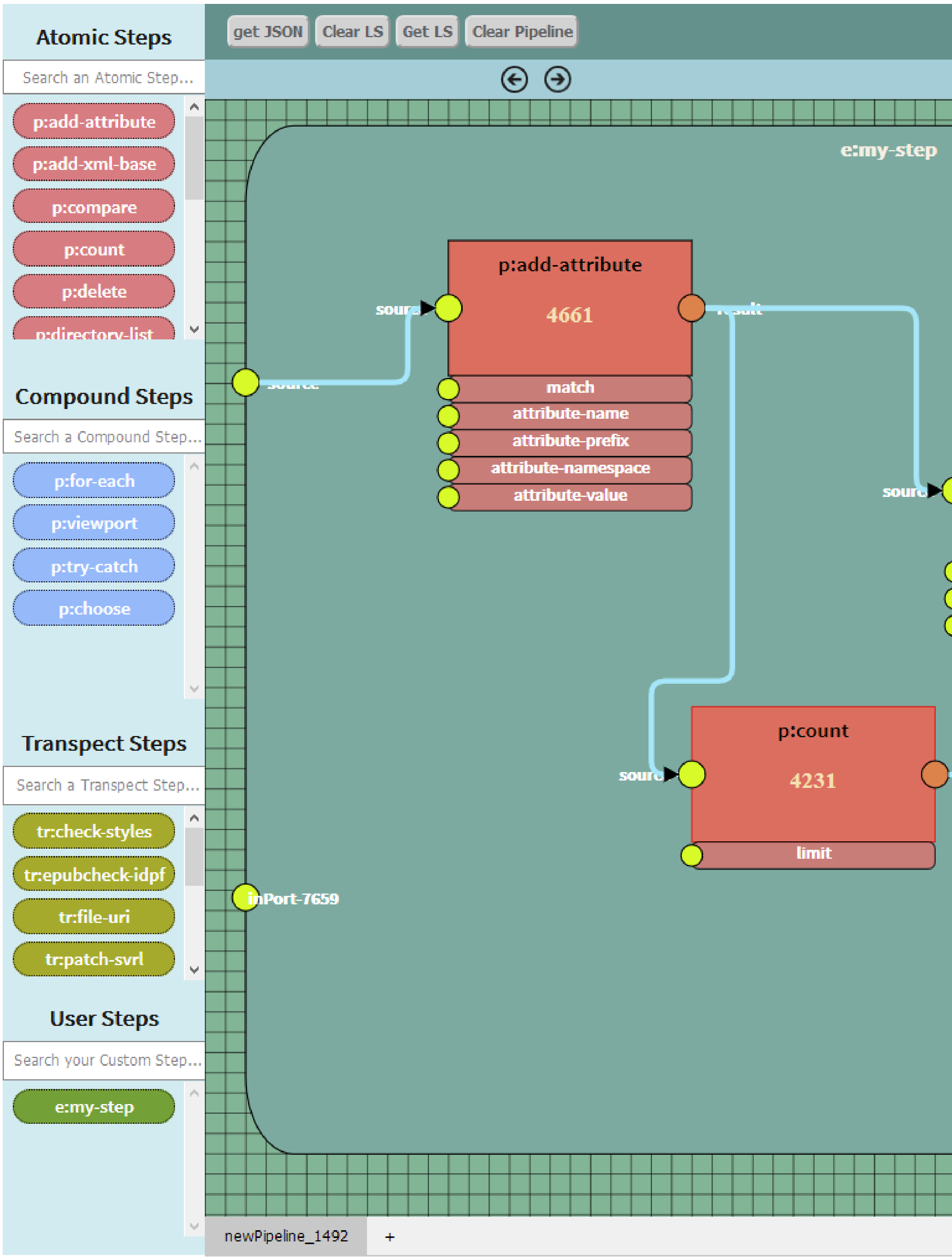
The subpipelines of compound steps such as `p:for-each` are displayed in their own tabs. Some bookkeeping in the Javascript application will make sure that they are included in the JSON representation upon export and that they are removed when the placeholder block in the containing graph is removed. JointJS would have permitted in-place folding of subpipelines, but expanding them would have quickly occupied much screen space and it would also have necessitated more advanced auto-layout capabilities.

Auto-layout is currently being added to the editor, its primary application being rendering the pipeline and its subpipelines (in separate tabs) initially after loading an existing pipeline.

For round-tripping (import, edit, export, import, …), in order to spare users the ordeal of recreating a decent layout in xprocedit over and over again, there is an option to preserve layout information in the serialized pipeline, using XProc `p:pipeinfo` elements.

The application's user interface is still under development; in March 2019 it looked like Figure 2, “User Interface”.

