- Entwicklung eines grafischen Editors für XProc-Pipelines mit dem SVG-basierten JavaScript-Framework JointJS -

---

MASTERTHESIS

---

zur Erlangung des Akademischen Grads
Master of Arts

vorgelegt von	Marco Geue Studiengang Master Informationsdesign und Medienmanagement
am	19. März 2019
Erstgutachter:	Dr. rer. nat. Thomas Meinike (HS Merseburg)
Zweitgutachter:	Dipl.-Phys. Gerrit Imsieke (le-tex)
Praxispartner:	le-tex publishing services GmbH
eingereicht im	Fachbereich Wirtschaftswissenschaften und Informationswissenschaften Hochschule Merseburg

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Abschlussarbeit mit dem Titel *Entwicklung eines grafischen Editors für XProc-Pipelines mit dem SVG-basierten JavaScript-Framework JointJS* in allen Teilen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel (einschließlich elektronischer Medien und Online-Quellen) benutzt habe. Die Arbeit wurde an keiner anderen Hochschule und in keinem anderen Studiengang als Prüfungsleistung eingereicht.

Alle wörtlich oder sinngemäß übernommenen Textstellen sind als solche kenntlich gemacht.

Leipzig, den 19. März 2019,

Marco Geue

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcode-Verzeichnis

Abkürzungsverzeichnis

CSS	Cascading Style Sheets
DRP	Default Readable Port
EPUB	Electronic Publication
GUI	Graphical User Interface
HTML	Hypertext Markup Language
JSON	JavaScript Object Notation
SVG	Scalable Vector Graphics
UI	User Interface
URI	Uniform Resource Identifier
XML	Extensible Markup Language
XProc	XML Processing
XSLT	Extensible Stylesheet Language Transformations

1 Einleitung

Bei Datenkonvertierungs- und Validierungs-Workflows, wie sie insbesondere im Publikationswesen vorkommen, ist die sogenannte „XML Pipeline Language“[1] XProc ein hilfreiches Werkzeug zur Erfassung und Verarbeitung komplexer Datensätze und verschiedener Dokumentenformate. Die Vorteile, die sich durch die Nutzung von XProc gegenüber anderen Konvertierungsabläufen ergeben, werden dabei von vielen unterschätzt. Mit XProc lassen sich eine Vielzahl von Verarbeitungsschritten[2] in einem einzigen Schritt bzw. einer Pipeline zusammenfassen. Die einzelnen Schritte können von Nutzer*innen präzise an eigene Erfordernisse angepasst werden. Das modulare System erweiterbarer Step-Libraries bietet zudem die Möglichkeit, selbst generierte Verarbeitungsschritte in neuen Kontexten wiederzuverwenden, sie in komplexe Konvertierungen einzubetten und erprobte Pipelines anderer Entwickler*innen zu nutzen. Der anfangs höhere Entwicklungsaufwand einer XProc-Konvertierung kann sich für Firmen und Entwickler*innen so in später stark vereinfachten, wiederverwendbaren und dynamisch anpassbaren Verarbeitungsschritten rechnen.

Mehrere Firmen arbeiten bereits seit Jahren erfolgreich mit XProc, jedoch bleibt ihre Anzahl, wie auch die Zahl aktiver XProc-Entwickler*innen vergleichsweise gering und etwa konstant. (Vgl. Siegel 2019: 17) Firmen mag die Einbeziehung von XProc in vorhandene Arbeitsabläufe oder gar der komplette Umstieg auf XProc-Konvertierungen als zu umständlich und risikoreich anmuten. Entwickler*innen scheint die komplexe und mitunter tief verschachtelte Struktur von XProc nach kurzer Zeit abzuschrecken. Und selbst unter XML-/ XSLT- Entwickler*innen ist „sehr kompliziert“ ein gern gewähltes Adjektiv zur Beschreibung von XProc.

Dabei wäre eine größere XProc-Community durchaus wünschenswert. Die aktiven XProc-Nutzer*innen zeigen bereits seit Jahren, wie bereichernd der fachliche Austausch sein kann, und wie viel Potenzial in der gemeinsamen Weiterentwicklung von XProc steckt. Sie berichten außerdem von einer steilen Lernkurve bei einer intensiven Beschäftigung mit XProc.[3] Auf Tagungen und im alltäglichen Austausch durch Fachartikel, Forenbeiträge oder (Online-)Arbeitsgruppen wird XProc diskutiert und stetig verbessert. In Kürze erscheint zudem XProc 3.0 mit neuen Funktionen und einer überarbeiteten Basis-Spezifikation.[4] Zudem gibt es mit *DAISY Pipeline*[5] und *transpect*[6] bereits zwei umfangreiche

1 Vgl. [W3C – 2010 – XProc]
2 z. B. Datensammlung, Validierung an einem Schema, Erstellung von XSL-Stylesheets, Konvertierung in ein neues Format etc.
3 Vgl. [GitHub – XProc Workshop]
4 Vgl. [XProc – XProc 3.0]
5 Vgl. [DAISY Pipeline]
6 Vgl. [Transpect]

Open Source-Frameworks, die Entwickler*innen über importierbare XProc-Pipelines vielfältige Konvertierungsmöglichkeiten bieten. Mit *Calabash*[7] und *MorganaXProc*[8] existieren darüber hinaus zwei voll funktionsfähige Prozessoren, die Pipelines verarbeiten, mit Input versorgen und Output produzieren.

Dennoch bleibt XProc aktuell eine XML-Technologie, die nur von wenigen spezialisierten Unternehmen genutzt und weiterentwickelt wird. Im Feld der Programmierung und Entwicklung offenbart sich hierbei eine Kluft: Auf der einen Seite die Entwickler*innen, die XProc aufgrund von dessen Leistungsfähigkeit und der Einbeziehung quasi aller XML-Technologien als eine Art XML-König*in ansehen. Und auf der anderen Seite jene, die von XProc entweder noch nie etwas gehört haben, oder es als Nischentechnologie belächeln, die aufgrund ihrer Komplexität oder der Fokussierung auf XML-Technologien für eine breite Anwender*innenschaft niemals zugänglich sein wird.
Auch um diese beiden Lager ein Stück weit zusammenzuführen, liegt die Idee, XProc besser verständlich zu machen und Einstiegshürden abzubauen, nahe. Um XProc zu erlernen oder zu verstehen, gibt es aktuell eine handvoll deutsch- und englischsprachiger Online-Dokumentationen und Tutorials. Hervorzuheben sind dabei eine umfangreiche XProc-Einführung von *data2Type*[9] und das „XProc Tutorial“ von Roger Costello[10]. Seit April 2018 existiert zudem ein erstes Print-Kapitel in einem Nachschlagewerk zu XML-Technologien. (Vgl. Grupe 2018) Im Jahr 2019 soll mit *XProc 3.0 Programmer Reference* von Erik Siegel (vgl. 2019) zudem ein umfangreiches Buch mit Fokus auf der neuesten XProc-Version veröffentlicht werden.
Um die Funktionalität von XProc leichter verständlich zu machen, wird in den Dokumentationen und Einführungen teilweise mit Schaubildern gearbeitet. Besonders häufig wird dabei das Konzept der Kapselung von XProc-Pipelines veranschaulicht, das für viele schwer verständlich ist. Den inneren Aufbau und die Regeln der Verknüpfungen innerhalb eines Steps nachvollziehen zu können, ist elementar, um den Weg der Verarbeitung eines Input-Dokuments innerhalb einer Pipeline zu erfassen und einen solchen Weg selbst beschreiben zu können.

In der XProc-Community hat, neben anderen Akteuren, der Praxispartner dieser Arbeit seit einigen Jahren die Idee, einen Editor zu entwickeln, mit dem sich XProc grafisch darstellen lässt und mit dem sich XProc-Pipelines anhand grafischer Modelle interaktiv zusammenstellen lassen. Ein solcher Editor könnte, im Kontext der bisherigen

7 Vgl. [XML Calabash]
8 Vgl. [MorganaXProc]
9 Vgl. [Data2Type – XProc]
10 Vgl. [Xfront – XProc Tutorial]

Erläuterungen, gleich mehrere Funktionen erfüllen: Die Visualisierung von XProc könnte Interessenten mit und ohne Programmierfähigkeiten oder informationstechnisches Know-How das Verständnis von XProc erleichtern und einen Einstieg in die Entwicklung mit XProc attraktiver machen. Ein funktionierender XProc-Editor, der zugleich in der Lage wäre, Code-basierte XProc-Pipelines zu visualisieren, sie grafisch zu bearbeiten und komplett neue Pipelines zusammenzustellen, könnte darüber hinaus für Firmen und XProc-Entwickler*innen von großem Interesse sein. Firmeninterne Arbeitsabläufe und Produktionsprozesse könnten Kunden transparenter und verständlicher vermittelt werden. Zudem könnten die Abhängigkeiten und Kapselungen einer XProc-Pipeline, durch eine Visualisierung besser nachvollzogen werden, was z. B. auch die Fehlerprüfung unterstützen würde. Außerdem würde das Prototyping von Pipelines deutlich einfacher und intuitiver und der firmeninterne Austausch über geplante Konvertierungsabläufe würde erleichtert.

Erste Versuche, einen solchen Editor zu entwickeln, gab es bereits: Im Juli 2009 wurde eine erste Version des EMC XProc Designer veröffentlicht.[11] Diese Web-Anwendung basiert auf dem Google Web Toolkit[12] und ermöglicht es, XProc-Pipelines per Drag & Drop zusammenzustellen und sie zu validieren. Der Import externer Step-Libraries ist hier nicht möglich. Die Entwicklung wurde mittlerweile eingestellt.
Anfang 2015 hat der Praxispartner dieser Arbeit einen Versuch mit dem Framework The Graph Editor durchgeführt, das heute Teil der FlowHub Entwicklungsumgebung ist.[13] Da sich das zentrale XProc-Konzept der hierarchischen Kapselung, also der Verschachtelung mehrerer Pipelines ineinander, nicht abbilden ließ und zudem der Entwicklungsaufwand mit den zusätzlich benötigten Technologien und Frameworks bereits im Anfangsstadium äußerst hoch war, wurde die Entwicklung jedoch beendet. Eine erneute Begutachtung von Frameworks und Anwendungen, die möglicherweise in der Lage wären, einen XProc-Editor zu realisieren, hat im April 2018 dann zum Start eines neuen Versuchs geführt. Dieses Mal wurde das JavaScript-Framework JointJS als Basistechnologie ausgewählt, das die notwendige Grundstruktur für die Entwicklung eines XProc-Editors mitzubringen schien. Im Rahmen eines Studienpraktikums und der anschließenden, hier vorliegenden Abschlussarbeit wurde aus vielen Vorüberlegungen ein erster, im Webbrowser lauffähiger Entwurf erstellt, der in mehrmonatiger Entwicklungsarbeit immer wieder verändert und erweitert wurde.

---

11 Die letzte Version des EMC XProc Designer kann hier abgerufen werden: [DellEMC – XProc Designer]

12 Vgl. [GWT Project]

13 Vgl. [GitHub – The Graph]

Die vorliegende Arbeit dokumentiert den beschriebenen Entscheidungs- und Entwicklungsprozess und erörtert die folgende zentrale Fragestellung: Ist es unter Verwendung des SVG-basierten JavaScript-Framework JointJS möglich, das Problem der Erstellung eines grafischen XProc-Editors - zu lösen?

Zur Beantwortung dieser Frage wird im folgenden zweiten Kapitel „Theorie, Problemanalyse und Zielformulierung“ zunächst genauer auf XProc, dessen Struktur, grundlegende Elemente und einige Besonderheiten eingegangen. Im zweiten Teil des zweiten Kapitels wird auf mathematische Grundlagen der Graphentheorie eingegangen. Dies bildet die Basis zum Verständnis der Struktur von XProc-Pipelines und ist eine Annäherung an die Frage, wie sich XProc visualisieren lässt. Im dritten Teil des zweiten Kapitels werden die wichtigsten Anforderungen zusammengefasst, die ein grafisches Framework erfüllen muss, um XProc visualisieren zu können. Es werden anschließend verschiedene Frameworks vorgestellt, die für die Umsetzung des XProc-Editors infrage kamen. Schließlich wird begründet, warum die Entscheidung auf JointJS fiel.
Das dritte Kapitel dieser Arbeit gibt eine Einführung in JointJS. Hierbei wird auf den groben Aufbau des Frameworks und seine Abhängigkeiten eingegangen. Mit Ausblick auf die Entwicklung des XProc-Editors wird auf wesentliche Elemente, deren Datenstruktur und die Funktionsweise der grafischen Implementierung eingegangen.
Im vierten Kapitel „Entwicklung des XProc-Editors“ wird anfangs kurz auf den Entstehungsprozess, und die ersten Entwürfe des Editors eingegangen. Der Fokus liegt dann auf der Erläuterung des aktuellen Entwicklungsstands. Hierbei wird auf die Grundkonstruktion des Editors, sowohl im Quellcode, als auch in der visuellen Darstellung eingegangen. Die (geplante) Funktionsweise der XProc-Konvertierung wird erläutert. Und es wird erklärt, welche Zielgruppe und welche Endgeräte sowohl bei der Gestaltung, als auch bezüglich der technischen Funktionalität, berücksichtigt wurden. Eine praktische Einführung in die Bedienung der aktuellen Version des XProc-Editors beendet dieses Kapitel.
Im fünften Kapitel werden mit Rückblick auf die zentrale Frage dieser Arbeit die Forschungsergebnisse zusammengefasst. Der Entwicklungsstand des Editors wird erörtert und im Forschungskontext wird darauf eingegangen, inwiefern die Entwicklung eines XProc-Editors mit JointJS sinnvoll erscheint.

Zwar wird in der vorliegenden Arbeit die Kenntnis zentraler Technologien vorausgesetzt, in einem Glossar am Ende der Arbeit werden jedoch einige Grundlagen kurz erklärt. Da in der Spezifikation von JointJS einige Begriffe vorkommen, die sich mit in dieser Arbeit häufig vorkommenden Begriffen decken, obwohl die Bedeutung nicht identisch ist, sind im Fließtext alle Begriffe, die unmittelbar zu JointJS gehören zur leichteren

Orientierung blau hinterlegt.
Neben den Literaturangaben im Fließtext sind an vielen Stellen Fußnoten mit Verweisen zu Internet-Quellen gesetzt. Die Verweise sind chiffriert und mit Hyperlinks versehen. Die aufgeschlüsselten Verweise mit den kompletten URLs finden sich im Literaturverzeichnis.

2 Theorie, Problemanalyse und Zielformulierung

XProc hat eine Struktur, die anhand grafischer Abstraktion entstanden ist und anhand dieser stetig weiterentwickelt wird. Somit lässt sich XProc auch mit graphentheoretischen Begriffen und Modellen erklären und skizzieren. Eine grafische Skizzierung von XProc ist darüber hinaus die essenzielle Grundlage für die Entwicklung eines XProc-Editors. Nur so können die Anforderungen, die ein Visualisierungs-Framework haben muss, genau festgelegt werden. Um also die Frage beantworten zu können, wie XProc aussieht bzw. was für eine Art Graph XProc ist, wird im Folgenden zunächst auf die Code-basierte Struktur von XProc eingegangen. Anschließend wird auf einige Grundlagen der Graphentheorie eingegangen, um die graphentheoretische Struktur von XProc zu definieren. Anschließend wird anhand einer Feature-Matrix bewertet, welche Frameworks für die Abbildung der speziellen Eigenschaften eines XProc-Graph infrage kommen, und die Entscheidung für JointJS wird begründet.

2.1 XProc

Die „XML Pipeline Language“[14] XProc wurde entwickelt, um XML-Daten zu verarbeiten, einzelne Verarbeitungsschritte strukturiert zu beschreiben und einen geordneten Ablaufplan für die kollektive Verarbeitung mehrerer Schritte zu definieren. Der Name XProc leitet sich passend von „XML Processing“ ab.[15] Erik Siegel (2019: 16) schreibt, dass Ideen für eine XML-verarbeitende Sprache schon seit den Anfängen von XML in den 1990er Jahren existieren. In der standardisierten Version 1.0, die von Norman Walsh, Alex Milowski und Henry Thompson veröffentlicht wurde, ist XProc seit Mai 2010 eine Empfehlung des *World Wide Web Consortium (W3C)*. Der erste offizielle Entwurf von XProc wurde im September 2006 veröffentlicht.[16]

XProc 1.0 umfasst eine komplexe XML-basierte Syntax aus Elementen und deren Attributen.[17] Die größten Gruppen der XProc-Elemente sind Pipelines bzw. Steps, Input- und Output-Ports, Options, Parameter und Variablen. Auf diese Gruppen wird in den folgenden Abschnitten genauer eingegangen. Zudem wird auf das Konzept der **Kapselung** von XProc-Steps eingegangen, das XProc gegenüber anderen XML-Sprachen auszeichnet und einen wichtigen Grund für die Komplexität von XProc darstellt.

Das XML-Dienstleistungsunternehmen *data2type* beschreibt XProc als „[…] eine in XML verfasste Sprache, welche einen Satz an Befehlen für eine Ablaufsteuerung zur Erzeugung von XML-orientierten Workflows bereitstellt. Ein XProc-Dokument wird von einem

14 Vgl. [W3C – 2010 – XProc]
15 Vgl. [XProc.org]
16 Vgl. [W3C – 2006 – XProc]
17 Vgl. [W3C – 2010 – XProc]

Prozessor gelesen, der diese Befehle dann sequentiell abarbeitet."[18] Mit einer validen XProc-Pipeline kann also beispielsweise das Auslesen von Datensätzen aus einem unterstützten Dokumentenformat (meistens XML), die Validierung der Daten an einem Schema, die Umstrukturierung und anschließende Übertragung der Daten in verschiedene Dokumentenformate, wie HTML oder EPUB, zusammengefasst und anschließend durch einen XProc-Prozessor verarbeitet werden. Das Schaubild in Abb. 1 zeigt einen (stark vereinfachten) exemplarischen XProc-Workflow für die Erstellung mehrerer Dokumente aus einer XML-Datei.

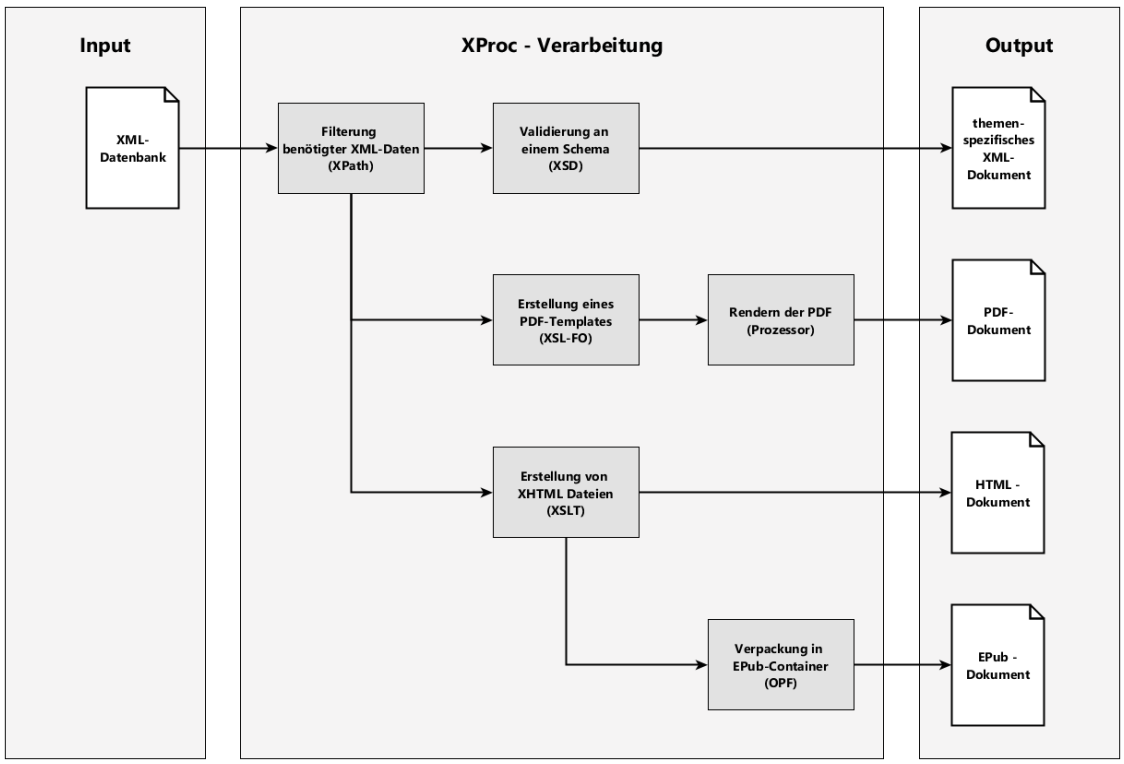


Abb. 1: Schema einer XProc-Konvertierung

Die Spezifikation von XProc wird von einem Kreis von Experten ständig weiterentwickelt. Die XProc-Version 3.0 soll in Kürze veröffentlicht werden.[19]

In der folgenden Beschreibung einiger XProc-Grundlagen stütze ich meine Ausführungen hauptsächlich auf die XProc-Spezifikation des *W3C* sowie eine deutschsprachige Online-Zusammenfassung von *data2type*. Darüber hinaus bezieht sich das Folgende fast ausschließlich auf XProc 1.0.

18 Vgl. [Data2Type – XProc]
19 Vgl. [XProc.org]

2.1.1 Pipelines und Steps

Wie alle wohlgeformten XML-Dokumente beinhaltet auch ein XProc-Dokument ein einziges Wurzelelement. In XProc 1.0 heißt dieses entweder `<p:declare-step>` oder `<p:pipeline>`. Jedes XProc-Dokument beschreibt somit jeweils eine Pipeline bzw. einen Step. Das vorangestellte `p:` ist das standardmäßige Präfix vordefinierter XProc-Steps, welches dem XProc-Namensraum mit der URI `http://www.w3.org/ns/xproc` zugeordnet ist. Dieser Namensraum beschreibt das gesamte XProc-Vokabular.[20] Auf die Bedeutung von Präfixen und Namensräumen wird in Abschnitt 2.1.3 noch genauer eingegangen.
Eine **Pipeline** ist in der W3C-Spezifikation definiert als „[...] set of connected steps, with outputs of one step flowing into inputs of another. A pipeline is itself a step and must satisfy the constraints on steps.“[21] In diesem Zitat wird die Komplexität von XProc deutlich: Eine Pipeline beinhaltet eine Reihe von Steps, die miteinander über Input- und Ouptput-Ports verbunden werden. Gleichzeitig ist die Pipeline selbst ein Step und muss somit die Anforderungen, die XProc an Steps hat, erfüllen. Die Begriffe ‚Pipeline‘ und ‚Step‘ werden auch in den folgenden Ausführungen häufig und im wechselnden Austausch verwendet. Deswegen sei an dieser Stelle darauf hingewiesen, dass es im Kern der Struktur von XProc liegt, diese Begriffe als nahezu synonym zu verwenden. Eine Pipeline definiert einen Step und ein Step ist dementsprechend eine Pipeline. In XProc 3.0 wird das Wurzelelement `<p:pipeline>` zur Vereinfachung der Struktur abgeschafft. (vgl. Siegel 2019: 35) Für das Verständnis von XProc sind dennoch beide Begriffe elementar. So kann eine Pipeline auch als Container verstanden werden, in dem Input-Dokumente von Step zu Step zu einem finalen Output weitergereicht und in jedem Step ein Stück verändert werden. Ein Step ist somit eher als abgeschlossene Einheit zu betrachten. Das Konzept der Steps ermöglicht dessen Wiederverwendung und den Austausch über Step-Libraries. (siehe Abschnitt 2.1.3)
Der Quellcode einer XProc-Pipeline ist in zwei Teile unterteilt. Im ersten Teil, der auch „Prolog“ genannt wird, werden die Input- und Output-Ports sowie Options und Parameter der Haupt-Pipeline beschrieben, also das äußere Erscheinungsbild einer Pipeline (vgl. Siegel 2019: 69). Im Prolog können auch Variablen deklariert werden, auf die später zugegriffen wird. Im sogenannten „Body“ folgt der Inhalt der Pipeline, der **Subpipeline** genannt wird. Das folgende Code-Beispiel zeigt den Aufbau einer simplen XProc-Pipeline:

```
<?xml version="1.0" encoding="UTF-8"?>
<p:declare-step
   xmlns:p="http://www.w3.org/ns/xproc"
   xmlns:c="http://www.w3.org/ns/xproc-step"
```

20 Vgl. [W3C – 2010 – XProc #Namespaces]
21 Vgl. [W3C – 2010 – XProc #PiplineConcepts]

```
   xmlns:my="me"
   version="1.0"
   name="my-pipeline"
   type="my:pipeline">

   <!-- PROLOG -->
   <p:input port="source" primary="true">
      <p:inline>
         <doc>Hello world!</doc>
      </p:inline>
   </p:input>
   <p:input port="insert" primary="false">
      <p:inline>
         <node>Text</node>
      </p:inline>
   </p:input>
   <p:output port="result"/>
   <p:option name="foo" select="'bar'"/>
   <!-- END OF PROLOG -->

   <!-- BODY -->
   <p:add-attribute match="/doc" attribute-name="foo">
      <p:with-option name="attribute-value" select="$foo"/>
   </p:add-attribute>

   <p:insert match="/doc" position="last-child">
      <p:input port="insertion">
         <p:pipe port="insert" step="my-pipeline"/>
      </p:input>
   </p:insert>
   <!-- END OF BODY -->
</p:declare-step>
```

Code-Beispiel 1 : Einfache XProc-Pipeline

Eine XProc-Pipeline wird bei der Erstellung in einem Editor als Datei mit der Endung `.xpl` abgespeichert. Der Oxygen XML Editor bietet derzeit die beste Unterstützung für die XProc-Syntax an.[22]

2.1.2 Step-Typen

Steps werden in drei Kategorien eingeteilt: Atomic Steps, Compound Steps und Multi-Container Steps. Zu diesen Kategorien gibt es eine Reihe von durch die XProc-Spezifikation vordefinierte Steps, bei denen mögliche Input- und Output-Ports sowie Options und Parameter bereits festgelegt sind.

Die meisten vordefinierten Steps gehören zu den **Atomic Steps**. Ein Atomic Step wird als ein Step definiert, der eine spezifische Einheit der XML-Verarbeitung darstellt, keine Subpipelines beinhaltet und trotz seiner kompakten Code-Größe große Rechenleistung

22 Vgl. [Oxygen – XProc]

benötigen kann.[23] Ein Beispiel für einen Atomic Step ist `<p:compare>`, der zwei XML-Dokumente auf ihre Gleichheit untersucht (vgl. Code-Beispiel 3 im nächsten Abschnitt). In der **Standard Step Library** von XProc werden Atomic Steps in Required- und Optional Steps unterteilt. **Required Steps** müssen von XProc-Prozessoren unbedingt unterstützt werden, um die Funktionalität einer Pipeline zu gewährleisten (siehe Abschnitt 2.1.6). **Optional Steps** können bei Bedarf eingebunden werden.[24]

Die Kategorie der **Compound Steps** ist sehr viel flexibler als die statischen Atomic Steps. Ein Compound Step ist in erster Linie als Step definiert, der eine Subpipeline enthält, in der eine unbegrenzte Anzahl weiterer Steps eingebunden werden kann.[25] Da Compound Steps als Container für andere Steps dienen, können z.B. zwei `<p:for-each>`-Steps sehr unterschiedlich aussehen. `<p:for-each>` gibt an den XProc-Prozessor den Auftrag, die diesem Step übergebenen Dokumente innerhalb der gegebenen Subpipeline als sich wiederholende Schleife zu verarbeiten. Ein weiteres Beispiel für einen Compound Step ist `<p:group>`, eine Art Standard-Compound Step, der hauptsächlich dazu dient, mehrere Steps zu einer neuen Pipeline zusammenzufassen.

Die letzte Kategorie, die **Multi-Container Steps**, stellen, wie die Compound Steps, Container für andere Steps und Subpipelines dar. Die Besonderheit eines Multi-Container Steps liegt darin, dass sich dessen Subpipelines als Alternativen gegenüberstehen, die je nach eintretendem Fall ausgeführt werden. Es gibt dabei nur zwei Multi-Container Steps: `<p:choose>` und `<p:try>`. `<p:choose>` kann beliebig viele Subpipelines beinhalten. Diese stehen sich innerhalb eines `<p:when>` gegenüber. Wenn die Bedingungen aller `<p:when>` scheitern, greift `<p:otherwise>` als letzte Option.

`<p:try>` beinhaltet genau zwei alternative Subpipelines und ermöglicht die Fehlerprüfung kritischer Verarbeitungsschritte vor ihrer Ausführung, um einen Programmabbruch zu verhindern (siehe Code-Beispiel 2). Die Versuchspipeline wird in einem `<p:group>` zusammengefasst und in `<p:try>` ausgeführt. Wenn die Ausführung fehlschlägt, wird der Inhalt von `<p:catch>` geladen.

```
<p:try
  name? = NCName>
  (p:variable*,
  p:group,
  p:catch)
</p:try>
```

Code-Beispiel 2 : Struktur von <p:try> [W3C – 2010 – XProc #pTry]

23 Vgl. [W3C – 2010 – XProc #AtomicSteps]
24 Vgl. [W3C – 2010 – XProc #StandardStepLibrary]
25 Vgl. [W3C – 2010 – XProc #ComoundStep]

2.1.3 Eigene Steps und Step-Libraries

> "Programming XProc is defining steps."
> (Siegel 2019: 44)

Wie schon mehrfach deutlich wurde, können neben den vordefinierten Steps eigene, wiederverwendbare Steps von Nutzer*innen erstellt werden. Dies passiert an sich automatisch, wenn eine neue XProc-Pipeline erstellt wird. Die zusätzliche Arbeit ist nun die Wiederverwendbarkeit dieser Pipeline in anderen Kontexten zu ermöglichen.
Hierzu muss der Pipeline zunächst ein eigener Namensraum und ein eigenes Präfix zugeordnet werden. Das Standard-Präfix `p:` ist den vordefinierten XProc-Steps vorbehalten. Und auch die Präfixe `c:` (reserviert für Parameter und Eingangsgrößen) und `err:` (reserviert für Fehlermeldungen) sollten nicht verwendet werden. Außerdem sollte darauf geachtet werden, dass sich die Präfixe nicht mit anderen importierten Steps, z. B. aus dem transpect-Framework (Präfix: `tr:`) decken.
Die Wiederverwendung eines Steps kann entweder innerhalb einer Pipeline passieren, indem der Step zu Beginn deklariert und später referenziert wird. Für eine externe Verwendung der eigenen Steps bietet es sich an, selbst erstellte Steps in einer eigenen Library abzulegen.
Eine XProc-Library macht es möglich, Steps (z. B. zu ähnlichen Konvertierungsschritten) zu sammeln und sie anderen Pipelines zur Verfügung zu stellen. Um eine Library zu erstellen, werden die Steps in ein eigenes XProc-Dokument kopiert. Das Wurzelelement ist hier ausnahmsweise `<p:library>`.[26] Um nun einen Step aus dieser Library zu importieren, wird innerhalb der Pipeline, in die der Step geladen werden soll, das Element `<p:import>` mit dem dazugehörigen Link-Attribut (z. B. `href` = „library.xpl“) aufgerufen.[27]
Im vorangegangenen Kapitel wurde bereits eine Library erwähnt, ohne dass darauf explizit hingewiesen wurde: Die vordefinierten Atomic Steps werden in der **Standard Step Library** von XProc zusammengefasst. Diese Library ist in XProc-Prozessoren automatisch implementiert und braucht nicht importiert werden. Auf der Internetseite der XProc-Spezifikation kann sie dennoch als XPL-Datei heruntergeladen werden. Das folgende Code-Beispiel zeigt einen kleinen Ausschnitt der Standard Step Library:

```
<?xml version="1.0" encoding="UTF-8"?>
<p:library xmlns:p="http://www.w3.org/ns/xproc" version="1.0">
   <p:declare-step type="p:add-attribute" xml:id="add-attribute">
      <p:input port="source"/>
```

26 Vgl. [W3C – 2010 – XProc #pLibrary]
27 Vgl. [W3C – 2010 – XProc #pImport]

```
      <p:output port="result"/>
      <p:option name="match" required="true"/>
      <p:option name="attribute-name" required="true"/>
      <p:option name="attribute-prefix"/>
      <p:option name="attribute-namespace"/>
      <p:option name="attribute-value" required="true"/>
   </p:declare-step>
   <p:declare-step type="p:add-xml-base" xml:id="add-xml-base">
      <p:input port="source"/>
      <p:output port="result"/>
      <p:option name="all" select="'false'"/>
      <p:option name="relative" select="'true'"/>
   </p:declare-step>
   <p:declare-step type="p:compare" xml:id="compare">
      <p:input port="source" primary="true"/>
      <p:input port="alternate"/>
      <p:output port="result" primary="false"/>
      <p:option name="fail-if-not-equal" select="'false'"/>
   </p:declare-step>

   <!-- [...] MISSING CODE [...] -->

   <p:declare-step type="p:xquery" xml:id="xquery">
      <p:input port="source" sequence="true" primary="true"/>
      <p:input port="query"/>
      <p:input port="parameters" kind="parameter"/>
      <p:output port="result" sequence="true"/>
   </p:declare-step>
   <p:declare-step type="p:xsl-formatter" xml:id="xsl-formatter">
      <p:input port="source"/>
      <p:input port="parameters" kind="parameter"/>
      <p:output port="result" primary="false"/>
      <p:option name="href" required="true"/>
      <p:option name="content-type"/>
   </p:declare-step>
</p:library>
```

Code-Beispiel 3 : Ausschnitt aus der Standard Step Library

2.1.4 Input- und Output-Ports

Wie bereits mehrfach beschrieben, stehen die verschiedenen XProc-Steps über Input- und Output-Ports miteinander in Verbindung. In der XProc-Spezifikation steht hierzu: „A typical step has zero or more inputs, from which it receives XML documents to process zero or more outputs, to which it sends XML document results [...].“[28] Die Input- und Output-Ports sind also Knotenpunkte einer Pipeline, an denen die zu verarbeitenden Daten ankommen und von denen aus sie an Subpipelines bzw. deren Ports weitergegeben werden. Jedem Step kann eine beliebige Menge an Input- oder Output-Ports zugewiesen werden. Obwohl manche XProc-Steps auf Daten, die nicht als XML formatiert sind, zugreifen und sie verarbeiten können, dürfen an den Ports selbst nur wohlgeformte XML-

28 [W3C – 2010 – XProc #StepConcept]

Dokumente ankommen oder weitergegeben werden.[29]
Ein Input-Port empfängt in der Regel nur ein Dokument. Wenn ihm jedoch das Attribut `sequence` mit dem Wert `true` gegeben wird, können ihm mehrere Dokumente zugeordnet werden. Gleiches gilt für Output-Ports.
Es gibt verschiedene Möglichkeiten, wie ein Input-Port seine Dokumente erhält: Externe Dokumente können über das Element `<p:document>` durch Angabe einer URI an einen Input-Port gebunden werden.[30] In Code-Beispiel 1 sind zudem zwei weitere Möglichkeiten zu entdecken: Ein Dokument kann mit dem Element `<p:inline>` direkt im entsprechenden Input-Port erzeugt werden. Außerdem kann der Input-Port (hier: „insertion“) seine Daten über das Element `<p:pipe>` explizit von einem vorherigen Port (hier: der Input-Port „insert“ der umgebenden Pipeline) erhalten.
Die meisten vordefinierten XProc-Steps verfügen zudem über sogenannte **Primary Ports**. In der XProc-Verarbeitung werden Primary In- und Output-Ports vom XProc-Prozessor automatisch miteinander verbunden, ohne dass die Verbindung explizit definiert sein muss.[31] Entscheidend für die Verknüpfung ist die **Dokumentenreihenfolge** bzw. Dokumentensequenz, also die Reihenfolge, in der die Steps im Quellcode der Pipeline aufgerufen werden. Der Primary-Output-Port des Vorgänger-Steps wird dabei zum **Default Readable Port** (DRP). Wenn kein Vorgänger-Step mit einem Primary-Output-Port existiert, könnte es noch eine automatische Verbindung mit einem Primary-Input-Port der umgebenden Pipeline geben, sofern ein solcher vorhanden ist. (vgl. Siegel 2019: 48) Grundsätzlich kann jeder Port zu einem Primary Port werden, indem ihm das Attribut `primary` mit dem Wert `true` zugeordnet wird.
Die Benennung von Ports ist bei selbst erstellten Pipelines dem User überlassen. Jedoch müssen die Namen von Ports innerhalb eines Steps einmalig sein. Der erste Input-Port eines Steps wird in der Regel mit ‚source‘ und der erste Output-Port mit ‚result‘ benannt. Dies sind bei vordefinierten Steps auch die häufigsten Primary-Ports.

2.1.5 Options und Parameter

Neben Input- und Output-Ports können Steps eine (beliebige) Anzahl an Options haben. Eine Option ordnet einem Step zusätzliche Informationen zu, die dessen Verhalten zusätzlich beeinflusst. Über das Attribut `select` kann einer Option z. B. anhand eines XPath-Ausdrucks ein spezifischer Pfad zugewiesen werden.

---

29 Vgl. [W3C – 2010 – XProc #InputsOutputs]
30 Vgl. [Data2Type – XProc #PortBinding]
31 Vgl. [W3C – 2010 – XProc #PrimaryInputOutput]

Jeder Option wird über das Attribut `name` ein eindeutiger und innerhalb eines Steps einmaliger Name gegeben. Zusätzlich wird bestimmt, ob für die Option zwingend ein Wert eingetragen werden muss (Attribut `required = true/ false`). Wenn die Angabe freiwillig ist, wird meistens der default-Wert des `select`-Attibuts verwendet.[32]
Options werden entweder mit dem Element <p:option> im Prolog eines `<p:declare-step>` deklariert oder mit `<p:with-option>` einem Step zugeordnet. In Code-Beispiel 1 sind beide Möglichkeiten zu finden.
Auch Options verfügen über einen Input-Port, der jedoch in der Regel nicht explizit beschrieben wird. Ohne weitere Angaben beziehen die Options ihren Input vom Primary-Input-Port der umgebenden Pipeline. Über ein `<p:pipe>`-Element kann jedoch auch ein anderer zuvor aufgerufener Port verbunden werden. Das folgende Code-Beispiel zeigt eine solche explizite Verknüpfung:

```
<p:add-attribute match="/doc" attribute-name="foo">
  <p:with-option name="attribute-value" select="$foo">
    <p:pipe port="source" step="my-pipeline"/>
  </p:with-option>
</p:add-attribute>
```

Code-Beispiel 4 : Verknüpfung über ein <p:pipe> - Element

Eine weitere Möglichkeit, um Steps zusätzliche Informationen zuzuweisen, ist über Parameter. Einem Parameter wird, wie bei einer Option, ein Name und ein Wert zugeordnet. Bei den vordefinierten Steps gibt es manche Steps, die Parameter akzeptieren bzw. benötigen. Die Generierung eines Parameters beginnt in einer Pipeline mit einem Input-Port des Typs `parameter`, der die benötigten Daten empfängt. Aus diesem Input wird anschließend ein XProc-Parameter mit dem `c:`-Präfix erzeugt. Dies entspricht der folgenden Syntax:

```
<c:param
  name = QName
  namespace? = anyURI
  value = string />
```

Code-Beispiel 5 : Syntax von XProc-Parametern [W3C – 2010 – XProc #cParam]

Dieser Parameter kann nun durch den XProc-Prozessor verarbeitet werden. Parameter werden insbesondere in der XSLT-Transformation innerhalb XProc verwendet.[33]

32 Vgl. [W3C – 2010 – XProc #pOption]
33 Vgl. [XProcBook – 2010]

2.1.6 Prozessoren und Fehlermeldungen

In den vorangegangenen Ausführungen wurde bereits ein paar Mal der XProc-Prozessor erwähnt. Diese zentrale Verarbeitungsinstanz prozessiert die XProc-Pipeline zusammen mit ihren Eingangsgrößen (Input-Dokumente, Options, Parameter etc.). Die derzeit verwendeten XProc-Prozessoren sind der von Norman Walsh entwickelte Prozessor XML Calabash[34] und der Prozessor MorganaXProc[35], welcher von Achim Berndzen entwickelt wurde. Beide Prozessoren wurden in Java geschrieben und ermöglichen eine Verarbeitung von XProc auf Basis der Kommandozeile. MorganaXProc bietet zusätzlich eine grafische Benutzeroberfläche an.

Jede XProc-Pipeline hat eine Art primären Pfad, der vom XProc-Prozessor bevorzugt verarbeitet wird. Dieser Pfad ergibt sich aus der Dokumentenreihenfolge und den DRPs. Neben dem primären Pfad kann es mehrere parallel zu verarbeitende Step-Abfolgen geben. Bei der Verbindung von Ports gibt es einige Regeln, die unbedingt eingehalten werden müssen, damit eine Pipeline funktioniert und keine Fehler produziert. Da XProc als DAG strukturiert ist, darf der Output-Port eines Steps nicht mit einem der Input-Ports desselben Steps verbunden werden. Ein Output-Port darf außerdem nicht mit dem Output-Port eines anderen Steps verbunden werden, außer es ist der Output-Port der umgebenden Pipeline. Genauso darf ein Input-Port nicht mit einem Input-Port verbunden werden, außer es ist der Input-Port der umgebenden Pipeline. Aufgrund der Dokumentenreihenfolge darf ein Output-Port außerdem nicht mit dem Input-Port eines vorherigen Steps verbunden werden.

Ein wichtiger und umfangreicher Bereich der XProc-Spezifikation ist die Definition von statischen und dynamischen **Fehlermeldungen**, die z. B. ausgegeben werden, wenn ein Primary-Output-Port nicht verbunden ist (`err:XS0006`) oder wenn zwei Ports eines Steps derselbe Name zugewiesen wurde (`err:XS0011`).[36] Eine weitere Fehlermeldung, die im Zuge der Definition von XProc als DAG bereits aufgetaucht ist, steht in der XProc-Spezifikation als `err:XS0001` an erster Stelle. Dort heißt es: „It is a static error if there are any loops in the connections between steps: no step can be connected to itself nor can there be any sequence of connections through other steps that leads back to itself.“[37]

Für die XProc-Prozessoren und das Debugging von Pipelines sind diese Informationen der Fehlermeldungen somit elementar.

---

34 Vgl. [XML Calabash]
35 Vgl. [MorganaXProc]
36 Vgl. [W3C – 2010 – XProc #Errors]
37 Vgl. [W3C – 2010 – XProc #ErrS0001]

2.2 Graphentheorie

> „Graphen sind mathematische Modelle für netzartige Strukturen in Natur und Technik."
> (Tittmann 2011: 11)

Ausgehend vom berühmten Königsberger Brückenproblem (vgl. Nitzsche 2009: 19f.), das 1736 von Leonard Euler bearbeitet wurde, hat sich eine komplexe Graphentheorie entwickelt, die heute ein Kerngebiet der diskreten Mathematik darstellt und in vielfältigen Bereichen Anwendung findet: Neben Stadtplanung, der Modellierung chemischer Molekülstrukturen, der Analyse persönlicher Beziehungen oder der Projektplanung, ist die Graphentheorie eine essenzielle Grundlage der Informatik (vgl. ebd.). Hier findet sie in der Planung von Computernetzwerken, der Konzeption von Datenstrukturen oder der Entwicklung von Algorithmen Anwendung. (vgl. Krumke und Noltemeier 2012: 16ff.) Ebenso ist die Graphentheorie eins der wichtigsten Werkzeuge für die Strukturierung und Entwicklung von Auszeichnungs- und Programmiersprachen und ist *vice versa* die wichtigste Grundlage, wenn es, wie in der vorliegenden Arbeit, darum geht, eine solche Sprache grafisch darzustellen.

Eine umfangreiche Beschreibung der Graphentheorie ist im Kontext dieser Arbeit nicht zielführend. Daher orientieren sich die folgenden Ausführungen an der grafischen Struktur von XProc und den Grundlagen, die für dessen Definition notwendig sind.

2.2.1 Definitionen und Grundbegriffe

> "Graphs are so named because they can be represented graphically, and it is this graphical representation which helps us understand many of their properties."
> (Bondy und Murty 1976: 2)

In allen Anwendungsbereichen der Graphentheorie wird versucht, anhand der grafischen Abstraktion alltäglicher bzw. real existierender Problemstellungen theoretische Lösungs(an)sätze und Schemata zu entwickeln. Die Fülle bereits existierender graphentheoretischer Sätze und Definitionen macht es heute möglich, komplexe Probleme zu bearbeiten und sie grafisch darzustellen.
Aus der Abstraktion konkreter physischer oder semantischer Elemente und ihrer Verbindungen untereinander ergeben sich die zwei wesentlichen Objekte der Graphentheorie: **Knoten** und **Kanten**. Im Beispiel des Königsberger Brückenproblems waren die Knoten Orte, zu denen man gelangen wollte. Die Kanten standen für die Brücken, die diese Orte miteinander verbunden haben. In anderen Kontexten könnten die

Knoten z. B. für Personen, Computer oder auch Wörter stehen. Ein Knoten wird im Graph in der Regel als Punkt dargestellt. Die Kanten werden als Linie dargestellt, welche die Knoten miteinander verbindet, und können demnach persönliche Beziehungen, Netzwerkverbindungen oder syntaktische Regeln repräsentieren.
Die Grundlagen der Graphentheorie beziehen sich überwiegend auf **ungerichtete Graphen**. Ein ungerichteter Graph wird definiert als eine Menge von Knoten, die durch eine Menge von Kanten verbunden ist. Hieraus ergibt sich die Formel G = (V, E), wobei G für Graph, V für die **Knotenmenge** (engl.: „vertex“) und E für die **Kantenmenge** (engl.: „edge“) steht. Die Elemente von V werden meistens mit (v_1, v_2, v_3 …) bezeichnet. Die Elemente der Wertpaare von E können mehrfach auftauchen, da an einem Knoten mehrere Kanten „hängen“ können. (vgl. Ruohonen 2013: 1) Sie werden häufig in der folgenden Form notiert: {(e_1, e_2), (e_2, e_3), (e_1, e_3) …}. Ein Knoten kann also durch Kanten mit einer Vielzahl anderer Knoten verbunden werden. Jedoch besteht bei einem ungerichteten Graph zwischen zwei Knoten immer nur eine Kante. Die Wertpaare der Kantenmenge sind hier vertauschbar ((e_1, e_2) = (e_2, e_1)), da die Kante keine Richtung hat. Anfangs- und Endknoten sind nicht festgelegt.
Im Fall von XProc sind besonders **gerichtete Graphen** (auch **Digraphen**, von engl.: Directed Graph) von Interesse. Die Kanten eines gerichteten Graphen haben stets eine Richtung, also einen Anfangs- und einen Endknoten und werden mitunter auch als **Pfeile** oder **Bögen** bezeichnet. (Vgl. Krumke und Noltemeier 2012: 7ff.) Anders als bei ungerichteten Graphen kann es hier zwischen zwei Knoten auch zwei Kanten geben, wenn sie in entgegengesetzten Richtungen verlaufen. Diese werden **parallele Kanten** genannt. (Vgl. Tittmann 2011: 13) Das Wertpaar einer Kantenmenge (e_1, e_2) lässt sich somit nicht beliebig vertauschen. Das Wertpaar ist geordnet. (Vgl. Hartmann 2012: 242) Krumke und Noltemeier (ebd.) definieren einen gerichteten Graphen durch die Formel G = (V, R, α, ω), wobei V für die Knotenmenge, R für die **Pfeilmenge**, α für den Anfangsknoten und ω für den Endknoten steht.

In der Konzeption und Analyse von Graphen gibt es einige wiederkehrende Fragen, aus denen sich wesentliche Grundsätze der Graphentheorie ableiten lassen. Häufige Fragen sind:

- Wie stehen Knoten und Kanten in Beziehung zueinander?
- Welche benachbarten Knoten hat ein bestimmter Knoten?
- Was ist der schnellste und direkteste Weg zwischen zwei Knoten?
- Wie groß ist die Entfernung zweier Knoten?

- Wie viele Verbindungen bzw. Kanten hat ein Knoten?
- Welche Struktur hat ein Graph und welche Regeln und Hierarchien sind in der Navigation durch seine Elemente zu beachten?

  (Vgl. Tittmann 2011: 11f.)

Im Bereich der Programmierung sind insbesondere die Fragen nach dem **kürzesten Weg** innerhalb einer grafischen (Daten-)Struktur entscheidend. Hierdurch kann die Geschwindigkeit, z. B. bei der Ausführung einer Funktion mit dem Zugriff auf bestimmte Element-Knoten, deutlich beeinflusst werden.
Ein wichtiger Anhaltspunkt für die Ermittlung des kürzesten Weges ist die Untersuchung der (unmittelbaren) Nachbarschaft von Knoten: die **Adjazenz**. Bei ungerichteten Graphen ist hier zunächst die Ermittlung des **Knotengrads** von Interesse. Der Knotengrad zeigt an, wie viele Kanten mit einem Knoten verbunden sind. (Vgl. Tittmann 2011: 13) Er definiert also die möglichen Wege, die von diesem Knoten ausgehen.[38] Sind zwei Knoten miteinander verbunden, werden sie **adjazent** genannt.
Die Nachbarschaft von Knoten ist auch bei gerichteten Graphen für die weitere Analyse grundlegend. Ein wesentliches Merkmal für die Bestimmung der Adjazenz sind hier **Eingangsgrad** und **Ausgangsgrad** eines Knotens. (Vgl. Hartmann 2012: 243) Der Eingangsgrad bestimmt die Anzahl der Kanten, die in einen Knoten ‚hineinlaufen'. Der Ausgangsgrad bestimmt demgegenüber die Anzahl der Kanten, die von einem Knoten abgehen. Krumke und Noltemeier (vgl. 2012: 8) bezeichnen Eingangs- und Ausgangsgrad etwas anschaulicher als mündende und ausgehende „Pfeilbüschel".