Figure 2. User Interface



The step library palettes on the left are still supplied statically as JSON structures. There will be an import process that processes `p:import` statements and makes available the imported steps, grouped by namespace prefix. A difficulty that has already been solved is that imports often (in case of transpect, at least) use canonical import URIs that are not identical with their locations on the Web server that xprocedit runs on. A catalog resolver written in XSLT that runs in the browser will perform the required URI translations when recursively resolving the imports, and it will also restore the canonical URIs when serializing the XProc XML from the internal representation.

Other Visual XProc Editors

Even before XProc 1.0 was finalized, EMC published the interactive XProc Designer [EMCXProcDesigner] that ran in the browser. It was built using the Google Web Toolkit [GWT]. The editor was visually appealing, but lacked an important feature: It was not possible to import other steps or step libraries. This and other features are mentioned in a feature "pipeline" but development seems to have stalled since many years.

Another recently developed visual XProc editor is GProc [GProc]. It is written in Python with a GTK + interface, therefore it does not run in the browser. Similar to xprocedit, it is the result of a master's thesis.

Outlook

xprocedit has been written as part of one of the author's master's thesis.Much effort has gone into adapting the Javascript graph framework for XProc, therefore, given the limited amount of time available, some crucial features such as pipeline and library import are not functional yet. Refactoring some of the user interface components, such as the option editor, can use some rework. It is conceivable to use Saxon-JS to a larger extent for generating these types of forms.

While pipelines are currently "stored" in main memory, we will probably add a RESTXQ service, provided by a BaseX database, to store the edited pipelines and to load step libraries from.

If there is interest and funding, XProc can become a native graph type in JointJS or its commercial derivative, Rappid [JointJS].

If there is interest not only in editing pipelines in the browser but also in executing them in the browser, it is conceivable that, using Saxon-JS and interfacing other Javascript libraries, a subset of XProc 3.0 will be made available in the browser at some stage, probably as part of another master's thesis.

An intermediate solution would be to run an XProc processor on a server and to post pipelines (and payloads) from the editor to the server via HTTP. A specific appeal of this solution is that both nascent XProc 3.0 processors, Calabash and Morgana, will accept alternative pipeline serialization formats than XML. So xprocedit might be able to post its internal graph representation as RDF or as its native JSON model, without the need to do the default readable port optimization that is only meant to simplify further editing in XML format.

Conclusion

A prototype of a visual XProc editor has been presented. Although the design choice for the graph library has been vindicated, a considerable amount of effort was necessary and will be necessary in order to add XProc as another supported graph type to the library, JointJS. Given that the visual editor will never been a complete programming environment (since much project-specific code will be written in XSLT, XQuery, schema languages, etc.) and XProc is a niche language, it is not clear whether xprocedit will be developed further or whether it will experience the fate of XProc Designer, whose development stalled shortly after the first release. At least the code is open source and can be picked up and modified by anyone.

Bibliography

[Blender] Blender Documentation: Node Editor https://docs.blender.org/manual/en/latest/editors/node_editor/

[BoldtSousa2012] Boldt Sousa, Tiago, (2012). Dataflow Programming: Concept, Languages and Applications https://paginas.fe.up.pt/~prodei/dsie12/papers/paper_17.pdf

[Boshernitsan1998] Boshernitsan, Marat, Downes, Michael (1998). Visual Programming Languages: A Survey https://www2.eecs.berkeley.edu/Pubs/TechRpts/2004/6201.html

[EMCXProcDesigner] EMC XProc Designer https://community.emc.com/docs/DOC-4382

[GeueMT] (2019). Entwicklung eines grafischen Editors für XProc-Pipelines mit dem SVG-basierten JavaScript-Framework JointJS Master’s Thesis, Hochschule Merseburg

[GProc] GProc – A Graphical Authoring Tool for XML Pipelines https://gitlab.com/in2erval/gproc/

[GWT] Google Web Toolkit (GWT) https://opensource.google.com/projects/gwt

[JointJS] JointJS https://www.jointjs.com/

[Quin2019] (2019). XProc in XSLT: Why and Why Not. In: XML Prague 2019 Conference Proceedings, p. 255. http://archive.xmlprague.cz/2019/files/xmlprague-2019-proceedings.pdf

[Serialization31] XSLT and XQuery Serialization 3.1, W3C Recommendation, 21 March 2017 https://www.w3.org/TR/2010/REC-xproc-20100511/

[transpect] transpect, An Open-Source Framework for Converting and Checking Data https://transpect.github.io/

[XProc1] XProc: An XML Pipeline Language, W3C Recommendation, 11 May 2010 https://www.w3.org/TR/xslt-xquery-serialization-31/

[XProc3] XProc 3.0: A Pipeline Language http://spec.xproc.org/

[xprocedit] xprocedit https://github.com/mag-letex/xprocedit/