Grundsätzlich gilt, dass das **Durchlaufen eines Graphen** in einer sogenannten **Kantenfolge** beschrieben wird. Eine Kantenfolge beschreibt Tittmann (2011: 15) als eine „[…] Folge v_1, e_1, v_2, e_2, v_3, …, v_{k-1}, e_{k-1}, v_k von Knoten und Kanten eines Graphen G, sodass die Kante e_i für i = 1, k - 1 jeweils die Endknoten v_i und v_{i+1} besitzt." Aus der Anzahl der vorhandenen Kanten ergibt sich somit die Länge der Kantenfolge.
Es wird weiterhin unterschieden zwischen **offenen** und **geschlossenen** Kantenfolgen. Eine Kantenfolge ist offen, wenn der erste und letzte Knoten (v_1 und v_k) nicht identisch sind. Im Gegenzug ist eine Kantenfolge geschlossen, wenn jene Knoten identisch sind.
Beim eulerschen Brückenproblem wurde nach einem Weg gesucht, der jede Kante *genau* einmal durchläuft, also eine Kantenfolge in der jede Kante genau einmal vorhanden ist. Wenn dies gelingt, spricht man deshalb in der heutigen Graphentheorie von einem

38 Theoretisch kann ein Graph aus einem einzigen, unverbundenen Knoten bestehen, dessen Knotengrad somit gleich null ist. Hier spricht man von einem **isolierten Knoten**.

eulerschen Weg. (Vgl. Krumke und Noltemeier 2012: 44) In der deutschsprachigen Literatur zur Graphentheorie wird ein **Weg** als solcher bezeichnet, wenn in einer Kantenfolge jeder Knoten eines Graphen *höchstens* einmal vorkommt. (Vgl. Hartmann 2012: 228) Der Weg durch einen Graphen wird von Ruohonen (vgl. 2013: 6) als endliche Sequenz bezeichnet. Es gibt also immer einen Start und einen Endpunkt.
Ist ein Weg geschlossen ($v_1 = v_k$), so spricht man von einem **Kreis**. (Vgl. Nitzsche 2009: 46f.) Hartmann (ebd.) definiert einen Kreis als „[…] geschlossene Kantenfolge, in der bis auf Anfangs- und Endknoten alle Knoten verschieden sind." Ein Graph, der aus einem „[…] einfachen geschlossenen Weg besteht […]" (Turau und Weyer 2015: 25) wird als **zyklisch** bezeichnet. Die Definition von Kreisen ist auch mit Blick auf XProc von Bedeutung, da es sie dort *nicht* geben darf und XProc **azyklisch** strukturiert ist. Ein Graph wird somit als azyklisch oder **kreisfrei** bezeichnet, wenn in ihm kein Kreis, also kein geschlossener Weg vorkommt. (Vgl. Krumke und Noltemeier 2012: 34f.)
Eine besondere Form des Kreises ist die **Schlinge**. Diese verbindet einen Knoten mit sich selbst. Anfangs- und Endknoten fallen hier zusammen. (Vgl. Tittmann 2011: 15)
Hinsichtlich der Frage nach dem kürzesten Weg, ist die Suche hiernach in der Informatik zwar sehr bedeutsam und das Forschungsfeld der „Knotenbesuchsalgorithmen" (Hartmann 2012: 240) wird intensiv erforscht. Jedoch gibt es viele Situationen in denen Kürze und Geschwindigkeit nicht die einzig wichtigen Maßstäbe bei der Strukturierung von Programmen, Daten oder Sprachen sind. So kann es z. B. ebenso notwendig sein, Knoten mehrfach anzulaufen, um Anfragen in verschiedenen Kontexten zu stellen. Dies geschieht in offenen sowie geschlossenen Kantenfolgen häufig.

2.2.2 Planarität und Baumstrukturen

Um Graphen voneinander zu unterscheiden und wiederverwendbare Systeme für verschiedene Kontexte zu entwickeln, werden sie in verschiedene Kategorien oder Typen eingeteilt. Im Folgenden werden drei Graphentypen vorgestellt, die im Kontext der grafischen Darstellung von XProc relevant sind.

Planare Graphen

Als planar oder auch **plättbar** werden Graphen bezeichnet, „[…] die sich ohne Überkreuzungen von Kanten in eine Ebene zeichnen lassen." (Tittmann 2011: 43) Die Kanten dürfen sich höchstens in den Schnittpunkten der Knoten kreuzen.
Planare Graphen gehören zum mathematischen Themengebiet der **Topologie**, da veränderliche Eigenschaften wie Gestalt und Größe hier keine Rolle spielen, sondern der

Fokus auf dem Zusammenhang der Knoten des Graphen liegt. Da Kanten auf unterschiedliche Weise gezeichnet werden können, ist es nicht trivial zu bestimmen, ob ein Graph planar ist oder nicht.

Die grundsätzliche Möglichkeit Graphen überschneidungsfrei zu zeichnen, ist demnach eine Frage der vorhandenen Dimensionen. In einem dreidimensionalen Vektorraum lassen sich die Knoten eines endlichen Graphen immer so anordnen, dass sich die Kanten höchstens in den Knoten schneiden. (Vgl. Hartmann 2012: 226f.) Ein solcher Graph wird **geometrischer Graph** genannt. Wird ein geometrischer Graph in den zweidimensionalen Raum ‚gedrückt' bzw. wird er ‚geplättet', ist eine überschneidungsfreie Darstellung in der Ebene nicht immer möglich. Abb. 2 zeigt einen planaren Graphen. Durch eine veränderte Positionierung der Knoten zeigt sich, dass der Graph überschneidungsfrei dargestellt werden kann.

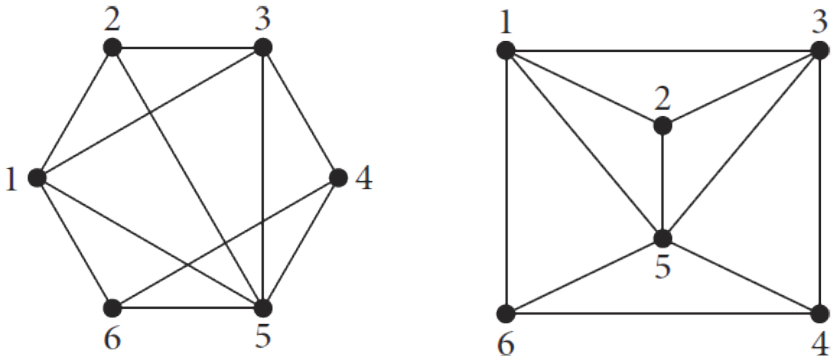


Abb. 2: Planarer Graph (Turau und Weyer 2015: 28)

Zusammenhängende und unzusammenhängende Graphen

Ein Graph wird als zusammenhängend bezeichnet, wenn er aus einer einzigen **Zusammenhangskomponente** besteht. (Vgl. Hartmann 2012: 228) Zwischen je zwei Knoten existiert hier ein Weg. Das Schaubild in Abb. 3 zeigt demgegenüber einen unzusammenhängenden Graphen mit zwei Zusammenhangskomponenten. Ein unzusammenhängender Graph entsteht, indem sogenannte **Brücken** eines Graphen entfernt werden. Eine Brücke ist eine Kante, durch deren Entfernen ein Graph in zwei Komponenten aufgeteilt wird. (Vgl. Tittmann 2011: 18) Durch **Fusion** lassen sich Graphen wieder zusammenführen. Wichtig: Zusammenhangskomponenten sind nicht zu verwechseln mit **Untergraphen**, die lediglich Teilmengen eines Graphen beschreiben, jedoch nicht voneinander unabhängig sind. (Vgl. Tittmann 2011: 15)

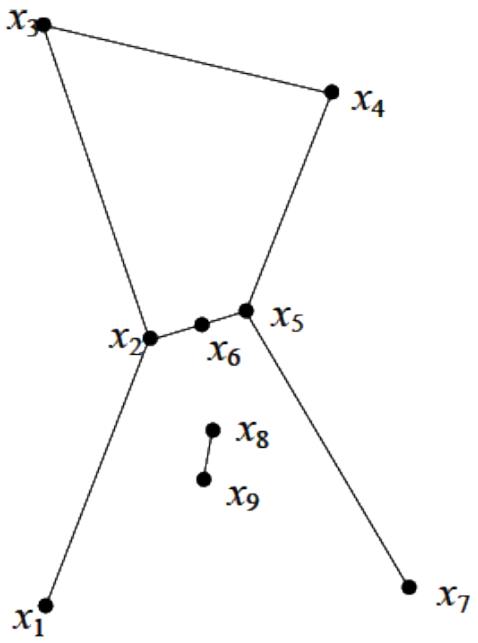


Abb. 3: Unzusammenhängender Graph
(Hartmann 2012: 228)

Bäume

Der Baum ist eins der bekanntesten Konstrukte der Graphentheorie. Er taucht im Alltag in unterschiedlichen Ausformungen auf: Als Ablaufdiagramm, Entscheidungsbaum, Organigramm u. v. m. Im Feld von Auszeichnungs- und Programmiersprachen kommen Bäume häufig und mit verschiedenen Funktionen vor.

Ein Baum wird in der Graphentheorie definiert als „[…] zusammenhängender Graph, der keinen Kreis besitzt." (Tittmann 2011: 21) Hieraus folgt, dass der Knoten eines Baumes ausschließlich mit seinen direkt benachbarten Knoten verbunden ist und eine Kante bei der Verbindung zweier Knoten niemals einen Knoten ‚überspringt'. Ein Baum ist somit stets auf eine lineare Erweiterung bzw. Verästelung seiner Struktur ausgerichtet (Vgl. Hartmann 2012: 229f.). Von jedem Knoten eines Baumes können neue **Zweige** abgehen und die letzten Knoten am Ende dieser Zweige werden als **Blätter** bezeichnet. Das Schaubild in Abb. 4 zeigt zwei Bäume mit simpler Struktur.

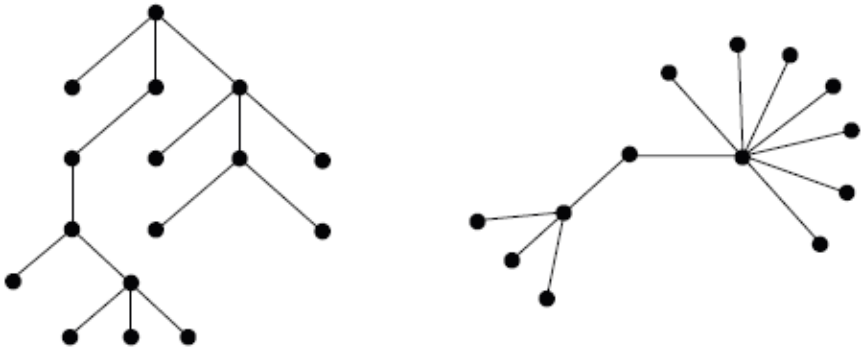


Abb. 4: Baumstrukturen (Hartmann 2012: 229)

Die Kanten eines Baumes haben eine besondere Bedeutung für den Zusammenhang des Graphen. Wird eine beliebige Kante des Baumes entfernt, zerfällt dieser zwangsläufig in zwei Bäume und zwei Zusammenhangskomponenten. Jede Kante eines Baumes ist somit eine Brücke. (Vgl. Hartmann 2012: 230) Enthält ein Graph mehrere Bäume, spricht man in der Graphentheorie anschaulich von einem **Wald**.

Das **Durchlaufen eines Baumes** folgt einer stark festgelegten Route. Die Wege in einem Baum können niemals geschlossen werden, da er kreisfrei ist. Zwei Knoten sind somit immer nur durch einen Weg verbunden. (Vgl. ebd.) Ein Weg führt immer linear zum nächsten Knoten bis ein letzter Knoten erreicht ist, von dem aus es nur noch zurück zu einem bereits besuchten Knoten ginge. Jeder Knoten, dessen Kantengrad größer ist als 1, bietet die Möglichkeit einer Entscheidung, wie der Weg weitergeführt werden soll. Turau und Weyer heben in ihrem Buch *Algorithmische Graphentheorie* die Bedeutung der direkten Wege in Bäumen für die Programmierung hervor: „Die Eigenschaft, dass es keine geschlossenen Wege gibt, führt […] zu effizienteren Datenstrukturen." (Turau und Weyer 2015: 68)

Es liegt nahe, einen Baum als ‚von Natur aus' gerichteten Graphen zu verstehen. Die eindeutige Orientierung seiner Wege und die häufige Darstellungsweise als von oben nach unten orientiertem Graph stützen diese Ansicht. Dies würde jedoch die wichtige Tatsache ignorieren, dass ein einfacher Baum keinen Ursprungsknoten hat, sondern alle Knoten eines Baumes „gleichberechtigt" (Hartmann 2012: 231) sind. Die Laufrichtung ist somit bei einem normalen Baum nicht festgelegt und er kann sowohl gerichtet, als auch ungerichtet sein. Dies ist anders beim sogenannten **Wurzelbaum**, der als gerichteter und hierarchischer Baum erscheint und eine große Bedeutung im Kontext von XML-Sprachen hat. Der folgende Abschnitt geht hierauf ein.

2.2.3 XML, ein Wurzelbaum

> „The data model for XML is very simple - or very abstract, depending on one's point of view. XML provides no more than a baseline on which more complex models can be built."
> [W3C – 2005 – XML – Datamodel]

In simplen Darstellungen von XML-Strukturen werden stark vereinfachte XML-Elemente als Begriffe in einem Rechteck oder einem Kreis miteinander verbunden, ohne Inhalt oder zugewiesene Attribute mit einzubeziehen. (siehe Abb. 5) Neben solchen simplen Modellen stellt die grafische Modellierung von XML eine eigenständige und elementare XML-Technologie dar. Im Folgenden beziehe ich mich auf die Baumstruktur-Modelle der XML-Pfadbeschreibungssprache XPath (vgl. Bongers 2008: 55ff.) sowie des Document Object Models (DOM). (Vgl. ebd.: 389ff.)

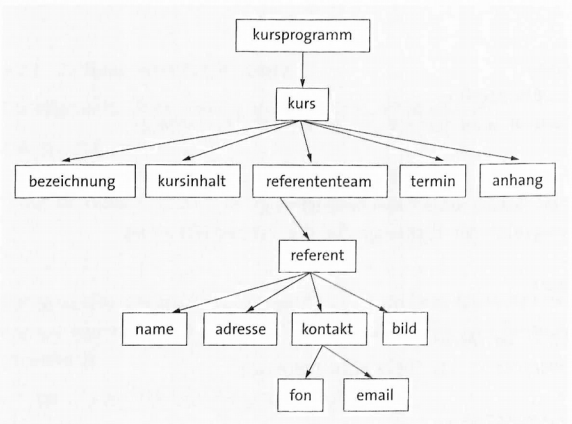


Abb. 5: XML-Baumstruktur (Vonhoegen 2015: 85)

Die grafische Struktur eines XML-Dokuments entspricht weitestgehend einer einfachen Baumstruktur, die von oben nach unten ausgerichtet ist. Das Vokabular, das in der Beschreibung und Definition von XML-Dokumenten benutzt wird, gibt hierbei einige Hinweise auf die graphentheoretische Struktur, sowie auf die Herkunft vieler verwendeter Begriffe: Jedes Element eines XML-Dokuments steht im XML-Baum für einen **Elementknoten**. Dieser Elementknoten enthält gleichzeitig den Start- und Endtag des Elements. Jedem Elementknoten ist außerdem ein Name zugeordnet. Attribute und Inhalt des Elements werden in Form von **Attributknoten** und **Textknoten** an den Elementknoten angehängt. Polyzotis und Garofalakis (2002: 2) definieren XML diesbezüglich als einen „[…] directed, node-labeled *data graph* […]". Das Schaubild in Abb. 6 zeigt ein XML-Dokument mit der entsprechenden grafischen Baumstruktur:

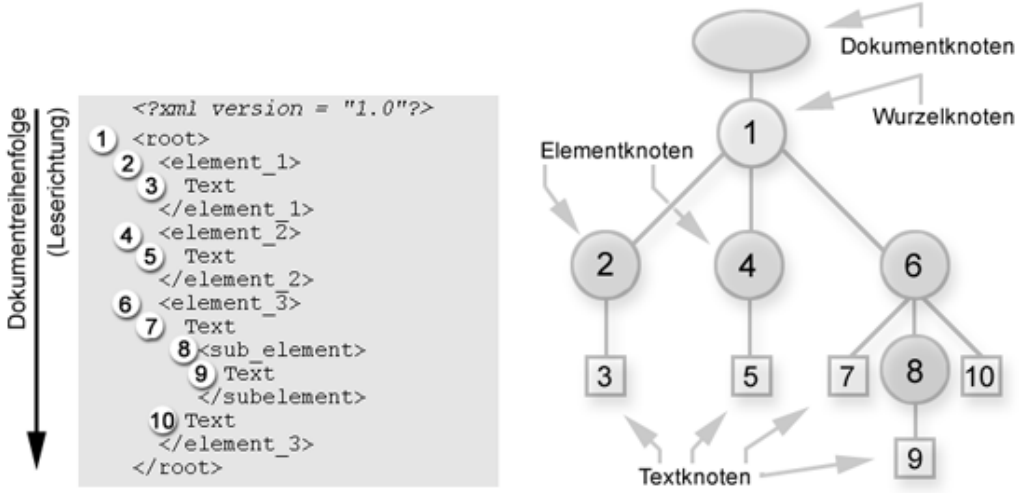


Abb. 6: XML-Dokumentenstruktur mit Darstellung als Baum (Bongers 2008: 56)

Anders als bei einer normalen Baumstruktur sind nicht alle Knoten „gleichberechtigt“, wie Hartmann (2012: 231) es ausdrückt. Da jedes XML-Dokument ein einziges Wurzelelement hat, gibt es auch in der grafischen Abbildung einen Knoten, von dem alle anderen Knoten abstammen. Hierbei darf nicht vergessen werden, dass selbst das Wurzelelement eines XML-Dokuments noch einem weiteren Element untergeordnet ist: dem Dokument selber. Im DOM- und XPath-Modell wird dieses als **Dokumentenknoten** bezeichnet. In der Graphentheorie wird dieser Knoten **Wurzel** oder **Wurzelknoten** genannt. Ein Baum mit Wurzel heißt hier **Wurzelbaum**. (Vgl. ebd.)

Die Wurzel ist die wesentliche Größe, wenn es um die Ermittlung der Verwandtschaftsbeziehungen und der Nachbarschaft von Knoten geht. Dadurch, dass alle Knoten von einer Wurzel abstammen, entsteht eine klar definierte Verwandtschaftsstruktur mit **Elternknoten** und **Kindknoten**.

Das DOM dient der Strukturierung und Speicherung der Verwandtschaftsbeziehungen von XML-Elementen und bietet eine datenbasierte Repräsentation des XML-Wurzelbaums. (Vgl. Vonhoegen 2015: 389f.) Hierdurch ist es möglich von jedem XML-Element aus auf diesen Datensatz zuzugreifen. Dabei erscheint das DOM dank der *Node*-Schnittstelle stets im Kontext des Elements, von dem aus es aufgerufen wurde. So lässt sich direkt auf über- und untergeordnete sowie benachbarte Elemente zugreifen und wiederum deren Verwandtschaft betrachten.

Die Reihenfolge der Knotenverknüpfungen im XML-Wurzelbaum basiert auf zwei Größen: den Verwandtschaftsbeziehungen der XML-Elemente und der **Dokumentenreihenfolge** des XML-Dokuments. Durch die Verschachtelungen der XML-Elemente ineinander entstehen deren Beziehungen und Verknüpfungen implizit. Die Dokumentenreihenfolge wird durch die Position der XML-Elemente und ihrer Attribute und Inhalte im XML-

Dokument festgelegt. (Vgl. Vonhoegen 2015: 190) In Abb. 6 ist die Reihenfolge und Positionierung der Knoten in der Baumstruktur auf Grundlage der Dokumentenreihenfolge angegeben.

Die XML-Baumstruktur wird in vielen verschiedenen Bereichen aktiv verwendet. In der Webentwicklung dient der DOM-Baum dem schnellen Zugriff z. B. durch JavaScript-Funktionen. Im Bereich der XSLT-Transformation ist die Baum-Adressierbarkeit mit XPath ein essentielles Werkzeug und hat hier gleich mehrere Funktionen (Vgl. Bongers 2008: 62f.): Bei einer Transformation mit XSLT werden sowohl das XML-Quelldokument als auch das erstellte Stylesheet durch den XML-Parser zunächst in eine Baumstruktur überführt. In der Verarbeitung geht der Prozessor von oben nach unten durch den Dokumentenbaum des XML-Dokuments und arbeitet sich anhand des Konzepts der **Current Node** von einem Knoten zum nächsten ab, wobei an jeder Stelle ggf. aktivierte Template-Regeln abgefragt und umgesetzt werden. Bei der Verarbeitung entsteht sukzessive ein sogenannter Ergebnisbaum, der schließlich in ein neues Dokument serialisiert wird. Das Schaubild in Abb. 7 verbildlicht den beschriebenen Transformationsprozess.

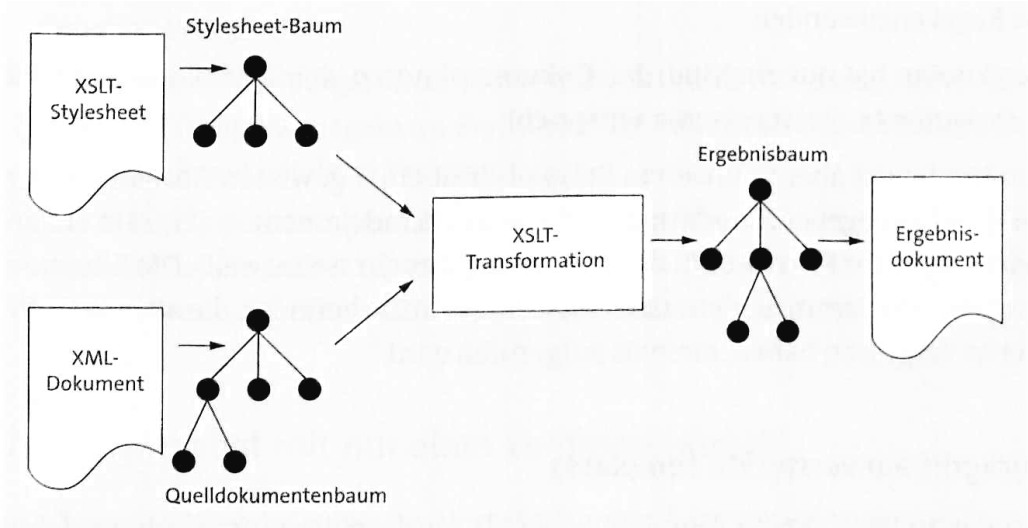


Abb. 7: Baumstrukturen in der XSLT-Transformation (Vonhoegen 2015: 267)

Vonhoegen (2015: 190) beschreibt, beim Schritt der **Serialisierung** werde die Baumstruktur zunächst geplättet, also in die Ebene gebracht, und die Knoten anschließend in Dokumentenreihenfolge in das neue Dokument geschrieben. Dies zeigt, dass sich die Dimensionalität der Baumstruktur im Transformationsprozess verändert (siehe ‚Planare Graphen' in Abschnitt 2.2.2), auch wenn dies nicht sichtbar ist. In Kapitel 2.3 wird die Frage mehrdimensionaler Datenstrukturen im Kontext der grafischen Struktur von XProc erneut aufgegriffen.

2.2.4 Gerichtet, Kreisfrei, XProc

In der Beschreibung des XProc-Graphen ist wesentlich zwischen der graphentheoretischen Struktur eines XProc-Dokuments und der Struktur einer deserialisierten bzw. zu serialisierenden XProc-Pipeline zu unterscheiden.
Da ein XProc-Dokument gleichsam ein XML-Dokument ist, fällt es nach den Erörterungen im Abschnitt 2.2.3 leicht, dessen grafische Grundstruktur zu beschreiben. Das folgende Zitat von Frank Bongers fasst die wichtigsten, graphentheoretischen Aspekte der XML-Baumstruktur zusammen, die sich somit auch auf XProc anwenden lassen:

> „Noch genauer: Es handelt sich um einen »gerichteten azyklischen Graph« (directed acyclic graph, DAG) im Sinne der Graphentheorie. Dass der Baum »azyklisch« ist, also keine Zweige in sich selbst zurücklaufen, hat den Vorteil, dass beim Traversieren des Baums keine Endlos-Schleifen entstehen können."
> (Bongers 2008: 56)

Die im Zitat genannte Abkürzung DAG wird auch in den folgenden Ausführungen für gerichtete azyklische Graphen genutzt. Bongers weist in dem Zitat auf die lineare und effiziente Verarbeitungsstruktur in einem DAG hin. Für die Verarbeitung einer XProc-Pipeline ist es elementar, dass keine Endlosschleifen vorkommen dürfen. Die Serialisierung als XML-Baum und die Ausführung der Pipeline würden andernfalls fehlschlagen. Dies hat gleichermaßen Implikationen für die Verknüpfungsstruktur in einem XProc-Editor.
Der Graph eines XProc-Dokuments lässt sich diesbezüglich als DAG mit Baumstruktur beschreiben.
XProc-Pipelines enthalten eine wesentliche Besonderheit, die den XProc-Graphen von einem normalen XML-Graphen unterscheidet: die Default Readable Ports. Für die Verarbeitung einer XProc-Pipeline ist das Konzept der DRPs elementar. In der Deserialisierung spannen diese impliziten Verbindungen den XProc-Graphen auf. Die Visualisierung eines XProc-Dokuments folgt demnach klaren Regeln. Es gibt eine Dokumentenreihenfolge, anhand derer die Elemente in Flussrichtung angeordnet und Verbunden werden.
Der XProc-Editor soll in zwei Richtungen funktionieren: Es sollen Pipelines aus einem XProc-Dokument visualisiert werden und eine XProc-Pipeline anhand grafischer Elemente zusammengestellt bzw. bearbeitet werden. Es besteht somit eine doppelte Herausforderung: Aus einem XProc-Dokument soll ein zweidimensionaler Graph geladen werden, der alle Verknüpfungen der Pipeline abbildet. Außerdem soll ein im 2D-Editor neu erstellter XProc-Graph in ein eindimesionales XProc-Dokument übertragen werden.

Bei der hierfür notwendigen Serialisierung muss der aufgespannte XProc-Graph geplättet und in eine Reihenfolge gebracht werden. Das zentrale Problem eines zweidimensionalen Visualiserungsframeworks ist, dass es hier keine natürliche Dokumentenreihenfolge gibt. Eine Serialisierung wäre somit uneindeutig. Zur Veranschaulichung dieses Problems, zeigt das folgende Schaubild eine exemplarische XProc-Pipeline:

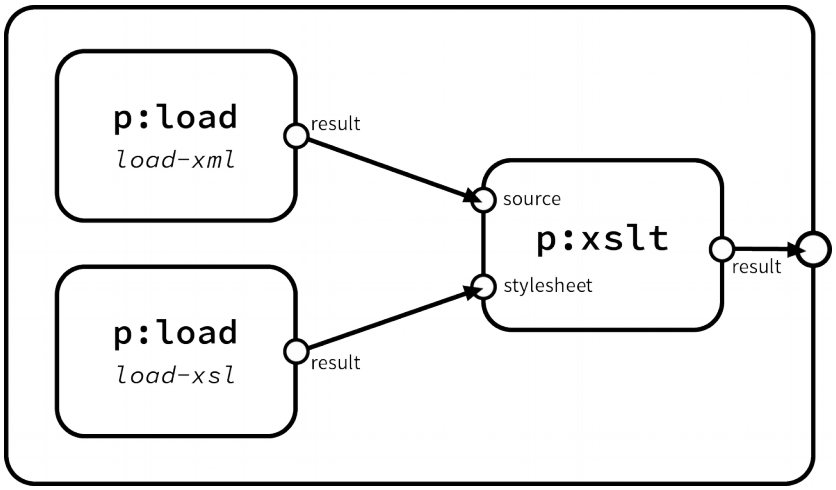


Abb. 8: Grafische XProc-Pipeline

Die Pipeline beschreibt einen `p:xslt`-Step, der zur Ausführung seiner Transformation von zwei `p:load`-Steps mit XML-Daten und einem XSL-Stylesheet versorgt wird. Würde eine solche Pipeline in einem Editor zusammengestellt werden, wären die Verbindungen zwischen den `p:load`-Steps und `p:xslt` gleichberechtigt. Es gäbe keine definierte Verarbeitungsreihenfolge oder einen primären Pfad. Aus diesem Graphen könnten somit verschiedene XProc-Dokumente serialisiert werden. Das folgende Code-Beispiel zeigt zwei mögliche Pipelines:

```
<?xml version="1.0" encoding="UTF-8"?>
<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>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<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"
```

```
    <p:input port="parameters">
      <p:empty/>
    </p:input>
  </p:xslt>

</p:declare-step>
```

```
      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>
```

Code-Beispiel 6 : Gegenüberstellung von zwei serialisierten XProc-Pipelines

Neben der unterschiedlichen Zeilenlänge gibt es hier mehrere Unterschiede in der Code-Struktur. Es ist dabei grundlegend, dass die beiden `p:load`-Steps in unterschiedlicher Reihenfolge in das Dokument serialisiert wurden. Aufgrund der DRPs der beiden Steps verändert sich die folgende Verarbeitung. Im linken Beispiel bleibt der primäre Output-Port von `load-xsl` unverbunden, da `load-xml` keinen Input-Port hat. In der XSLT-Verarbeitung wird load-xsl dann über ein `p:pipe` explizit mit dem `stylesheet`-Input-Port verbunden. Der Step `load-xml` wird über seinen DRP automatisch mit dem DRP `source` von `p:xslt` verbunden.
Im rechten Beispiel würde der DRP von `load-xsl` automatisch mit dem `source`-Port von `p:xslt` verbunden werden. Dies würde zu einer fehlerhaften Verarbeitung führen, weshalb der Output von `load-xsl` mit einem `p:sink`-Step abgefangen werden muss. `p:sink` empfängt den Output und löscht ihn. Da `p:sink` keinen Output hat, müssen in der Folge beide `p:load`-Steps explizit mit den entsprechenden Input-Ports von `p:xslt` verbunden werden.

Das Code-Beispiel zeigt die Bedeutung der DRPs und der Dokumentenreihenfolge für eine effiziente Verarbeitung. Und es ist auch zu erkennen, dass das Konzept des DRP nur dann sinnvoll ist, wenn es auch eine Dokumentreihenfolge gibt. Die zentrale Herausforderung eines grafischen XProc-Editors ist es somit, die Dokumentenreihenfolge durch ein ‚künstliches' DRP-Konzept zu erzeugen. Die Maxime für diese Konzeptualisierung ist die Optimierung des serialisierten Output-Dokuments. Hierbei lässt sich als Faustformel festhalten: Je weniger `p:sink`-Steps im serialisierten XProc-Dokument enthalten sind, desto besser war die Serialisierung.

2.3 Visualisierung von XProc

> „[...] XProc code, like all code, is one dimensional. The text your code consists of runs from top to bottom. As a consequence, the resulting flow of data is not immediately apparent when things get a little complicated."
> (Siegel 2019: 91)

Erik Siegel (ebd.) vergleicht XProc in seiner Einführung zu XProc 3.0 mit einem verzweigten, dreidimensionalen Netz aus Rohren, wie beispielsweise in einer Gas-Raffinerie. An Stelle von Gas, fließen durch die Rohre jedoch XML-Daten, die an verschiedene Stationen weitergegeben, und dort verarbeitet werden. Im Zuge dieses Vergleichs, weist Siegel auf ein wesentliches Problem hin, wenn es darum geht, sich die Funktionsweise von XProc vorzustellen bzw. XProc darzustellen: Die XProc-Verarbeitung geschieht dreidimensional, der XProc-Code ist, wie jeder Code, aber nur eindimensional. Siegel drückt diesbezüglich Verständnis dafür aus, dass der Datenfluss und der *Weg*, den die Pipeline beschreibt, aus dem Quellcode mitunter schwer zu erfassen sind.
Diese Abschlussarbeit stellt sich darüber hinaus die Frage, wie der eindimensionale XProc-Code, in eine zweidimensionale Darstellung übersetzt werden kann, obwohl sich die XProc-Verarbeitung dreidimensional vollzieht.
Aus den Ausführungen zur Graphentheorie, der Definition des XProc-Graphen und der genaueren Beschreibung von XProc selbst, gehen wesentliche Aspekte hervor, die bei der grafischen Darstellung von XProc unbedingt zu beachten sind. Diese Aspekte werden im Folgenden zusammengefasst und als Feature-Matrix zur Prüfung möglicher passender grafischer Frameworks genutzt.

2.3.1 Anforderungen an ein Framework

Ein grafisches Framework oder eine Software sollte folgende Aspekte auf jeden Fall erfüllen, um XProc grafisch bzw. visuell umsetzen zu können:

Gerichtet

Das Framework muss in der Lage sein, gerichtete Graphen zu erstellen. Den Kanten eines Graphen müssen eindeutige Richtungen zugewiesen werden können.

Interaktiv

Nutzer*innen müssen die Möglichkeit haben, innerhalb einer grafischen Benutzeroberfläche (GUI) mit allen Elementen des Graphen zu interagieren. Das Framework sollte intuitive Bedienelemente anbieten, die sowohl eine Bearbeitung der

grafischen Darstellung, als auch eine dynamische Daten-Interaktion ermöglicht. Die Bedienelemente sollten den jeweiligen Erfordernissen angepasst werden können.

Anpassbar

Es reicht nicht, wenn ein Framework in der Lage ist, eine XProc-Pipeline visuell darzustellen. Die Erstellung einer eigenen Pipeline und die Bearbeitung aller Elemente des Graphen müssen möglich sein.
Dies schließt auch ein zugängliches und bearbeitbares Datenmodell mit ein. Es sollte sowohl möglich sein, den Datensatz des gesamten Graphen mit allen vorhandenen Elementen abzurufen, als auch, auf die Daten einzelner Elemente zuzugreifen und diese zu manipulieren. Elementen sollten spezifische Eigenschaften zugeordnet werden können. Zudem sollten wiederverwendbare Vorlagen erstellt werden können, um die Erstellung rekursiver Graphen zu ermöglichen.

Hierarchische Kapselung

Es muss möglich sein, Elemente einander unterzuordnen und sie ineinander zu verschachteln. Dabei reicht es nicht, Elemente einer übergeordneten Gruppe zuzuordnen. Das übergeordnete Element muss ein eigenständiges Modell mit In- und Output-Ports sowie einem vollständigen Datenmodell sein. Die hierarchische Kapselung sollte bestenfalls visuell erfahrbar gemacht werden.

Ports

Den Elementen eines Graphen müssen Verbindungs-Ports hinzugefügt werden können. Den Ports müssen Eingangs- und Ausgangseigenschaften (In- und Output-Port) zugewiesen werden können. Es müssen mehrere Verbindungen gleichzeitig von den Ports abgehen oder in sie münden dürfen.

Azyklisch

Es muss möglich sein, die Verknüpfung von Elementen bzw. Ports zu Kreisen zu unterbinden. Genauer: Es müssen Regeln definiert werden können, die die Verknüpfung eines Input-Ports von Element A mit einem Output-Port von Element B verhindern. Genauso darf ein Input-Port von Element B nicht mit einem Output-Port von Element A verbunden werden.
Solche Verbindungen sollten bestenfalls bereits in der grafischen Bearbeitung verhindert werden oder zumindest anhand des zugrundeliegenden Datenmodells erkannt werden können.

Default Readable Ports

In der Visualisierung von XProc braucht es Hilfsmittel, um die impliziten Verbindungen einer XProc-Pipeline zu erkennen und darzustellen. Es muss ein Weg gefunden werden, den primären Verarbeitungsweg der grafischen Pipeline für die Serialisierung zu definieren. Das Konzept der DRPs bezieht sich auf die Dokumentenreihenfolge der Pipeline. Diese lässt sich in einer zweidimensionalen Ansicht nicht ohne weiteres darstellen bzw. automatisch erkennen. Das Framework muss in der Lage sein, DRPs festzulegen, um anhand der Verknüpfung von DRPs eine Dokumentenreihenfolge zu erzeugen. Die beschriebenen Regeln zur Prüfung der Azyklizität unterstützen dieses Feature.

Open Source

Das Framework sollte quelloffen und frei zugänglich sein. Der XProc-Editor soll der Community zur kostenlosen Nutzung und gemeinsamen Weiterentwicklung zur Verfügung stehen.
Quelloffenheit schließt auch die Qualität und Zugänglichkeit der Dokumentation des Frameworks mit ein. Nutzer*innen sollten dabei unterstützt werden, sich das Framework selbständig zu erarbeiten. Der Fokus sollte nicht darauf liegen, Premium-Dienste zu verkaufen.

2.3.2 Kandidatenfeld

Für die Erstellung des XProc-Editors wurden mehrere Frameworks in Betracht gezogen. Die engere Auswahl wird im Folgenden kurz vorgestellt. Es wurden außerdem mehrere bekannte Frameworks bzw. Anwendungen als Nullgrößen einbezogen, die augenscheinlich nicht für die Erstellung eines autonomen XProc-Editors geeignet sind, jedoch das Feld grafischer Visualisierungs-Frameworks vervollständigen. Am Ende dieses Abschnitts werden die Frameworks in einer Feature-Matrix anhand der Kriterien im vorherigen Abschnitt bewertet.

Gephi

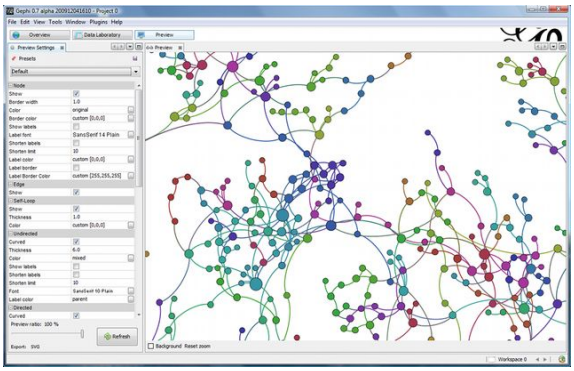


Abb. 9: Anwendungsbeispiel von Gephi [Gephi -Features]

Gephi ist eine Software zur Erstellung komplexer Datenvisualisierungen und zur Analyse von Netzstrukturen.[39] Gephi ist für alle gängigen Betriebssysteme verfügbar und basiert auf Java und OpenGL. Die Software wurde im Rahmen eines Universitätsprojekts entwickelt und im Jahr 2008 erstmals veröffentlicht.[40] Gephi nutzt für die Visualisierung eine „[...] special 3D render engine to render graphs in real-time." (Bastian und Heymann 2009: 361) Hiermit lassen sich laut eigenen Angaben über 20.000 Knotenpunkte auf einmal erstellen und bearbeiten. Verschiedene Layout-Algorithmen ermöglichen die automatische Anordnung der Daten. Die Software ist Open Source und wird aktuell unter der GNU General Public License 3 zum kostenfreien Download angeboten.[41]

GoJS

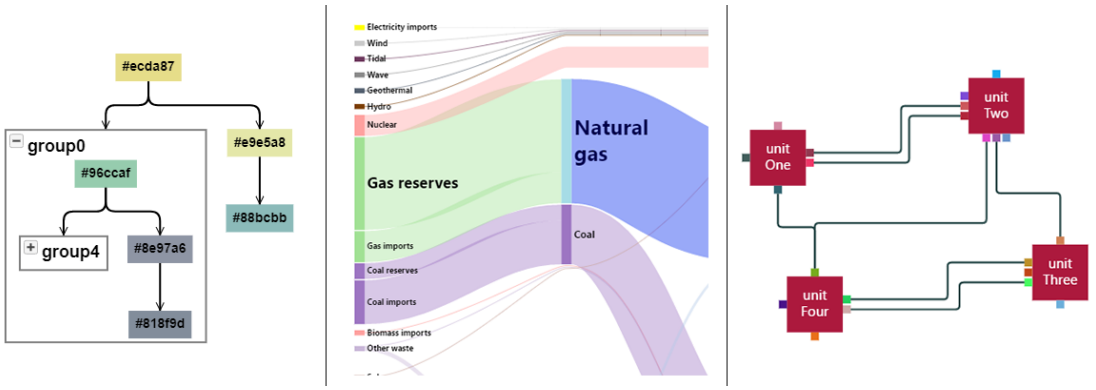


Abb. 10: Anwendungsbeispiele von GoJS [GoJS – Samples]

Mit GoJS lassen sich auf vielfältige Weise interaktive Graphen und Diagramme erstellen. Die GoJS-Library umfasst vielfältige Features, unter anderem in den Bereichen Geometrie, Layout und Modelling.[42] GoJS bietet eine anpassbare, JSON-basierte Modell-Struktur an. Die GoJS-Elemente können ineinander gruppiert werden und es ist möglich, sie über Ports miteinander zu verbinden.

---

39 Vgl. [Gephi – About]
40 Vgl. [Launchpad.net – Gephi]
41 Vgl. [Gephi – Download]
42 Vgl. [GoJS – API]

GoJS ist für die Anwendung im Web-Browser konzipiert und basiert auf JavaScript und TypeScript.[43] GoJS-Elemente können nach SVG oder HTML Canvas gerendert werden. GoJS wird seit 1998 von der Firma Northwoods Software entwickelt und vertrieben. Der Quellcode von GoJS ist abrufbar und im privaten Bereich ist eine kostenfreie Nutzung unter Berücksichtigung strenger Lizenzvorschriften möglich. Die unbeschränkte und kommerzielle Nutzung ist kostenpflichtig.

JointJS

Das JavaScript-Framework JointJS dient der Erstellung von Diagrammen, Graphen und interaktiven, grafischen Anwendungen.[44] Die JointJS-Library bietet vordefinierte, dynamische Elemente, wie Formen und Links, sowie eine komplexe Modell-Struktur mit optionalen Ports. Die JointJS-Elemente haben eine flexible, auf JSON aufbauende Datenstruktur. Die grafische Darstellung erfolgt komplett in SV.

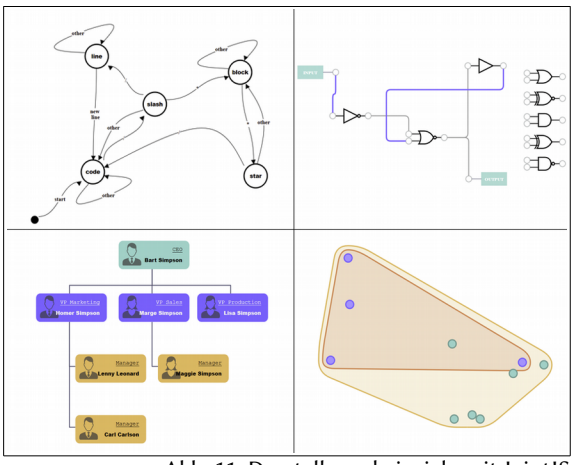


Abb. 11: Darstellungsbeispiele mit JointJS

JointJS bietet zudem eine eigene Syntax zur Programmierung von Events für die grafische Interaktion. JointJS wird seit 2009 von der Firma *client.IO* entwickelt und steht als Open Source Framework zum kostenlosen Download zur Verfügung.

43 Vgl. [GoJS]
44 Vgl. [JointJS – OpenSource]

NoFlo

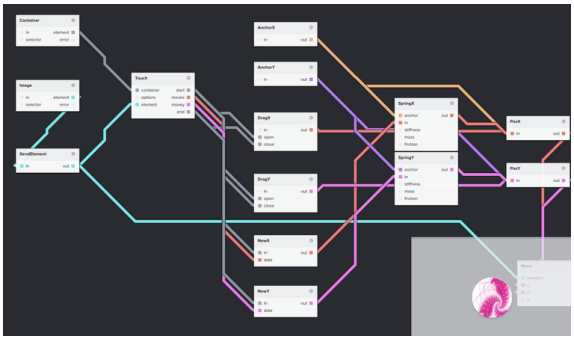


Abb. 12: Beispielanwendung mit NoFlo

NoFlo dient laut der Umsetzung des „Flow-Based-Programmings“, einer Technologie, die in den 1970er Jahren von IBM entwickelt wurde.[45] Der Fokus von NoFlo liegt auf der Visualisierung von Software-Logik. Es lassen sich jedoch auch gewöhnliche Graphen und Diagramme erstellen. Wie bei GoJS und JointJS, lassen sich mit NoFlo Datenmodelle erstellen, die in der Darstellung als grafische Datencontainer dienen. Den Modellen können Input- und Output-Ports zugewiesen werden, über die sie Daten empfangen und senden.

NoFlo basiert auf JavaScript und unterstützt zudem die Programmierung mit CoffeeScript. Neben der Möglichkeit mit NoFlo selbstständig Browser-Anwendungen zu generieren, kann auf die Entwicklungsumgebung FlowHub zurückgegriffen werden, in die NoFlo-Anwendungen eingelesen werden können.[46]

NoFlo ist ein Open Source Framework und steht unter der MIT Lizenz zum kostenlosen Download bereit. Das Framework wird von Henri Bergius und der *FlowHub UG* entwickelt.

PapDesigner

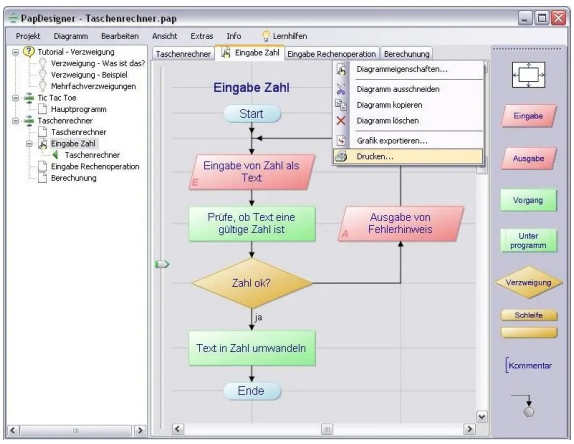


Abb. 13: Screenshot des PapDesigner [Heise - PapDesigner]

Der PapDesigner ist ein Flussdiagramm-Editor für Windows-System, der die Erstellung strukturierter Programm-ablaufpläne ermöglicht.[47] Die Formen und Verbindungen entsprechen der DIN 66 001. Die erstellten Pläne können in verschieden Formaten abgespeichert werden.

Der PapDesigner entstand in einem Hochschul-Projekt und wird aktuell von Friedrich Folkmann entwickelt.

45 Vgl. [NoFloJS – Doku]
46 Vgl. [FlowHub]
47 Vgl. [Heise – PapDesigner]

vis.js

Auf der offiziellen Homepage wird vis.js als „browser based visualization library“ beschrieben.[48] Mit vis.js lassen sich Datenvisualisierungen zu verschiedensten Themen erstellen. Hierzu ist das Framework in fünf Bereiche unterteilt: „DataSet, Timeline, Network, Graph2d [und] Graph3d.“[49] Der Bereich Network umfasst die massenhafte Generierung von Knoten und Kanten. Hierzu werden Module, Funktionen zum Datenaustausch und Events für die grafische Interaktion angeboten. Die Datenstruktur von vs.js ist sehr simpel gehalten, auch um die Visualisierung großer Datenmengen zu ermöglichen. Vis.js bietet außerdem eine Schnittstelle zur bereits beschriebenen Gephi-Software an, um dort erstellte Daten im Browser zu visualisieren. Vis.js ist Open Source, wird teilweise von der Community entwickelt und wird seit 2010 von *Almende B.V.* gewartet. Es steht unter den Lizenzen Apache 2.0 und MIT zum kostenlosen Download bereit. Zur Installation muss lediglich die JavaScript-Datei viz.js und die zugehörige CSS-Datei in eine HTML-Datei eingebunden werden.

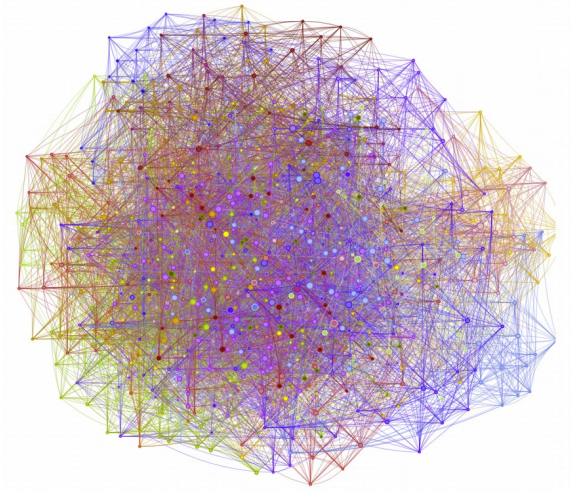


Abb. 14: Darstellungsbeispiel mit vis.js [VisJS]

yFiles

Die „diagramming company“[50] *yWorks* entwickelt Software-Komponenten zur Erstellung von interaktiven Graphen und Diagrammen. In den „yFiles Diagramming Libraries“ werden diese Komponenten für verschiedene Programmiersprachen als Plugin-Pakete zur Verfügung gestellt. *yFiles for HTML* ist eine komplexe, auf JavaScript aufbauende Entwicklungsumgebung, mit der Datenvisualisierungen für Web-Browser erstellt werden können.[51] Die Knotenpunkte der Graphen können beliebig gestaltet werden. Sie können als Datencontainer dienen und

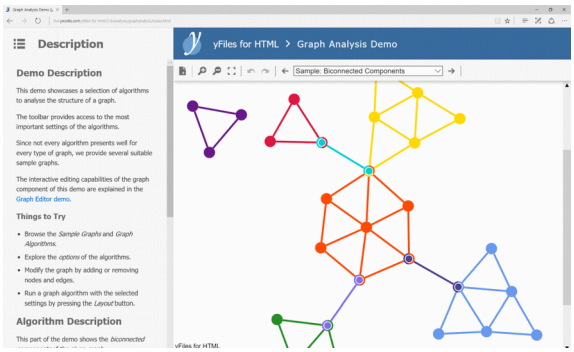


Abb. 15: Darstellungsbeispiel von *yFiles für HTML* [yWorks – yFiles for HTML]

48 Vgl. [VisJS]
49 Vgl. ebd.
50 Vgl. [yWorks]
51 Vgl. [yWorks – yFiles for HTML]

es lassen sich Verbindungsports hinzufügen. Zudem lassen sich die Elemente der Graphen gruppieren.
Die Darstellungen mit *yFiles for HTML* können in SVG, WebGL und HTML5 Canvas gerendert werden. Alle Komponenten, die von *yWorks* angeboten werden, Testversionen ausgeschlossen, sind kostenpflichtig.

2.3.3 Feature-Matrix

Die Bewertung der Frameworks soll einen groben Vergleich ermöglichen. Sie basiert hauptsächlich auf einer Analyse der online verfügbaren Ressourcen. Einige der Frameworks wurden installiert und getestet. Dies war jedoch nicht bei allen ohne größeren Aufwand möglich. Es ist daher nicht ausgeschlossen, dass bei der näheren Beschäftigung mit einem Framework, Lösungswege für scheinbar fehlende Features gefunden werden könnten. Die Feature-Matrix erfasst somit hauptsächlich die Features, auf die in den Ressourcen der Frameworks explizit hingewiesen wird.

In der Feature-Matrix sind jedem Framework zwei Spalten zugeordnet. In der ersten Spalte, wird vermerkt, ob ein Feature ganz (✓), teilweise (?) oder gar nicht (✘) unterstützt wird. In der zweiten Spalte wird die Unterstützung nach Punkten von 0 bis 4 bewertet. Insgesamt können somit 32 Punkte erreicht werden. Am Ende der Spalten ist die Anzahl unterstützter Features bzw. die Gesamtpunktzahl vermerkt.

	Gephi		GoJS		JointJS		NoFlo		Pap-Designer		vis.js		yFile	
Gerichtet	✓	4	✓	4	✓	4	✓	4	✓	4	✓	4	✓	4
Interaktiv	✓	4	✓	4	✓	4	✓	4	?	2	✓	4	✓	4
Anpassbar	?	1	✓	4	✓	4	✓	4	✘	0	✓	4	✓	4
Kapselung	✘	0	?	2	✓	4	?	2	✘	0	✘	0	?	2
Ports	✘	0	✓	4	✓	4	✓	4	✘	0	✘	0	✓	4
Azyklisch	?	2	?	3	?	3	?	2	✘	0	✘	0	?	3
DRP	✘	0	?	2	?	3	?	3	✘	0	✘	0	?	3
Open Source	✓	4	?	1	✓	4	✓	4	✓	4	✓	4	✘	0
ERGEBNIS	**3/8**	**15**	**4/8**	**24**	**6/8**	**30**	**5/8**	**27**	**2/8**	**10**	**4/8**	**16**	**4/8**	**24**

Tabelle 1: Feature-Matrix

2.3.4 Entscheidung für JointJS

Die beiden Referenzgrößen, PapDesigner und Gephi, liegen, wie zu erwarten war, in der Zahl unterstützter Anforderungen und besonders in der Gesamtpunktzahl deutlich unter dem Durchschnitt der anderen Frameworks. Von den infrage kommenden Frameworks (GoJS, JointJS, NoFlo, vis.js und yFile), fällt nur vis.js, bezogen auf die Gesamtpunktzahl, deutlich zurück. Von vis.js werden zwar genauso viele Features vollständig unterstützt, wie von GoJS und yFile, jedoch werden ebenso viele Anforderungen von vis.js gar nicht unterstützt, was sich in der Gesamtpunktzahl deutlich niederschlägt.
Die Unterschiede zwischen GoJS, JointJS, NoFlo und yFile sind in der Gesamtbewertung nicht besonders groß und lassen sich vor allem auf die Anforderung „Open Source“ zurückführen, die von GoJS und yFile nicht oder nur mäßig erfüllt wird. Diese beiden Frameworks liegen dementsprechend am weitesten zurück. Bei den restlichen Features sind die Unterschiede gering. Die ersten drei Features (gerichtet, interaktiv, anpassbar) werden von allen vier Frameworks voll erfüllt, ebenso die Unterstützung von Input- und Output Ports. Bei der hierarchischen Kapselung im Sinne einer XProc-Pipeline, sticht JointJS mit einem hierauf zugeschnittenen Konzept hervor. Bei den anderen Frameworks wird zwar meistens eine Gruppierung von Elementen unterstützt, jedoch ist die Möglichkeit einer echten Kapselung von Elementen nicht ersichtlich und bedürfte wohl größerem Entwicklungsaufwand.
Es gibt bei keinem der Frameworks ein eingebautes Feature, das es ermöglichen würde, die Bildung von Kreisen zu verhindern, um azyklische Graphen zu erstellen. Angesichts der Framework-Architektur scheint es dennoch bei allen Kandidaten möglich zu sein, ein solches Feature zu programmieren, wobei der Entwicklungsaufwand etwa gleich hoch eingeschätzt wird.
Ein ähnliches Bild ergab sich bezüglich der Unterstützung von DRPs und der hierzu benötigten Dokumentenreihenfolge. Keins der Frameworks bietet eine vorinstallierte Möglichkeit, die Dokumentenreihenfolge eines Dokuments zu erkennen oder bei der Erstellung von Graphen eine Reihenfolge festzulegen. Außerdem lassen sich keine bevorzugten Ports bestimmen, für die eine automatische Verknüpfung aktiviert werden könnte. Die Datenstruktur aller Frameworks scheint die Entwicklung eines solches Systems jedoch zu ermöglichen. Es besteht die Möglichkeit allen Elementen spezifische Informationen zuzuordnen, wodurch ein Regelsystem zur Generierung von DRPs und einer Dokumentenreihenfolge machbar erscheint.

Das Framework JointJS erfüllt im Vergleich die meisten Anforderungen vollständig (6 von 8) und erreicht mit 30 von 32 möglichen Punkten, die höchste Gesamtpunktzahl. Wie

bereits erwähnt, liegt JointJS mit den Frameworks GoJS, NoFlo und yFile nahezu gleich auf. Für eine einfache Entscheidung, mit welchem Framework die Entwicklung des XProc-Editors versucht werden sollte, sind die quantitativen Ergebnisse somit nicht ausreichend. Bei der näheren Betrachtung der einzelnen Aspekte zeigt sich JointJS dennoch als klarer Favorit:

JointJS ist das einzige Framework, das die hierarchische Kapselung von vornherein unterstützt und eine Model-Struktur anbietet, die für die datenbasierte und grafische Repräsentation der XProc-Steps geeignet erscheint. Bei den anderen drei Frameworks ist ungewiss, inwiefern das Feature hinzu programmiert werden kann. Da das Konzept der Kapselung zudem Auswirkungen auf die Verknüpfungsstruktur eines Graphen hat, wirkt sich die umfassende Unterstützung dieses Features positiv auf die Features „Azyklisch“ und „DRP“ aus. Mit jeweils drei Punkten ist die Möglichkeit, diese Anforderungen zu erfüllen, hier ohnehin optimistisch bewertet. Die flexible Struktur von JointJS scheint es zu ermöglichen, diese Funktionen mit überschaubarem Aufwand zu entwickeln.

Zuletzt ist hervorzuheben, dass neben NoFlo nur JointJS komplett quelloffen ist. Die nutzer*innenfreundliche Dokumentation, die aktive Wartung und Weiterentwicklung des Frameworks und eine wachsende Community, lassen auf eine nachhaltige Entwicklung mit diesem Framework hoffen.

3 JointJS

JointJS ist ein umfangreiches JavaScript-Framework, das die Erstellung von komplexen Diagrammen, Projekt- und Prozessmanagement-Tools, Workflow-Visualisierungen und vielem mehr ermöglichen soll.[52] Hierzu werden vielfältige Features und vordefinierte Elemente angeboten. Zusätzlich gibt es einige Plugins, die zu spezifischen Themen Vorlagen anbieten. JointJS wird von *client.IO*, einem IT-Unternehmen mit Sitz in Prag entwickelt und wurde 2011 in erster Version veröffentlicht. Aktuell steht es als Version 2.2.1 (Stand März 2019) zum kostenlosen und lizenzfreien Download als Open Source-Framework zur Verfügung. Neben JointJS bietet *client.IO* das kostenpflichtige Framework Rappid an, das als erweiterte Variante von JointJS verstanden werden kann. Rappid bietet neben dem gesamten JointJS-Paket einige Extra-Funktionen, insbesondere im Bereich interaktiver Bedienelemente.[53] Die mit JointJS erstellten Darstellungen basieren, nach der Generierung im Browser komplett auf SVG. Somit lässt sich JointJS auch als Tool zur dynamischen SVG-Manipulation verstehen.

Aufgrund der wenigen Quellen zu JointJS basiert die folgende Beschreibung des Frameworks fast ausschließlich auf der offiziellen Dokumentation.[54] Aufschlussreich waren zudem einige Forenbeiträge bei `stackoverflow.com`.

3.1 Abhängigkeiten und MVC-Frameworks

Um eine Anwendung mit JointJS zu erstellen, müssen zunächst die JavaScript-Frameworks jQuery, Lodash und Backbone in den Quellcode eingebunden werden.

jQuery bietet eine umfangreiche Bibliothek mit Befehlen zur Vereinfachung von komplexen JavaScript-Funktionen.[55] Insbesondere im Bereich des Event-Handling und der Animation hat jQuery viel Popularität erlangt und auch für Datenabfragen mit Ajax bietet jQuery viele Möglichkeiten. Innerhalb von JointJS ist das initiale $-Zeichen von jQuery-Befehlen insbesondere in der Ansprache von Objekten für Entwickler*innen sichtbar. Ebenso folgt die Syntax von JointJS-Events der von jQuery.

Durch die Implementierung von Befehlen der Bibliothek von **Lodash.js** sollen JavaScript-Funktionen kürzer und einheitlicher werden, was insbesondere dazu dient, die Wartung des Codes zu vereinfachen.[56] Dies geschieht vor allem durch die einfachere Verwaltung von Objekten, Strings und Arrays. Lodash-Befehle sind durch den vorangestellten Unterstrich erkennbar. Lodash ist eine Weiterentwicklung des Frameworks Underscore.js,

52 Vgl. [JointJS – OpenSource]
53 Vgl. [JointJS – Rappid]
54 Vgl. [JointJS – Docs]
55 Vgl. [jQuery – API]
56 Vgl. [Lodash]

welches dieselbe Syntax verwendet.[57] Bei der Nutzung von JointJS kommen Lodash-Befehle erst bei komplexeren Anwendungen vor und dienen zunächst der Strukturierung des JointJS-Quellcodes.
Auch **Backbone.js** dient in erster Linie der besseren Strukturierung von JavaScript-Anwendungen.[58] Hier liegt der Fokus auf der Etablierung eines systematischen Datenaustauschmodells nach dem **Model View Controller** (MVC)-Konzept, auf das im nächsten Absatz genauer eingegangen wird. In JointJS wird Backbone besonders für die Erstellung und Erweiterung grafischer Elemente genutzt. Außerdem tritt Backbone in der Konvertierung und Manipulation der Objekt-Daten in Erscheinung. So gibt es z. B. eine an die Backbone-Models angepasste Version des JavaScript-Befehls `.toJSON()` zur Erzeugung eines JSON-Strings.

Wie Lodash und Backbone ist auch JointJS ein sogenanntes **MVC-Framework**. MVC ist ein seit 1979 existierendes theoretisches Konzept der Programmierung, nach dem der Code einer Programmiersprache bzw. die resultierende Applikation in die drei Bereiche Model, View und Controller unterteilt wird. (Vgl. Ackermann 2015: 423f.) **Model** steht dabei für die Daten bzw. das Datenmodell der Applikation oder einzelner Elemente. **View** umfasst die sichtbare Darstellung der Elemente, z. B. in einem Browser. Der **Controller** ist „[…] eine Art Steuerungseinheit […].“ (Steyer 2017: 7) Er macht die Trennung von Model und View überhaupt erst möglich, indem er zwischen ihnen vermittelt und Informationen weitergibt. Wenn (Front-End-)Nutzer*innen z. B. auf einer Webseite in einem Suchfeld (View) Daten abfragen, wird die Abfrage über eine Funktion (Controller) beispielsweise an hinterlegten JSON-Daten (Model) geprüft und das Ergebnis anschließend über die Funktion (Controller) an eine Ergebnisauflistung (View) zurückgegeben. Ackermann (2015: 424) schreibt zu den Vorteilen dieser Trennung von Daten und Ansicht: „Dies […] hat zur Folge, dass relativ einfach verschiedene Ansichten bzw. Views für die gleichen Daten implementiert werden können, ohne dass das Datenmodell angepasst werden muss. Hinzu kommt, dass das Datenmodell viel einfacher isoliert getestet werden kann.“
Im Quellcode des XProc-Editors wird die Anwendung des MVC-Ansatzes sehr deutlich. In den folgenden JointJS-Code-Beispielen werden immer wieder Datenmodelle (Model) an eine grafische Darstellung (View) übergeben. Das folgende Schaubild (Abb. 16) zeigt ein typisches MVC-Schema, nachdem in JointJS vorgegangen werden kann und welches auch in der Entwicklung des XProc-Editors angewendet wurde. Im weiteren Verlauf der Arbeit werden einzelne Aspekte hierzu noch deutlicher.

57 Vgl. [UnderscoreJS]
58 Vgl. [BackboneJS]

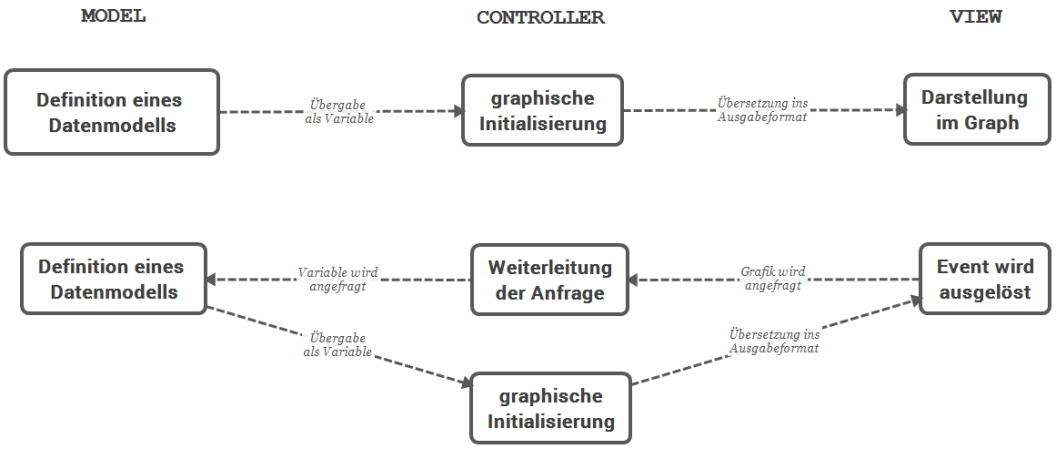


Abb. 16: MVC-Schema

3.2 Aufbau und Elemente

Da JointJS der Umsetzung verschiedenster Projekte dienen soll, befinden sich im Framework-Baukasten Formen, Modelle, Befehle und Events, die auf vielfältige Weise miteinander kombiniert werden können. Dabei ist JointJS sehr flexibel und es lassen sich auch als festgelegt erscheinende Elemente bei tieferem Verständnis der eingebundenen Frameworks (siehe Kapitel 3.1) an die eigenen Bedürfnisse anpassen.

Die offizielle Dokumentation von JointJS ist sehr umfangreich und bietet zu vielen Elementen Code- und Anwendungsbeispiele.[59] Einen Überblick zu bekommen, wie JointJS strukturiert ist, und zu verstehen, welche Abhängigkeiten zwischen Elementen, Events etc. bestehen, kann einige Zeit in Anspruch nehmen. Die vorhandenen Demos[60] und Tutorials[61] helfen dabei, die theoretischen Grundlagen mit praktischen Anwendungsbeispielen zu verknüpfen. Seit kurzem (Stand: März 2019) gibt es die offizielle Dokumentation von JointJS auch als interaktive Mindmap.[62] Hierdurch lässt sich die Struktur leichter nachvollziehen und Elemente und Befehle erscheinen in ihrem Kontext.

Die folgende, selbst erstellte Mindmap (Abb. 17) zeigt eine Übersicht der im XProc-Editor häufig verwendeten JointJS-Elemente. Es sei vorab erwähnt, dass die in der Mindmap immer wieder vorkommende Vorsilbe `dia` für ‚diagram' steht. Sie wird im geschriebenen Code nur in bestimmten Fällen vor das eigentliche Element gesetzt und dient in der offiziellen Dokumentation der Strukturierung. Ebenso ist `dia.Element` lediglich ein Platzhalter für verschiedene mögliche JointJS-Elemente.

59 Vgl. [JointJS – API]
60 Vgl. [JointJS – Demos]
61 Vgl. [JointJS – Tutorial]
62 Vgl. [JointJS – MindMap]

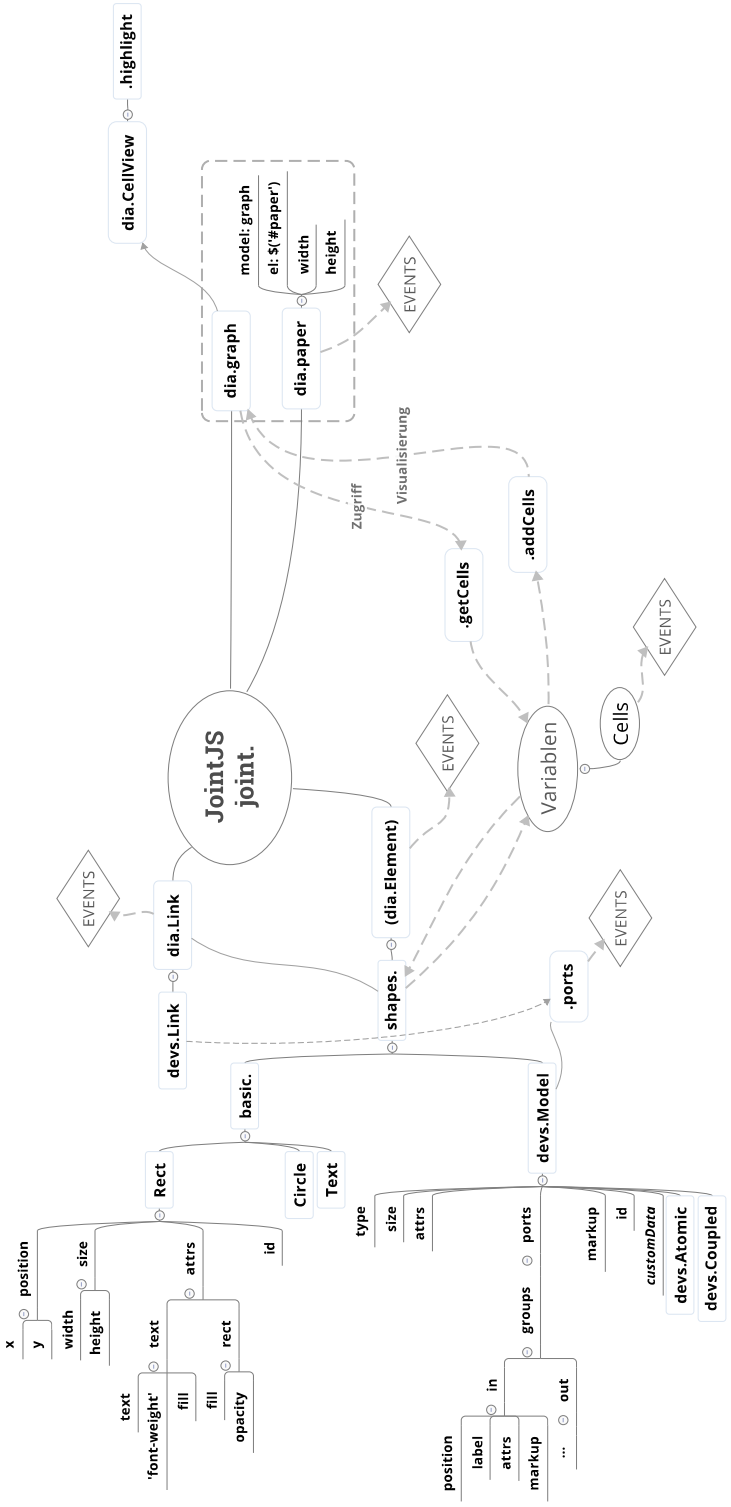


Abb. 17: Struktur von JointJS

Alles was mit JointJS im Browser angezeigt wird, erscheint auf einem sogenannten Paper. Das Paper ist die sichtbare Fläche eines unendlich großen Graph, der immer aufgerufen werden muss, bevor das zugehörige Paper erzeugt werden kann. Innerhalb einer Webseite

können mehrere Paper (mit dazugehörigem Graph) erstellt werden. Jedes Paper braucht zur Initialisierung einen Platzhalter im HTML-Dokument. In der Regel wird hierzu einem div-Element eine ID zugeordnet, welche im Datenmodell des Papers angesprochen wird.[63] Das folgende Code-Beispiel zeigt eine simple Paper-Initialisierung mit entsprechender Zuordnung im HTML-Code:

```
<html lang="en">
<head>
  <!-- Missing Code -->
</head>
<body>
<div id="paper"></div>
<script>
  let graph = new joint.dia.Graph,
    paper = new joint.dia.Paper({
      el: $('#paper'),
      model: graph,
      width: 1200,
      height: 850
    });
</script>
</body>
</html>
```

Code-Beispiel 7 : Initialisierung eines JointJS-Paper

In der Datenstruktur des Paper-Objekts wird der Graph dem Paper anhand des Wertes von `model` zugeordnet. Gäbe es ein weiteres Paper, bräuchte dieses einen zusätzlichen Graph.

JointJS wird häufig für die Erstellung interaktiver grafischer Formen genutzt. Diese Shapes bringen mit sich, dass sie auf einem Paper bzw. im Graph frei bewegt werden können. Shapes können simple Formen wie Rechtecke oder Kreise sein, Text repräsentieren oder auch als komplexes Modell auftauchen. Für die Entwicklung des XProc-Editors ist das devs.Model[64] von besonderer Bedeutung. Ein devs.Model ist seiner Form nach ein einfaches Rechteck, welches jedoch einige Besonderheiten aufweist. Einem devs.Model können beliebig viele Ports[65] zugeordnet werden, welche sich als In- oder Output-Ports definieren lassen und einer Seite des Rechtecks (left, right, up, down) zugeordnet werden können. Über die Ports lassen sich devs.Model miteinander verbinden. Die Verbindung geschieht über einen Link[66]. Sie wird entweder im Code durch die Angabe von Port-IDs definiert oder die Elemente werden in der grafischen Darstellung per Drag & Drop verbunden.
Das devs.Model erfüllt mehrere Grundbedingung von XProc-Pipelines und hält scheinbar

63 Vgl. [JointJS – Docs #PaperConstructor]
64 Vgl. [JointJS – Docs #ShapesDevs]
65 Vgl. [JointJS – Docs #Ports]
66 Vgl. [JointJS – Docs #Link]

alle Elemente bereit, die für die Darstellung von XProc-Steps benötigt werden. Als untergeordnete Elemente von devs.Model hält JointJS die Modelle devs.Atomic und devs.Coupled bereit. devs.Atomic entspricht in seiner Funktionalität einem Atomic Step und devs.Coupled einem Compound Step. Wie bei einem Compound Step lassen sich innerhalb von devs.Coupled in beliebiger Tiefe weitere devs.Coupled sowie devs.Atomic verschachteln. Dies geschieht nach grafischer Initialisierung der Modelle über die Funktion element.embed(cell)[67]. Ein so eingebundenes Model wird in der grafischen Darstellung an das ‚Elternelement' gekoppelt, sodass es bei dessen Bewegung in gleicher Weise mit bewegt wird. Der folgende Screenshot von einer interaktiven JointJS-Demo[68] zeigt den schematischen Aufbau miteinander verbundener und ineinander verschachtelter devs.Model:

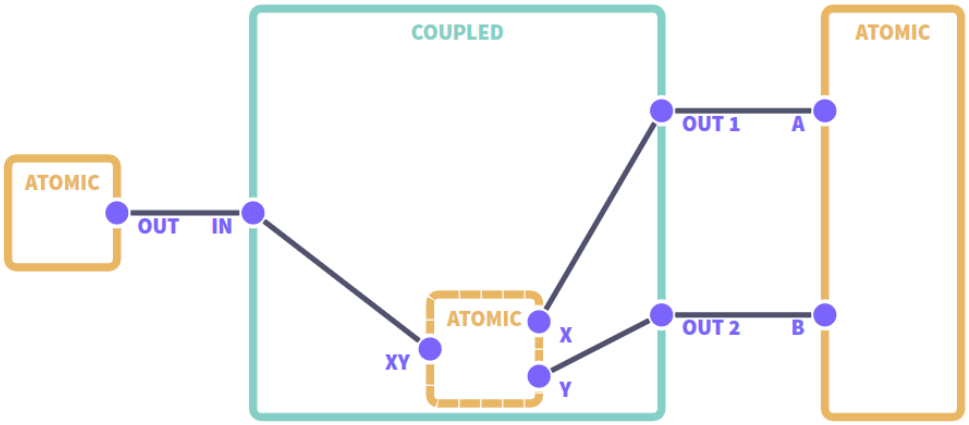


Abb. 18: Schaubild des devs.Model in JointJS

Das devs.Model lässt sich, wie bei allen JointJS-Elementen möglich, über die Funktionen .extend() bzw. .define() erweitern und es lassen sich eigene Modelle mit spezifischen Metadaten erstellen. Für die Entwicklung des XProc-Editors ist diese Erweiterbarkeit elementar (siehe Kapitel 4.2).

3.3 Datenstruktur und Initialisierung

Für die grafische Erzeugung von JointJS-Elementen wird, wie für das Paper in Code-Beispiel 7, zunächst ein JavaScript-Objekt definiert. Das Datenmodell kann verschiedene Informationen zu dem Element beinhalten, z. B. Position, Größe, ID, Beschriftung oder eigene Style-Definitionen. Die möglichen Angaben sind in den verschiedenen JointJS-

67 Vgl. [JointJS – Docs #ElementEmbed]
68 Vgl. [JointJS – Demos #Devs]

Vorlagen festgelegt. Über das Schlüsselwort `new` und die Referenz auf eine JointJS-Vorlage (z.B. joint.shapes.basic.Rect) wird ein Element initialisiert und einer selbst gewählten Variable zugeordnet. Die Variable kann anschließend von der Funktion graph.addCell() aufgerufen werden, um das Element in den Graph zu laden. Das folgende Code-Beispiel zeigt das Datenmodell eines Texts und eines Rechtecks mit anschließender Initialisierung in den Graph:

```
let text1 = new joint.shapes.basic.Text({
  position: {x: 20, y: 310},
  size: {width: 140, height: 30},
  attrs: {
    text: {text: 'Ein Text', 'font-weight': 'bold', 'font- family':
'Courier'}
  }
});
let rect1 = new joint.shapes.basic.Rect({
  position: {
    x: 20,
    y: 60
  },
  size: {
    width: 140,
    height: 20
  },
  attrs: {
    text: {text: 'Ein Rechteck', 'font-weight': 'bold', fill: 'white'},
    rect: {fill: '#a876ff', rx: '10', ry: '10', opacity: 0.8}
  }
});
graph.addCells([text1, rect1]);
```

Code-Beispiel 8 : Initialisierung von Text und Rechteck in JointJS

Bei der grafischen Erzeugung wird das definierte Element zu einer Cell[69]; einer Zelle des Graph. Diese wird im sichtbaren Paper bzw. in der dynamischen HTML-Struktur als SVG-Grafik erzeugt. Die übergebenen Daten bleiben dabei in der Cell erhalten und können durch die Funktion graph.getCell() wieder abgerufen werden. Dabei ist zu beachten, dass auf diesem Weg ein JavaScript-Objekt der gesamten Cell erstellt wird, das neben den selbst generierten Daten, viele grafische und dynamisch-kontextbezogene Informationen beinhaltet, die nicht immer von Interesse sind. Um die selbst erstellten Daten aus dem Cell-Objekt rauszufiltern kann über die JavaScript-Funktion `element.toJSON()` das initiale Daten-Modell aus dem Cell-Objekt ermittelt werden.[70] Das folgende Schaubild (Abb. 19) stellt das Cell-Objekt eines simplen Kreises dessen initialem Objekt gegenüber:

69 Vgl. [JointJS – Docs #Cell]
70 Vgl. [JointJS – Docs #GraphToJSON]

Cell - Objekt


```
{…}
  _changing: false
 ▸ _events: Object { "change:ports": (2) […],
"change:attrs": (2) […], all: (1) […], … }
  _listenId: "l15"
 ▸ _listeners: Object { l16: {…} }
  _pending: false
 ▸ _portSettingsData: Object { ports: [],
groups: {}, portLayoutNamespace: {…}, … }
 ▸ _previousAttributes: Object { type:
"basic.Circle", angle: 0, id: "f2279010-938c-
41ab-bdd8-3b833cf99aa5", … }
 ▸ _transitionIds: Object {  }
 ▾ attributes: {…}
     angle: 0
   ▸ attrs: Object { circle: {…}, text: {…},
   ".": {…} }
     id: "f2279010-938c-41ab-bdd8-3b833cf99aa5"
   ▸ position: Object { x: 50, y: 50 }
   ▸ size: Object { width: 100, height: 100 }
     type: "basic.Circle"
     z: 2
   ▸ <prototype>: Object { … }
 ▸ changed: Object { z: 2 }
  cid: "c9"
 ▸ collection: Object { length: 2, models: (2)
[…], _byId: {…}, … }
 ▸ graph: Object { cid: "c5", _changing: false,
_pending: false, … }
  id: "f2279010-938c-41ab-bdd8-3b833cf99aa5"
 ▸ ports: Object {  }
 ▸ <prototype>: Object { defaults: {…}, markup:
"<g class=\"rotatable\"><g class=\"scalable
\"><circle/></g><text/></g>", constructor:
child() ↱≡ }
```


Initiales Objekt


```
{…}
  angle: 0
 ▸ attrs: Object { text: {…} }
  id: "f2279010-938c-41ab-bdd8-3b833cf99aa5"
 ▸ position: Object { x: 50, y: 50 }
 ▸ size: Object { width: 100, height: 100 }
  type: "basic.Circle"
  z: 2
 ▸ <prototype>: Object { … }
```


Abb. 19: Unterschied zwischen Cell-Objekt und initialem Objekt (Screenshot)

Wie hier zu sehen ist, wird durch `element.toJSON()` kein JSON-String, sondern ein JavaScript-Objekt erzeugt. Über die Funktion `JSON.stringify()` könnte das Objekt in einen JSON-String übersetzt werden und dieser String über die Funktion `JSON.parse()` zurück in ein JavaScript-Objekt. Diese Funktionen wurden in der Entwicklung des XProc-Editors sehr häufig verwendet (siehe Kapitel 4.2).

Das Cell-Objekt ist insbesondere für die dynamische Daten-Interaktion nützlich und sollte hierzu ohne Veränderung angesprochen werden. Für die Entwicklung und Darstellung von Anwendungen mit JointJS, ist das Konzept der Cell elementar. Es basiert auf einem erweiterten Backbone-Model (siehe Kapitel 3.1) und erfüllt neben der Funktion als Daten-Container vielfältige Aufgaben. Alle JointJS-Elemente werden bei ihrer Initialisierung in den Graphen als Cell erzeugt. Im folgenden Kapitelabschnitt wird zudem auf die Funktion der cellView eingegangen, einem wichtigen Funktionsparameter, der die Ansicht einer Cell im Browser als JavaScript-Objekt wiedergibt und in den meisten JointJS-Events Anwendung findet.

3.4 Events

Es gibt sehr viele Möglichkeiten, wie mit JointJS Events erstellt werden können. Die Nutzung der Framework-eigenen Event-Struktur hat den Vorteil, dass hierdurch auf die Daten bzw. Objekte der JointJS-Elemente sehr leicht zugegriffen werden kann. Das folgende Code-Beispiel zeigt eine Funktion, mit der beim Doppelklick auf eine Cell des Papers (z. B. ein Rechteck), dem Graph eine Kopie dieser Cell hinzugefügt werden soll.

```
paper.on('cell:pointerdblclick', function(cellView, evt, x, y){
  let elem = cellView.model;
  let newPosX = x + 50;
  let newPosY = x + 20;
  graph.addCell(elem.clone().position(newPosX, newPosY));
});
```

Code-Beispiel 9 : Doppelklick-Funktion auf einem JointJS-Paper

An diesem Beispiel erkennt man sehr gut die Grundstruktur eines JointJS-Events. Nachdem der Kontext des Events (hier: Paper) angesprochen wurde, wird das Element genannt, das auf ein Ereignis reagieren soll (hier: Cell). Über einen Doppelpunkt getrennt wird das Ereignis selbst beschrieben (hier: Pointerdblclick). Im anschließenden Funktionsaufruf eines solchen Events sind standardmäßig vier Parameter enthalten. Diese sind mit dem auslösenden Element und dem Ereignis verknüpft und geben wichtige Daten an die Funktion weiter.
Die sogenannte cellView[71] ist die datenbasierte Repräsentation des sichtbaren Elements im Paper. Das cellView-Objekt unterscheidet sich dabei sowohl vom Ergebnis des Aufrufs graph.getCell(), als auch vom initialen Objekt des Element-Aufrufs, wie es im vorherigen Abschnitt gegenübergestellt wurde (siehe Abb. 19). Das cellView-Objekt des dort erzeugten Kreises, sähe folgendermaßen aus:

71 Vgl. [JointJS – Docs #CellView]

cellView - Objekt

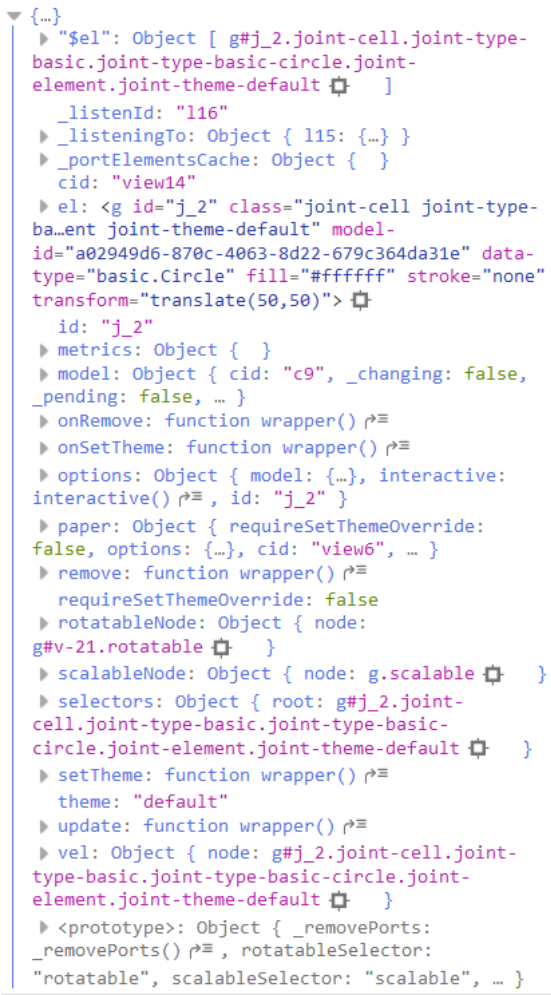


Abb. 20: Screenshot eines CellView-Objekts

Um dieses Objekt außerhalb eines Events abzurufen, hält JointJS die Funktion paper.findViewByModel(cell) bereit. An diesem Funktionsaufruf fällt auf, dass das cellView-Objekt nicht aus dem Graph, sondern aus dem Paper, also dem sichtbaren Bereich des Graph erzeugt wird. In der Objektstruktur lässt sich dies mitunter an mehreren Instanzen der SVG-Repräsentation erkennen, die hier (z.B. unter el) aufgeführt sind. Im folgenden Abschnitt wird auf die SVG-Implementierung genauer eingegangen. Im XProc-Editor ist das cellView.model eines der am häufigsten abgefragten Unterobjekte des cellView-Objekts. Es ist identisch mit dem Cell-Objekt (siehe Abb. 19) und somit könnte von hier aus wieder auf die initialen bzw. selbst generierten Daten zugegriffen werden.

Das cellView-Objekt der Doppelklick-Funktion in Code-Beispiel 9 wird also an die Funktion übergeben. Dort wird anhand der Variable elem und dem zugeordneten cellView.model, also dem Cell-Objekt, die Vorlage für das zu kopierende Element erstellt.

Das neue Element soll ein Stück versetzt zur Position des Mauszeigers beim Doppelklick erstellt werden. Die Funktionsparameter x und y geben die Position des Ereignisses an die Funktion weiter. In den Variablen newPosX und newPosY werden anhand dieser Parameter die neuen (SVG-)Koordinaten errechnet. Über die JointJS-Funktionen element.clone() und element.position() wird das Element letztlich an der neuen Position in den Graph geladen. Das Klonen von Elementen bzw. die Erzeugung klonbarer Vorlagen ist eine wesentliche Funktion des XProc-Editors und taucht somit in der Beschreibung des Entwicklungsprozesses an mehreren Stellen auf (siehe Kapitel 4).
Neben cellView, x und y, ist der Funktionsparameter evt in dem Beispiel mit angegeben, er wird in der Funktion jedoch nicht verwendet. Die Abkürzung evt steht für ‚event' und dieser Parameter übergibt der Funktion ein komplexes Event-Objekt, aus dem der Auslöser des Events, Informationen zum Zielobjekt und vieles mehr entnommen werden können. Abb. 21 zeigt ein Event-Objekt, das beim Klick auf ein Paper entstanden ist.

Event - Objekt

Abb. 21: Screenshot eines Event-Objekts

Events werden hauptsächlich im Kontext der Paper ausgelöst.[72] Alle Bewegungen und Tasten der Maus können als Auslöser des Events verwendet werden. Außerdem können auch Veränderungen der JointJS-Elemente, z. B. die veränderte Position einer Cell oder das Auswählen einer Cell durch die Funktion cellView.highlight(), Auslöser für ein Event sein. Neben den Cells können Events z. B. auch spezifisch an Link-Elemente gebunden werden.

72 Vgl. [JointJS – Docs #PaperEvents]

3.5 SVG-Implementierung

Bei der Initialisierung eines JointJS-Elements wird dem angesprochenen Elementknoten im HTML-DOM ein neuer SVG-Elementknoten angehangen. Anhand der Entwicklerwerkzeuge des Browsers lässt sich diese SVG-Implementierung gut nachvollziehen. Bezugnehmend auf Code-Beispiel 8, zeigt die folgende Abbildung die grafische Repräsentation eines Rechtecks und eines Texts in einem Paper und nebenstehend die erzeugte Code-Struktur der SVG-Elemente:

Abb. 22: Grafische Initialisierung von JointJS-Elementen und HTML-Struktur (Screenshot)

Die SVG-Elemente werden innerhalb des Paper-divs und nach zwei div-Platzhaltern (für den Hintergrund und ein optionales Orientierungs-Gitter) eingefügt. Rechteck und Text werden in einer SVG-Gruppe mit der Klasse joint-viewport zusammengefasst. Innerhalb dieser Gruppe erhalten sie wiederum jeweils eine eigene Gruppe, die je nach Komplexität des Elements mehr oder weniger verschachtelt ist. Neben den vielen vordefinierten CSS-Klassen von JointJS, lassen sich den Elementen bei ihrer Initialisierung auch eigene CSS-Klassen zuordnen, sodass Style-Informationen in eine CSS-Datei ausgelagert und für mehrere Elemente gleichzeitig definiert werden können.

In der Initialisierung von JointJS-Elementen (siehe Kapitel 3.3) kann das vordefinierte SVG-

Markup bearbeitet werden. Hierdurch lässt sich tief in den inneren Aufbau der SVG-Repräsentation eingreifen und es sollte darauf geachtet werden, dass die Grundstruktur erhalten bleibt, da die Darstellung ansonsten möglicherweise fehlschlägt. In der Entwicklung des XProc-Editors wird in das SVG-Markup eingegriffen, um das Aussehen der Ports zu verändern und ihnen eigene CSS-Klassen zuzuweisen. Außerdem werden durch die Manipulation des SVG-Markups der devs.Model spezifische Hover-Elemente erzeugt (siehe Kapitel 4.2 und 4.6).

4 Entwicklung des XProc-Editors

Die Entwicklung des XProc-Editors begann im April 2018. Im folgenden Abschnitt wird die Entwicklungsgeschichte kurz zusammengefasst. Die Ausführungen in den weiteren Abschnitten dieses Kapitels beziehen sich hauptsächlich auf den aktuellen Entwicklungsstand (März 2019).

4.1 Grundkonstruktion und Entwicklungsgeschichte

Die benötigten Hauptbereiche des Editors und die grobe Anordnung waren am Anfang des Entwicklungsprozesses schnell klar: Im Zentrum müsste die zu bearbeitende XProc-Pipeline stehen. Zudem bräuchte es einen Bereich, in dem die Step-Libraries gesammelt werden und von dem aus Steps in die Pipeline geladen werden können. Und schließlich bräuchte es ein Feld, das es möglich macht, die (noch) unsichtbaren Einstellungen (Attributwerte, Bezeichnungen, Port-Informationen etc.) sowohl der Haupt-Pipeline, als auch aller eingebundenen Steps, anzuzeigen und zu bearbeiten. Die folgenden Abbildungen zeigen die Entwicklungsstufen dieser ersten Entwurfs-Phase.

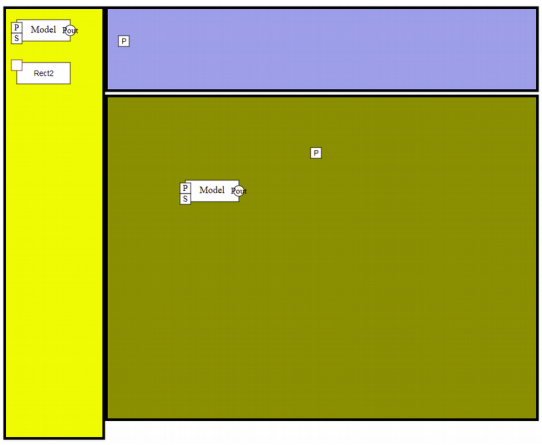


Abb. 23: Screenshot vom 3.5.2018

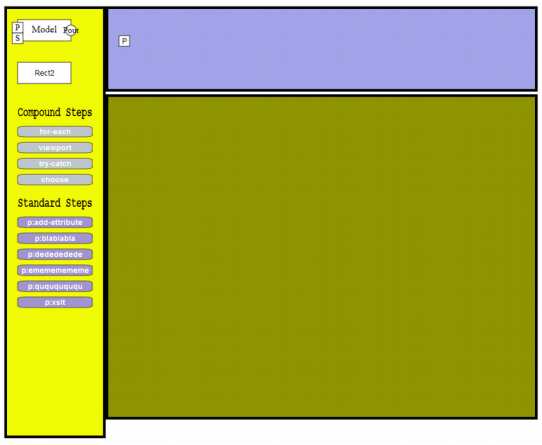


Abb. 24: Screenshot vom 4.5.2018

Der in den Abbildungen gelbe Bereich beinhaltet die Step-Libraries und wurde dementsprechend schnell als **Step-Panel** bezeichnet, wie auch in der folgenden Beschreibung. Der blaue, obere Bereich wird in den folgenden Abbildungen noch deutlicher als Bereich zur Bearbeitung der Metadaten erkennbar werden. Im Folgenden wird er **Meta-Panel** genannt. Der grüne Bereich im Zentrum, in dem die XProc-Pipeline bearbeitet werden kann, wird **Hauptbereich** genannt. Wie in den Abbildungen zu

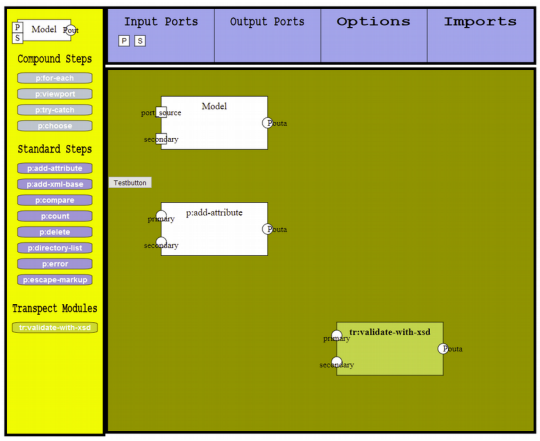


Abb. 25: Screenshot vom 8.5.2018

erkennen ist, lag in dieser Entwicklungsphase eine Kernaufgabe in der Erarbeitung eines grafischen und datenbasierten Modells für die XProc-Steps. In Anlehnung an Datenflussdiagramme und um den Seitenverhältnissen genutzter Endgeräte (siehe Kapitel 4.3) zu entsprechen, wurde sich für eine von links nach rechts fließende Pipeline entschieden. Die Input-Ports wurden somit an der linken Seite eines Steps angeordnet und die Output-Ports an dessen rechter Seite. Für Atomic- und Compound-Steps wurden anschließend zwei verschiedene Modelle entwickelt. Zudem musste eine Möglichkeit gefunden werden, die Steps aus dem Step-Panel in den Hauptbereich zu laden. Hierfür wurden zwei Funktionen geschrieben, die das Laden der Steps per Doppelklick und Drag & Drop ermöglichten.

Das Meta-Panel wurde in einer anschließenden Überarbeitung rechts vom Hauptbereich angeordnet, um mehr Platz für die XProc-Pipeline zu schaffen und den Gewohnheiten von User*innen in der Bedienung grafischer Editoren zu entsprechen. In Abb. 26 ist diese neue Anordnung zu sehen. Außerdem werden hier innerhalb des Meta-Panels (in einem neongrünen div-Element) zum ersten Mal Informationen zu einem Step angezeigt. Weiterhin ist zu sehen, dass die erzeugten Steps im Hauptbereich nun eine farbliche Zuordnung zu den Elementen der Step-Library haben.

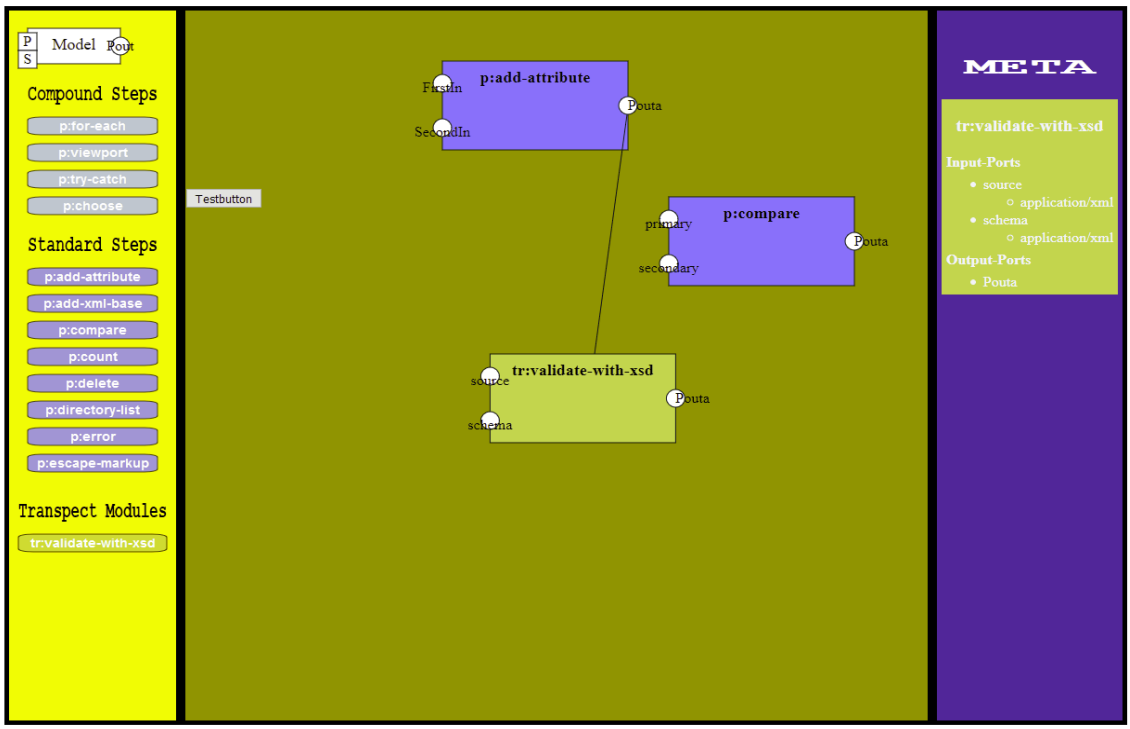


Abb. 26: Screenshot vom 23.5.2018

Im nächsten Entwicklungsschritt lag der Fokus insbesondere auf der Erarbeitung eines funktionalen XProc-Modells. Die zentrale Herausforderung war, eine Architektur zu generieren, die es möglich macht, Steps in eine Pipeline einzufügen. Es wurde ein

spezielles JointJS-Modell für die Haupt-Pipeline erarbeitet, das von nun an beim ersten Laden des Editors automatisch dem Hauptbereich hinzugefügt wurde. Alle anschließend in den Hauptbereich geladenen Steps, wurden automatisch in die Haupt-Pipeline eingebettet. Abb. 27 zeigt dieses neue Pipeline-Konzept:

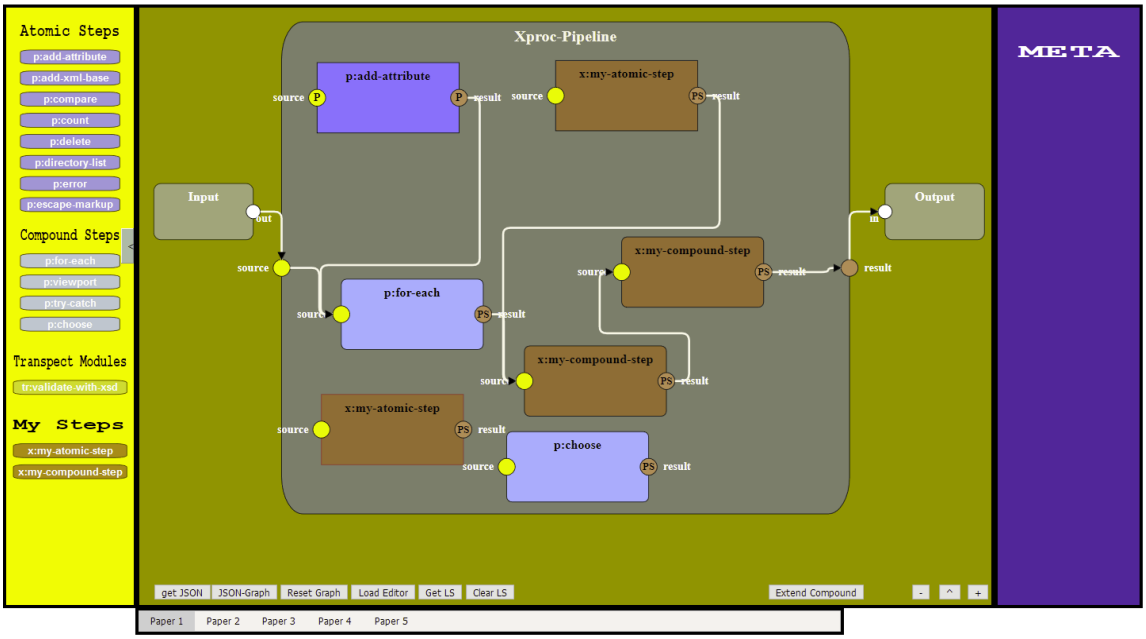


Abb. 27: Screenshot vom 16.7.2018

Um einem weiteren wesentlichen Aspekt der Pipeline-Erstellung gerecht zu werden, wurde ein grafisches Modell für selbst generierte Steps entwickelt, von denen einige Instanzen in der Haupt-Pipeline (Abb. 27) zu sehen sind. Im Step-Panel sieht man außerdem die neue Step-Library „My Steps“, aus der die eigenen Steps geladen werden. Im Hauptbereich ist nun auch eine erste Idee zum Bereitstellen von Input- und Output-Daten für die Pipeline zu sehen. Im Vergleich zu Abb. 26, ist in Abb. 27 zudem zu erkennen, dass sich die Verbindungslinks zwischen den Steps bzw. Ports in Farbe, Linienstärke und der Linienführung geändert haben. Hierzu wurde ein eigenes JointJS-Link-Modell erstellt, dem der Typ manhattan zugeordnet wurde, der es ermöglicht ‚im Weg liegende‘ Elemente zu umfließen. Des Weiteren ist in Abb. 27 neben einigen Funktionsbuttons, die vor allem der Entwicklung dienen, ein weiterer Aktionsbereich hinzugekommen: Die Leiste unterhalb des Hauptbereichs ermöglicht nun den Wechsel zwischen mehreren Arbeitsflächen bzw. JointJS-Papers. Zunächst sollte hierdurch ermöglicht werden, mehrere Pipelines parallel zu bearbeiten. In der weiteren Entwicklung hat sich diese Funktion leicht verändert, worauf in diesem Kapitel nochmals eingegangen wird. In der weiteren Beschreibung wird der neue Aktionsbereich **Paper-Switch** genannt. In Abb. 27 ist zudem ein Versuch zu erkennen, wichtige Eigenschaften der Ports bereits in

den entsprechenden Elementen im Hauptbereich kenntlich zu machen. Innerhalb der Input- und Output-Ports der Haupt-Pipeline wird hier mit einem ‚P' und einem ‚S' darauf hingewiesen, ob der Port ein Primary-Port ist und ob sein `sequence`-Wert auf `true` gesetzt ist.

In einem nächsten Entwicklungsschritt wurde für die XProc-Options ein eigenes Modell entwickelt (siehe Abb. 28). Beim Laden eines Steps in den Hauptbereich werden nun aus den Metadaten der Steps die Options ausgelesen und als grafische Elemente an den jeweiligen Step gebunden. Wird ein Step bewegt, bewegen sich die Options mit und können zudem über eigene Ports verbunden werden.

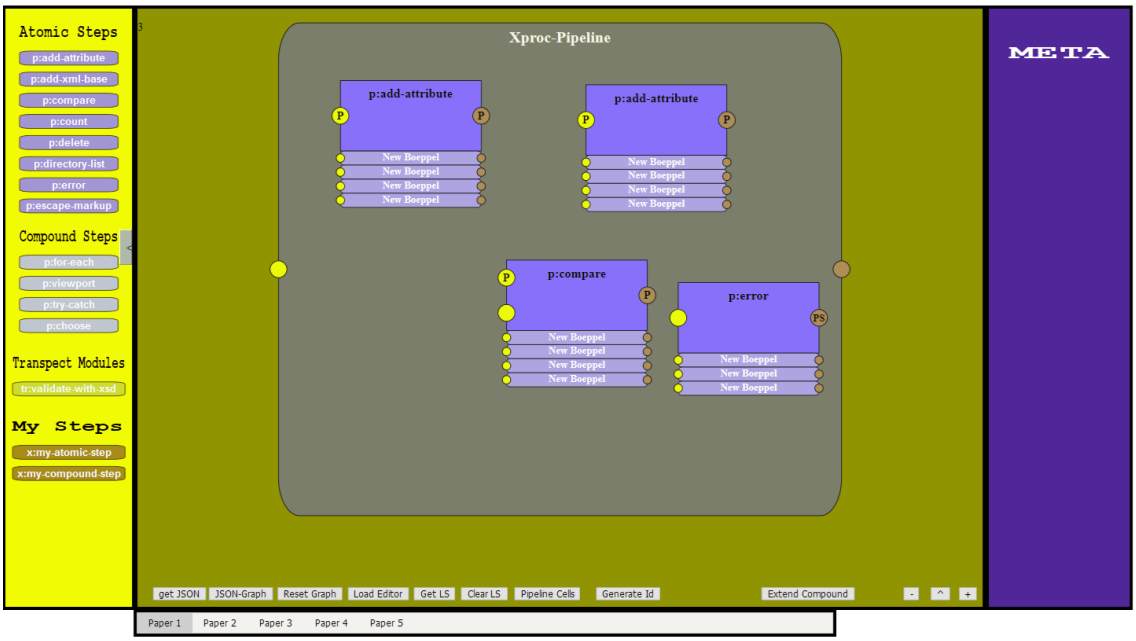


Abb. 28: Screenshot vom 17.7.2018

Die Interaktion mit den Metadaten der Steps bzw. der Haupt-Pipeline war eine immer wieder in den Fokus rückende Herausforderung. Hierbei gab es verschiedene Versuche, eine Interaktion zwischen der HTML-Repräsentation im Meta-Panel und den Daten der Steps zu ermöglichen. Hierauf wird in Abschnitt 4.2 genauer eingegangen. Gleichzeitig mussten die vielen Informationen und Metadaten eines Steps innerhalb des Meta-Panels untergebracht werden. Hierzu wurde die Gestaltung des Panels stetig angepasst. In Abb. 29 hat das Meta-Panel erste Schaltflächen zur Daten-Interaktion. Hier werden nun auch die Options eines Steps als aufklappbare Elemente aufgeführt. In der grafischen Darstellung werden mittlerweile die korrekten Namen der Options angezeigt. Diese werden beim Laden der Steps aus den Metadaten gelesen und in die Kopien des Option-Modells geschrieben.

Eine weitere Neuigkeit sind die Zahlen innerhalb der Step-Elemente. Diese IDs werden

hier zwar zum ersten Mal angezeigt, im Code mussten sie jedoch bereits in einem frühen Entwicklungsstadium generiert werden, um die Kopien der Step-Modelle eindeutig zu identifizieren und ebenso die Abhängigkeiten der Options zuzuordnen.

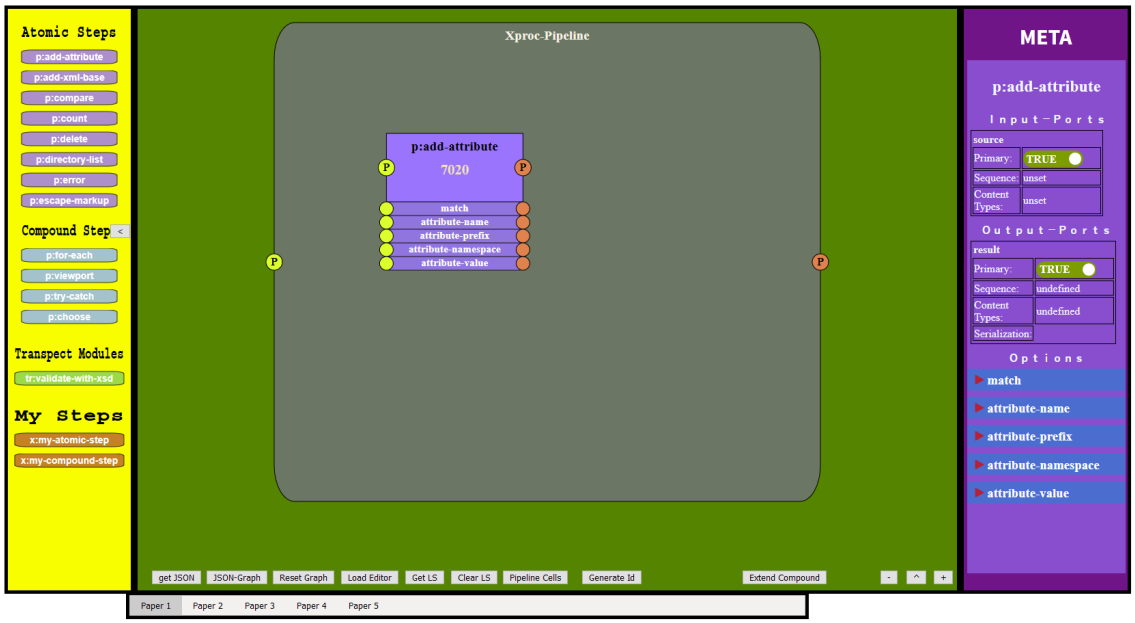


Abb. 29: Screenshot vom 25.10.2018

Der bisher beschriebene Entwicklungsstand enthält bereits viele Grundelemente der aktuellen Version des XProc-Editors. Zwischen November 2018 und Februar 2019 gab es dann erneut eine umfangreiche Überarbeitung aller Bereiche. Das Step-Panel wurde in eine dynamische HTML-Struktur überführt, wobei die Step-Libraries eine Suchfunktion erhielten und die Funktionen für das Laden der Steps neu programmiert wurden. Zuvor waren alle Inhalte des Step-Panels sowie das Step-Panel selbst als SVG-basierte JointJS-Elemente repräsentiert. Dies hatte zwar Vorteile für das Laden der Steps und für die responsive Gestaltung des Editor, jedoch gab es von Beginn an ein generelles Platzproblem und die Entwicklung eines User Interfaces (UI) auf SVG-Basis erschien zu umständlich. Bei der Überarbeitung des Meta-Panels wurden die Interaktionsmöglichkeiten mit den Metadaten stark erweitert. Es wurde ein Konfigurations-Formular erarbeitet, mit dem es nun möglich ist, Pipelines zu benennen, Ports hinzuzufügen und sie zu entfernen, die Eigenschaften der Ports anzupassen und außerdem neue Options hinzuzufügen oder diese zu entfernen.

Darüber hinaus wurde für den gesamten Editor ein neues Gestaltungskonzept entworfen, bei dem eine responsive Grundstruktur und ein intuitives GUI im Vordergrund standen. Der folgende Screenshot zeigt die beschriebenen Änderungen:

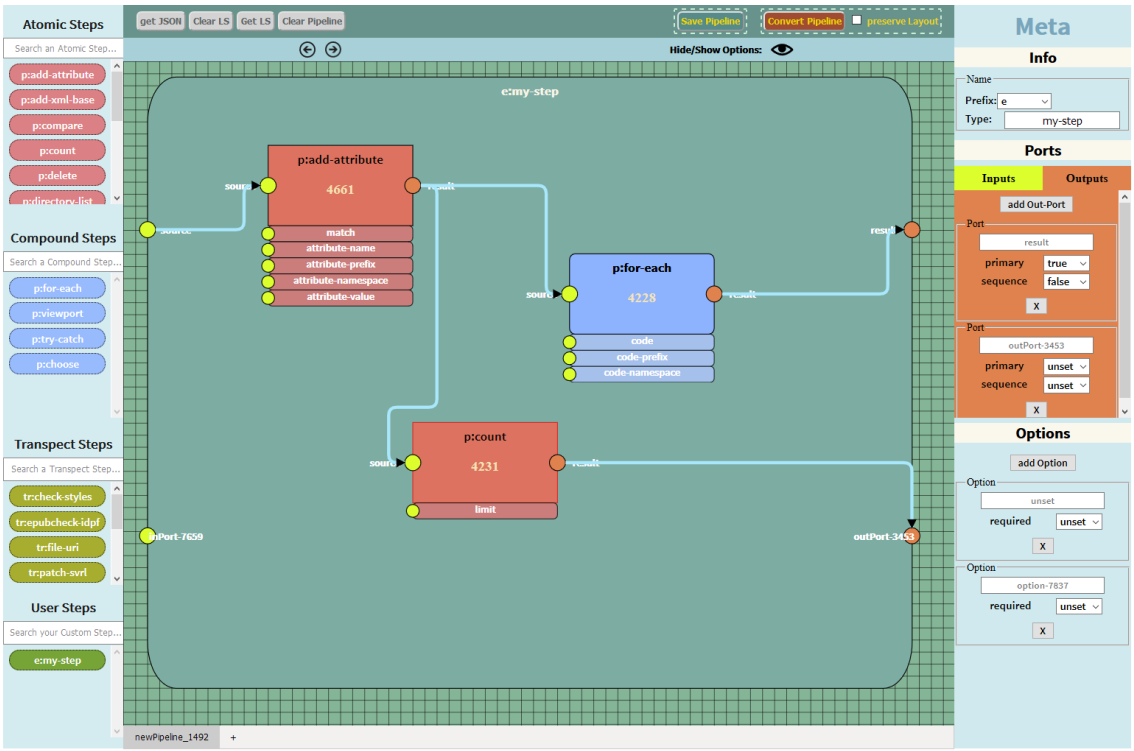


Abb. 30: Screenshot vom 3.4.2019

Die Abbildung zeigt den aktuellen (sichtbaren) Stand des XProc-Editors. Oberhalb des Hauptbereichs ist ein neuer Aktionsbereich hinzugekommen. Dieser, im Folgenden **Werkzeugleiste** genannte, Bereich umfasst vier (graue) Funktionsbuttons zum Anzeigen von JSON-Daten in der Browserkonsole und zum Zurücksetzen der Haupt-Pipeline. Außerdem gibt es einen (grünen) Button zum Speichern der Pipeline in einer eigenen Library und einen (roten) Button zum Konvertieren der Pipeline in ein XProc-Dokument. In der unteren Hälfte der Werkzeugleiste sind zudem Vor- und Zurück-Buttons zur Navigation in den vollzogenen Arbeitsschritten und ein Button zum Anzeigen und Ausblenden aller Options im Hauptbereich enthalten.

Der Paper-Switch dient in der aktuellen Version vor allem der Bearbeitung von Compound Steps bzw. Subpipelines. Wenn doppelt auf einen solchen Step geklickt wird, wird dieser in einem neuen Paper als großformatige Pipeline ‚geöffnet' und kann so im Innern bearbeitet werden. Dem Paper-Switch wird entsprechend eine neue Registrierkarte hinzugefügt. Über einen Plus-Button rechts der Registirerkarten können außerdem zusätzliche Haupt-Pipelines erstellt werden. Dieses Vorgehen löst bis auf Weiteres ein grundlegendes Platzproblem in der Bearbeitung von XProc-Pipelines.

Mit der Überarbeitung in dieser zuletzt beschriebenen Entwicklungsphase des Editors wurde auch eine Präsentation auf der Fachtagung *XML-Prague* anvisiert, die vom 7.-9.

Februar 2019 stattfand.[73] Zielstellung war es, beim jährlich stattfindenden DemoJam der Konferenz aus einer im XProc-Editor zusammengestellten Pipeline ein valides XProc-Dokument zu generieren. Die hierzu notwendige Implementierung einer Datenkonvertierung mit SaxonJS (siehe Kapitel 4.5) prägte die finalen Entwicklungsschritte dieser Phase.

73 Vgl. [XML Prague – 2019]

4.2 joint.shapes.xproc.Model

Die Grundlagen grafischer JointJS-Modelle und ihrer Initialisierung wurden in den Kapiteln 3.2 und 3.3 bereits erläutert. Aus der Erweiterung der beiden Untermodelle von joint.shapes.devs.Model (devs.Atomic und devs.Coupled) wurden XProc-spezifische und wiederverwendbare Untermodelle des joint.shapes.xproc.Model entwickelt: xproc.Atomic, xproc.Compound und xproc.Option. Aus xproc.Compound wurde außerdem ein zusätzliches Modell für die umgebenden Haupt-Pipelines (xproc.Pipeline) erzeugt. Neben Layout-spezifischen Anpassungen wurden diesen Modellen Platzhalter für XProc-spezifische Daten hinzugefügt. Diese Platzhalter wurden im Laufe der Entwicklung immer wieder erweitert und an neue Anforderungen angepasst. Das folgende Code-Beispiel zeigt diese selbst erstellte Datenstruktur in einem Ausschnitt der aktuellen Spezifikation von xproc.Compound. Bis auf den vordefinierten Wert von `stepGroup`, entspricht sie der Struktur von xproc.Atomic.

```
stepGroup: "xproc.Compound",
stepType: "unset",
stepId: "unset",
stepPrefix: "unset",
stepName: "unset",
portData: [
  {
    portId: "unset",
    portGroup: "unset",
    portPrimary: "unset",
    portSequence: "unset",
    portContentTypes: "unset",
    portSerialization: {indent: "unset"}
  }
],
stepOption: [
  {
    name: "unset",
    required: "unset"
  }
]
```

Code-Beispiel 10 : Spezifische Werte aus xpoc-Compound

Es sind hier zunächst mehrere Schlüsselwörter zur groben Kategorisierung und Beschreibung der Steps aufgeführt. Das `portData`-Array ermöglicht es außerdem, beliebig vielen Ports XProc-spezifische Eigenschaften zuzuweisen, wie z.B. die bereits häufig angesprochene Definition als Primary-Port. Zuletzt können in dem `stepOption`-Array vorhandene Options eines Steps aufgelistet werden, inklusive ihrer spezifischen Eigenschaften.

Diese selbst erstellte Datenstruktur definiert gleichzeitig die Schreibweise und Struktur der zu ladenden XProc-Libraries. Aktuell sind alle zu ladenden Step-Libraries in der Datei `steps.json` aufgeführt. Die Auflistung umfasst derzeit einige Beispiel-Steps, mit denen die

Funktion der HTML-Repräsentation der Libraries und das Laden der Steps getestet und weiterentwickelt werden. Das folgende Code-Beispiel zeigt das JSON-Objekt des Steps `p:try-catch` aus der Datei steps.json.

```
{
  "stepGroup": "xproc.Compound",
  "stepType": "pTryCatch",
  "inPorts": ["source"],
  "outPorts": ["result"],
  "attrs": {
    ".label": {
      "text": "p:try-catch"
    }
  },
  "portData": [
    {
      "portId": "source",
      "portGroup": "in",
      "portPrimary": false
    },
    {
      "portId": "result",
      "portGroup": "out",
      "portPrimary": true,
      "portSequence": true
    }
  ],
  "stepOption": [
    {
      "name": "code",
      "required": true
    },
    {
      "name": "code-prefix"
    },
    {
      "name": "code-namespace"
    }
  ]
},
```

Code-Beispiel 11 : Ausschnitt aus steps.json

Neben den Schlüsseln und Arrays der selbst erstellten Datenstruktur werden hier den JointJS-Schlüsselwörtern **`inPorts`**, **`outPorts`** und **`attrs`** Werte zugewiesen. Diese Schlüsselwörter müssen bedient werden, um den grafischen Step-Modellen die entsprechenden Ports hinzuzufügen bzw. das Modell zu beschriften. In der Entwicklung des XProc-Editors wird versucht, die dynamische Datenstruktur so kurz wie möglich zu halten und möglichst viele (statische) Eigenschaften die die Spezifikation der Modelle

auszulagern.[74]

Die selbst definierten Schlüsselwörter werden von vielen Funktionen des XProcs-Editors aktiv genutzt. Der Schlüssel `stepGroup` ermöglicht z.B. das Laden der Steps in den Hauptbereich. Wird im Step-Panel ein Step-Element per Doppelklick bzw. Drag & Drop in den Hauptbereich geladen, wird innerhalb der Funktion `stepLoad()` anhand des Wertes von `stepGroup` das zu ladende grafische Modell ausgewählt. Das folgende Code-Beispiel zeigt die Funktion `stepLoad()`:

```
function stepLoad(elem, placeX, placeY) {
  getId();
  let id = elem.id;
  let stepIdNum = newId;
  let stepId = "" + id + "_" + stepIdNum;
  let i;
  for (i = 0; i < superObj.length; i++) {
    if (superObj[i].stepType === id) {
      let group = superObj[i].stepGroup;
      if (group === "xproc.Atomic") {
        loadAtomicStep(i, stepIdNum, stepId, placeX, placeY);
      }
      if (group === "xproc.Compound") {
        loadCompoundStep(i, stepIdNum, stepId, placeX, placeY);
      }
      if (group === "xproc.Custom") {
        loadCustomStep(i, stepIdNum, stepId, placeX, placeY);
      }
    }
  }
  let cell = paperX.findViewByModel(stepId);
  metaPanel(cell);
}
```

Code-Beispiel 12 : Funktion stepLoad()

Der Funktion wird über den Parameter `elem` das HTML-Element übergeben, das angeklickt wurde. Wie hier zu sehen ist, beinhaltet `elem` eine ID. Diese wurde dem HTML-Step-Element beim Laden der Step-Libraries zugewiesen. Die HTML-Step-Libraries werden beim initialen Laden des XProc-Editors aus der Datei `steps.json` erzeugt. Über einen `XMLHttpRequest` wird zunächst die JSON-Struktur abgefragt. Innerhalb der Funktion `xhr.onload()` werden anschließend die in der JSON-Datei enthaltenen Library-Arrays (aktuell `atomicSteps` und `compoundSteps`) mit den Step-Objekten (siehe Code-Beispiel 11) ausgelesen. Für jedes Step-Objekt wird anschließend ein li-Element erzeugt. Den li-Elementen wird schließlich über den Wert des Schlüssels `stepType` eine ID zugewiesen.

74 Die vollständige Spezifikation der selbst definierten devs.Model, kann in der Datei initializeJoint.js nachvollzogen werden. Zeilen 436 – 980.

Das ist die ID, auf die der Parameter `elem` in Code-Beispiel 12 zugreift. Das folgende Code-Beispiel zeigt die Erzeugung der HTML-Step-Elemente für die `compoundSteps`-Library:[75]

```
for (let j = 0; j < obj.compoundSteps.length; j++) {
  document.querySelector("#compoundUL").innerHTML +=
    "<li id='" + obj.compoundSteps[j].stepType + "' class='step
     compoundStep' draggable='true'>"
    + obj.compoundSteps[j].attrs[".label"].text + "</li>";
  // Missing Code
}
```

Code-Beispiel 13 : Ausschnitt aus der Funktion xhr.onload()

Am intensivsten wird auf die XProc-spezifische Datenstruktur bei der Dateninteraktion des Meta-Panels zugegriffen. Die Funktion `metaPanel()` umfasst eine komplexes System mehrerer dynamischer Formulare, mit denen die Step-Informationen angezeigt und bearbeitet werden können.[76] Eine wesentliche Basis der Dateninteraktion sind die zu Beginn der Funktion deklarierten Variablen, die im folgenden Code-Beispiel zu sehen sind:

```
function metaPanel(cellView) {
  //Missing Code
  let step = cellView.model.toJSON();
  let stepType = step.stepType;
  let stepPrefix = cellView.model.attributes.stepPrefix;
  let stepName = cellView.model.attributes.stepName;
  let portData = cellView.model.attributes.portData;
  let inPorts = cellView.model.attributes.inPorts;
  let outPorts = cellView.model.attributes.outPorts;
  let stepOptions = cellView.model.attributes.stepOption;
  //Missing Code
}
```

Code-Beispiel 14 : Initiale Variablen in der Funktion metaPanel()

Beim Klick auf ein grafisches Step-Modell wird das cellView-Objekt an die `metaPanel()`-Funktion übergebenen. Aus dem Unterobjekt cellView.model (siehe Abb. 20 in Kapitel 3.3) werden die spezifischen Daten der Steps ausgelesen und über die Variablen zur Anzeige und Bearbeitung an die Formulare der Funktion übergeben.
Das letzte Code-Beispiel in diesem Kapitel zeigt einen Event-Listener, der die Dateninteraktion von `select`-Schaltflächen zur Anpassung der `primary`- und `sequence`-Eigenschaften ermöglicht:

```
select.addEventListener('change', function () {
  for (let j = 0; j < portData.length; j++) {
    if (portData[j].portId === dataId) {
```

75 Die gesamte Funktion xhr.onload() und der XMLHttpRequest können in der Datei initializeJoint.js – Zeilen 55 - 93 – nachvollzogen werden.

76 Die gesamte Funktion metaPanel() kann in der Datei stepInteraction.js – Zeilen 211 - 820 – nachvollzogen werden.

```
      if (type === "primary") {
        portData[j].portPrimary = this.value;
      } else if (type === "sequence") {
        portData[j].portSequence = this.value;
      }
    }
  }
  metaPanel(cellView);
});
```

Code-Beispiel 15 : Event-Listener eines Formular-Elements in metaPanel()

In einer `for`-Schleife werden hier die `portIds` aus dem `portData`-Array durchlaufen und mit einer dem `select`-Element übergebenen `dataId` abgeglichen. Anschließend wird je nach Schaltflächen-Typ, der entsprechende Wert im `portData`-Array überschrieben.

4.3 Zielgruppe und Endgeräte

Aus der Einleitung dieser Arbeit gehen mehrere Personengruppen hervor, für die der XProc-Editor von Interesse sein könnte. Im Folgenden werden die möglichen Nutzer*innen in zwei Zielgruppen eingeteilt und grob charakterisiert. Die Charakterisierung und die anschließende Beschreibung verwendeter Endgeräte stützt sich auf vorangegangene eigene Fachgespräche und Beobachtungen. Eine fundierte wissenschaftliche Analyse der Zielgruppe, ihrem Wissensstand und den von ihr verwendeten Endgeräten war jedoch im Rahmen dieser Arbeit nicht möglich.

Die erste Zielgruppe sind XML- und XProc-Entwickler*innen bzw. allgemein Fachkräfte aus dem Bereich IT, die XProc entweder bereits kennen oder mit (ähnlichen) Programmiersprachen vertraut sind. Den Vertreter*innen dieser Gruppe ist der Umgang sowohl mit grafischen Editoren, als auch mit komplizierteren Quellcodes und Kommandozeilen vertraut. Sie haben ein gutes Verständnis von Algorithmen und Datenstrukturen. Und der Umgang mit den Entwicklerwerkzeugen eines Browsers ist für sie entweder gewohnt oder schnell erlernbar.
Zu der zweiten Gruppe gehören Interessierte, Kunden und Neueinsteiger*innen, denen das Feld der Programmierung fremd ist und die daher mit XProc bisher keine Berührungspunkte hatten. Die Vertreter*innen dieser Gruppe bringen ein Grundverständnis im Umgang mit Computern und mobilen Endgeräten mit. Die Hintergrundprozesse einer Datenkonvertierung oder eines Editors sind für sie jedoch schwer vorstellbar. Technische Neuheiten betrachten sie mit distanziertem Interesse.

Aus der Beschreibung der Zielgruppen lässt sich das Feld der Endgeräte sowie der Browser ableiten und eingrenzen, mit denen der XProc-Editor nutzbar sein soll.
Für beide Zielgruppen wird so zunächst von einer Nutzung des XProc-Editors im

beruflichen Kontext ausgegangen. Der Editor wird vermutlich hauptsächlich auf einem Arbeitscomputer verwendet.
Bei den Nutzer*innen der ersten Zielgruppe ist dies entweder ein leistungsstarker Laptop oder Standrechner. Für die alltägliche Bedienung wird am liebsten die Tastatur verwendet, wenn nötig zusätzlich Maus bzw. Touchpad. Die Darstellung erfolgt auf dem Laptopbildschirm (Ø 14 bis 15 Zoll) oder einem externen Monitor (Ø 24 Zoll) mit zeitgemäßer Auflösung (Ø 1920 x 1080 Pixel). Die verwendeten Betriebssysteme sind Windows oder Linux. Am häufigsten wird der Browser Firefox verwendet, seltener die Browser Chrome oder Opera. Die Webbrowser werden regelmäßig mit Updates versorgt, wie auch die restliche Software auf den Computern.
Die Nutzer*innen der zweiten Zielgruppe verwenden im beruflichen Kontext ebenfalls hauptsächlich Laptops- oder Standrechner. Die Leistung reicht für Office-Anwendungen völlig aus und geht aufgrund des hohen Leistungsstandards aktueller Geräte häufig darüber hinaus. Für die alltägliche Bedienung wird am liebsten die Maus verwendet. Touchpad und Tastatur sind jedoch ebenso gewohnte Bedienwerkzeuge. Die Darstellung erfolgt am häufigsten auf einem externen Monitor und ebenfalls auf dem Laptopbildschirm. In dieser Zielgruppe gibt es einige Nutzer*innen, die iMac- und MacBook-Geräte nutzen. Die durchschnittliche Zollgröße der Darstellungsgeräte liegt insgesamt wie bei der ersten Zielgruppe bei etwa 24. Die Auflösung liegt aufgrund der hohen Standardauflösung der Apple-Geräte bei durchschnittlich 2560 x 1440 Pixel. Die verwendeten Betriebssysteme sind Windows und macOS. Am häufigsten werden die Webbrowser Chrome, Firefox und Safari verwendet. Die Browser Opera, Internet Explorer (IE) und Edge werden ebenfalls, jedoch selten genutzt. Auch bei dieser Zielgruppe werden die Webbrowser und die restliche Computersoftware regelmäßig mit Updates versorgt. Bei einer Begegnung von erster und zweiter Zielgruppe, z.B. einer Nutzung des XProc-Editors in einem Kundengespräch, wäre die Darstellung entweder über einen Beamer oder (z.B. auf einer Geschäftsreise) auch mit einem Tablet wünschenswert.

Trotz der großen Zahl verwendeter Webbrowser wurde der Editor bis zum aktuellen Entwicklungsstand für aktuelle Firefox-Versionen (verwendet wurde Firefox 65) optimiert, und mit den Entwicklerwerkzeugen dieses Browsers getestet und bearbeitet. In regelmäßigen Abständen wurde der Editor außerdem in Google Chrome (Version 73) und Opera (Version 58) getestet. Die Browser Safari, IE und Edge wurden bisher nicht berücksichtigt.
Für die Javascript-Entwicklung wurde die mit ECMAScript 6 neu eingeführte Technik der **Block-Scope** genutzt, um mehr Klarheit in die Reichweite von Variablen in Code-Blöcken zu bringen. (Vgl. Ackermann 2015: 179) Block-Scope wird neben aktuellen Versionen von

Firefox, Chrome und Opera auch in aktuellen Versionen von Edge und Safari und teilweise in IE 11 unterstützt.[77] Im Code wurde so anstelle des klassischen `var` das Schlüsselwort `let` genutzt. Da bisher keine statischen Variablen benötigt wurden, kommt `const` im aktuellen Quellcode nicht vor.
Die CSS-Regeln wurden bisher ebenfalls nur für die drei bevorzugten Browser angepasst. Im folgenden Abschnitt wird diesbezüglich näher auf das verwendete CSS-Grid-Layout eingegangen.

4.4 Layout und Gestaltung

Auch wenn die Gruppe anvisierter Endgeräte und Browser für die Entwicklung des XProc-Editors eingegrenzt wurde, ist die Zahl möglicher Displaygrößen und Auflösungen immer noch hoch. Um diesem Problem zu begegnen, auch mit Blick auf eine zukünftige Erweiterung der Gruppe in Frage kommender Endgeräte, war die Beschäftigung mit der **Responsivität** von Web-Anwendungen und der Entwurf eines responsiven Layouts für den XProc-Editor ein wichtiger Teil der Entwicklung.
Das Grundgerüst des Editors basiert aktuell auf einem Raster, das durch ein CSS-Grid erzeugt wird. Zur leichteren Orientierung in den folgenden Ausführungen, zeigt das folgende Code-Beispiel die HTML-Grundstruktur des Editors:

```
<div id="container">
  <div id="stepPanel">
    <!-- Missing Code -->
  </div>
  <div id="topPanel">
    <!-- Missing Code -->
  </div>
  </div>
  <div id="toolBar">
    <!-- Missing Code -->
  </div>
  <div id="metaPanel">
    <!-- Missing Code -->
  </div>
  <div id="papers">
    <div id="paper1" class="paperContent" style="background-color:
     #5fb598;"></div>
  </div>
  <div id="paperSwitch">
    <!-- Missing Code -->
  </div>
</div>
```

Code-Beispiel 16 : Ausschnitt aus der HTML-Struktur des XProc-Editors

77 Vgl. [Can I use – let]

Das umgebende div-Element (`container`) wird durch das CSS-Grid aktuell in drei Spalten und vier Zeilen unterteilt, denen die Bereiche des XProc-Editors zugeordnet werden. Die Werkzeugleiste ist hier in die Bereiche `topPanel` und `toolBar` unterteilt. Das folgende Code-Beispiel zeigt die Aufteilung durch das CSS-Grid und die allgemeinen Größenangaben des div-Elements:

```
#container {
  display: grid;
  width: 95vw;
  height: 95vh;
  grid-template-columns: [stepPanel] 150px [paper] 70% [metaPanel] 220px;
  grid-template-rows: [topPanel] 2.6em [toolBar] 1.8em [paper] 86%
                      [paperSwitch] 1.85em;
  margin-left: 1vw;
}
```

Code-Beispiel 17 : CSS-Regeln des div-Containers

Den Spalten für Step-Panel und Meta-Panel wurden feste Breiten zugewiesen, um die Gestaltung dieser komplexeren Bereiche zu vereinfachen. Ebenso haben die Zeilen für Werkzeugleiste und Paper-Switch feste Höhen.
In der Aufteilung fällt auf, dass die Maße der Grid-Zelle für das Paper in Prozentwerten angegeben sind. Dies hängt mit dem Versuch zusammen den Editor zumindest in Teilen responsiv zu gestalten. Hierzu sollen sich die Maße des Paper an der jeweils aktuellen Größe des Paper-div orientieren. Im Datenmodell des Papers sind hierzu für `width` und `height` die standardmäßigen Pixelwerte mit JavaScript-Properties ersetzt. Über die Variable `canvas` wird auf das div-Element des Papers zugegriffen und dessen Maße werden über `canvas.offsetWidth` bzw. `canvas.offsetHeight` ausgelesen. Das folgende Code-Beispiel zeigt dieses Vorgehen:

```
let canvas = document.querySelector('#papers');
let graph = new joint.dia.Graph,
  paper = new joint.dia.Paper({
    el: $('#paper1'),
    model: graph,
    // width: 938,
    // height: 854,
    width: canvas.offsetWidth,
    height: canvas.offsetHeight,
// Missing Code
});
```

Code-Beispiel 18 : Ausschnitt aus der Initialisierung des Papers

Damit sich die Größe des Papers dynamisch an die Größenveränderung des Browserfensters anpasst, wurde die Funktion `window.resize()` erarbeitet. Bei jeder Größenveränderung des Browserfensters werden die aktuellen Maße des Paper-div an eine

Variable übergeben und die Größe des Papers wird dynamisch hieran angepasst. Das folgende Code-Beispiel zeigt die Funktion:

```
window.addEventListener("resize", function () {
  let canvasWidth = canvas.offsetWidth;
  let canvasHeight = canvas.offsetHeight;
  paperX.setDimensions(canvasWidth, canvasHeight);
  let xplWidth = canvasWidth * 0.8 + (canvas.offsetWidth * 10 / 100);
  let xplHeight = canvasWidth * 0.6;
  let currentPipeline = graphX.getCell(globalPipeline);
  currentPipeline.resize(xplWidth, xplHeight);
});
```

Code-Beispiel 19 : Funktion window.resize()

Neben dem beschriebenen Grid-Layout des Container-divs, gibt es mehrere untergeordnete Raster. Das div-Element des Step-Panels ist mit einem weiteren CSS-Grid in aktuell vier Zeilen und eine Spalte unterteilt. Den Zeilen sind die verschiedenen Step-Libraries zugeordnet. Die beiden Bereiche der Werkzeugleiste basieren auf CSS-Flexbox. Auch mit Blick auf eine Erweiterung der Buttons und Schaltflächen in diesem Bereich erscheinen die Möglichkeiten der Flexbox, Elemente gleichmäßig zu verteilen, für diesen Bereich geeignet.

Neben der Responsivität sind intuitive und platzsparende UI-Elemente ein wesentlicher Aspekt in der Gestaltung des XProc-Editors. In Kapitel 4.1 wurde bereits auf die Neustrukturierung des Step-Panels eingegangen. Die Filtersuche innerhalb der Step-Libraries und die scrollbaren Step-Listen ermöglichen es, unbegrenzt viele Steps in einer Library unterzubringen. Die farbliche Unterscheidung der verschiedenen Step-Kategorien in den Step-Libraries und der entsprechenden Step-Modelle im Hauptbereich soll die Orientierung in komplexen XProc-Pipelines erleichtern.
Die Formulare des Meta-Panels sind aktuell in erster Linie funktional. Um den Bereich „Ports“ übersichtlicher und platzsparender zu gestalten, sind die Auflistungen von Input- und Outpout-Ports in wechselbare Bereiche unterteilt, dessen farbliche Gestaltung an die Farben der Ports im Hauptbereich angepasst ist.

4.5 XProc-Konvertierung

Die geplante Verarbeitung des XProc-Editors verläuft in zwei Richtungen. XProc-Pipelines sollen grafisch zusammengestellt und in ein XProc-Dokument übertragen werden. Und aus einem XProc-Dokument soll eine grafische, editierbare Darstellung generiert werden. Für diesen Prozess gibt es aktuell vier Schlüsselinstanzen: Das XProc-Dokument, der SaxonJS-Prozessor, JointJS bzw. die JavaScript-Funktionen und der Webbrowser. Das folgende

Schaubild fasst die geplanten Konvertierungsabläufe des XProc-Editors schematisch zusammen.

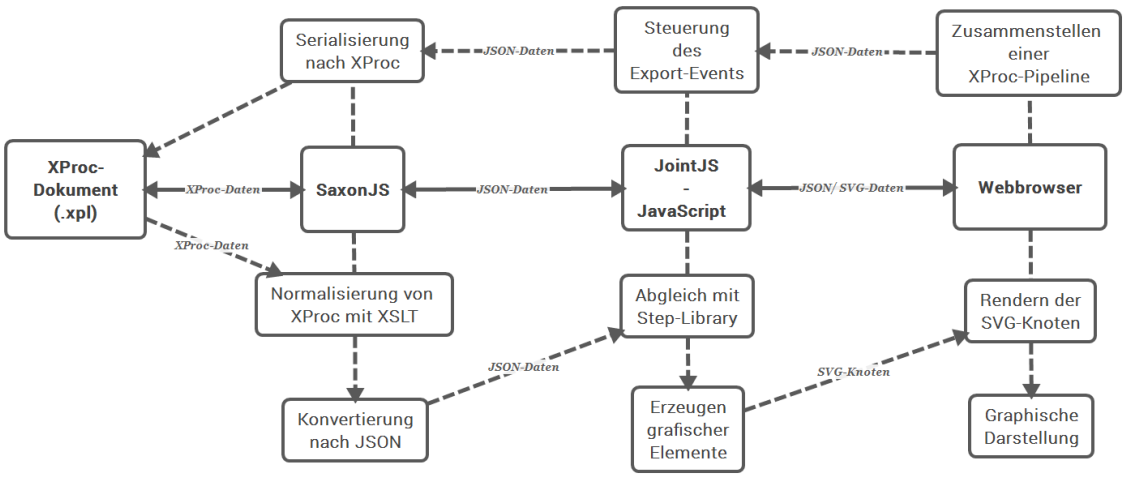


Abb. 31: Schema der Verarbeitungsprozesse des XProc-Editors

Auf einige Aspekte der Verarbeitung wurde in den vorangegangenen Ausführungen bereits eingegangen. Die zweite Schlüsselinstanz **SaxonJS** wurde bisher jedoch noch nicht genauer betrachtet. Diese auf JavaScript basierende Laufzeitumgebung dient der Ausführung von XSLT 3.0 im Webbrowser und der Transformation insb. von XML und JSON-Daten. (Vgl. Meinike 2017: 177) SaxonJS wurde von der Firma *Saxonica* entwickelt und im Februar 2017 in erster Version veröffentlicht.[78] Die Transformationen mit SaxonJS werden durch die Stylesheet Export Files (SEF) ermöglicht. Dies sind angepasste XML-Dokumente, die mit dem Prozessor Saxon-EE (ab Version 9.7) aus einem XSLT-Stylesheet erstellt werden.

Im XProc-Editor werden die SEF-Dateien zur Ausführung von SaxonJS innerhalb der Funktion `window.onload()` eingebunden, wie das folgende Code-Beispiel zeigt.

```
SaxonJS.transform({
  sourceLocation: "xsl/xproceditor.sef",
  stylesheetLocation: "xsl/xproceditor.sef"
});
```

Code-Beispiel 20 : SaxonJS-Aufruf in Funktion window.onload()

Das der SEF-Datei zugrunde liegende Stylesheet ist somit das Kernelement der XProc-Konvertierung. Dessen Entwicklung steht noch am Anfang und fokussiert sich bisher auf den Export bzw. die Serialisierung eines XProc-Dokuments aus einer grafischen Pipeline. Hierzu wird innerhalb des Stylesheets ein Event-Listener definiert, der an den Pipeline-

78 Vgl. [Saxonica – SaxonJS]

Export-Button (siehe Abb. 30) gebunden wird. Das folgende Code-Beispiel zeigt diesen Aufruf:

```
<xsl:template mode="ixsl:click" match="id('btn_pipe')">
  <xsl:result-document href="#xproc_xml" method="ixsl:replace-content">
    <xsl:call-template name="create-xpl"/>
  </xsl:result-document>
</xsl:template>
```

Code-Beispiel 21 : Event-Aufruf zum Export der XProc-Pipeline

Hier wird bereits festgelegt, wohin das XProc-Dokument exportiert werden soll. Aktuell ist dies ein verstecktes div-Element mit der id `xproc_xml`. Innerhalb dieser Referenz zum Ergebnisdokument wird anschließend ein Template (`create-xpl`) für das zu generierende XProc-Dokument aufgerufen. Innerhalb dieses Templates wird auf die JSON-Daten der grafischen Pipeline zugegriffen. Im folgenden Code-Beispiel ist zu sehen, wie hierzu die Funktion `JSON.stringify(graphX.toJSON())` ausgeführt wird:

```
<xsl:template name="create-xpl">
  <xsl:variable name="graph" as="document-node(element(fn:map))"
    select="json-to-xml(
              ixsl:eval('JSON.stringify(graphX.toJSON())')
            )"/>
  <xsl:variable name="simplify-json-representation">
    <xsl:apply-templates select="$graph" mode="simplify-json-
        representation">
      <xsl:with-param name="retain-layout" select="ixsl:get(id('layout-
        checkbox',ixsl:page()), 'checked')" as="xs:boolean" tunnel="yes"/>
    </xsl:apply-templates>
  </xsl:variable>
  <!-- Missing Code -->
</xsl:template>
```

Code-Beispiel 22 : XSL-Template zur XProc-Erzeugung

Das Element `graphX` referenziert den aktuell geöffneten JointJS-Graph. Hier wird somit ein JSON-String aus den im Graph enthalten Cells erzeugt. Die JSON-Daten werden anschließend über den `mode „simplify-json-representation"` vereinfacht und den nächsten Konvertierungsschritten zur Verfügung gestellt. Die XSLT-Entwicklung des XProc-Editors wird vom Verfassers dieser Abschlussarbeit nicht selbst durchgeführt, weswegen auf weitere Details an dieser Stelle nicht eingegangen wird.

4.6 Dokumentation

Die folgende Dokumentation soll eine Orientierung bei der Bedienung des XProc-Editors bieten und die Interaktionsmöglichkeiten erklären.

Inhalte der Dokumentation

1. Erste Schritte
2. Laden eines XProc-Steps
3. Bewegen und Verknüpfen von XProc-Steps
4. Interaktionsmöglichkeiten mit den Steps im Graph
5. Das Meta-Panel
6. Funktionen der Werkzeugleiste

1. Erste Schritte

(1) Öffnen Sie die Datei index.html des Programmordners in einem Web-Browser (Firefox oder Chrome).
→ Der XProc-Editor wird geladen.

(2) Der XProc-Editor ist in fünf Aktionsbereiche unterteilt, welche in dieser Anleitung folgendermaßen bezeichnet werden:

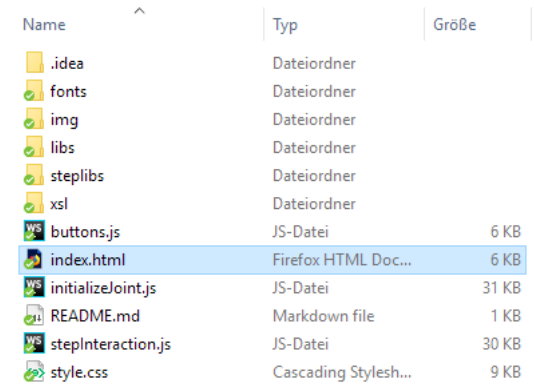


Abb. 32: Screenshot Dateiordner

(A) Hauptbereich

(B) Step-Panel

(C) Werkzeugleiste

(D) Meta-Panel

(E) Paper-Switch

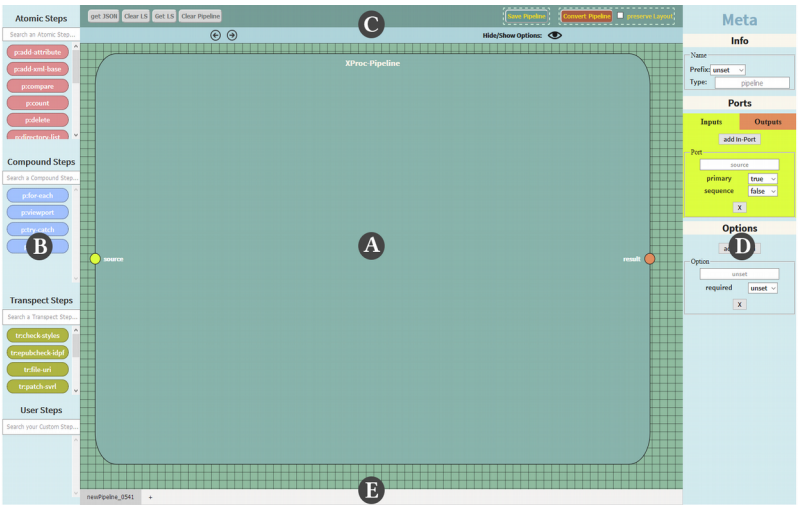


Abb. 33: Aufteilung des XProc-Editors (Screenshot)

2. Laden eines XProc-Steps

Beim Laden des Editors erscheint im Hauptbereich eine frei editierbare XProc-Pipeline. Um dieser Pipeline einen Step bzw. eine Subpipeline hinzuzufügen, gehen Sie folgendermaßen vor:

(1) Suchen Sie einen Step aus der gewünschten Bibliothek (Step-Panel) aus. Nutzen Sie bei Bedarf das jeweilige Suchfeld, um den Step zu finden.

> ✗ Hinweis: Die Steps der „Transpect"-Bibliothek sind derzeit nicht verfügbar.

(2) Um den gewählten Step in Ihre Pipeline zu laden, haben Sie zwei Möglichkeiten:

(a) Klicken Sie doppelt auf den gewünschten Step, um ihn in die Pipeline zu laden. Auf diesem Weg wird der Step immer an der gleichen Stelle erzeugt und Sie können ihn anschließend an die gewünschte Stelle verschieben.

(b) Ziehen Sie den Step per Drag & Drop an die gewünschte Stelle in ihrer Pipeline.

→ Beim Laden des Steps erscheinen automatisch dessen Meta-Informationen im Meta-Panel.

3. Bewegen und Verknüpfen von XProc-Steps

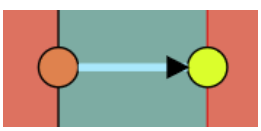
Sobald Sie einen XProc-Step erzeugt haben, können Sie diesen per Drag & Drop frei im Hauptbereich bewegen.

Um zwei Steps miteinander zu **verknüpfen**, gehen Sie folgendermaßen vor:

(1) Klicken Sie in den Output-Port eines Steps und halten Sie die Maustaste gedrückt

(2) Bewegen Sie den Mauszeiger über den Input-Port eines anderen Steps und lassen Sie die Maustaste anschließend los.
→ Die Verknüpfung wird erstellt und die beiden Steps werden verbunden

> ✗ Achten Sie darauf, Input- und Output-Ports immer in Richtung des Dokumentenflusses zu verbinden, also von Input nach Output. Die spätere Konvertierung schlägt ansonsten fehl.



4. Interaktionsmöglichkeiten mit den Steps im Graph

Wenn Sie mit dem Mauszeiger über ein Step-Model fahren, erscheinen zwei Icons, die folgende Funktionen haben:

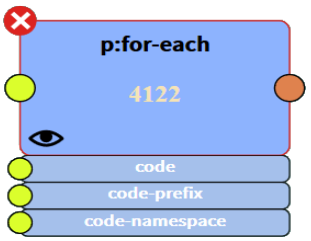
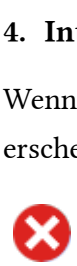


Abb. 34: Screenshot Step-Model


Durch Klick auf das Lösch-Icon wird der Step aus der Pipeline bzw. dem Hauptbereich entfernt.

> ✗ Sie können den Step alternativ mit der Entf-Taste Ihrer Tastatur entfernen. Hierzu muss der Step zunächst per Klick ausgewählt werden.


Durch Klick auf das Augen-Icon werden die Optionen eines Steps ein- bzw. ausgeblendet.

5. Das Meta-Panel

Das Meta-Panel erfüllt mehre Funktionen gleichzeitig. Es zeigt Informationen bzw. Metadaten zu ausgewählten Steps an. Die Informationen sind in drei Bereiche unterteilt: Info, Ports und Options. Je nach ausgewähltem Step-Typ, lassen sich bestimmte Metadaten verändern. Wenn ein Step aus den Bibliotheken „Atomic“ oder „Compound“, ausgewählt wird, sind (bisher) keine der angezeigten Metadaten veränderbar.

> ✗ In der aktuellen Version können Sie den Wert der Optionen eines Steps nur verändern, indem Sie auf eine Option eines Step-Model (Hauptbereich) klicken und den Wert anschließend in das Eingabefeld im Meta-Panel eintragen.

Wenn eine Haupt-Pipeline ausgewählt ist, können Sie im angezeigten Meta-Panel (siehe Abb. 35) folgende Werte verändern:

Info

(1) Wählen Sie aus der Auswahlliste (I1) ein **Präfix** für Ihre Pipeline.

✓ Die Beschriftung der Pipeline passt sich an und die Information wird ins Datenmodell des Steps geschrieben.

(2) Definieren Sie den **Typ** Ihrer Pipeline. Geben Sie hierzu einen Namen in das Eingabefeld (I2) ein. Der gewählte Name sollte die Funktion der Pipeline beschreiben. Großbuchstaben sind (bisher) erlaubt, sollten jedoch vermieden werden. Um der Standardschreibweise zu entsprechen, sollten bei Namen mit zwei oder mehr Wörtern durch Bindestriche verbunden bzw. getrennt werden, z. B.: „my-new-step“.

(3) Klicken Sie zum Bestätigen neben das Eingabefeld oder drücken Sie die Enter-Taste.

✓ Die Beschriftung der Pipeline passt sich an und die Informationen werden gespeichert.

Ports

(1) Wählen Sie die Schaltfläche „Inputs“ oder „Outputs“ (P1) um die entsprechenden Ports zu bearbeiten.

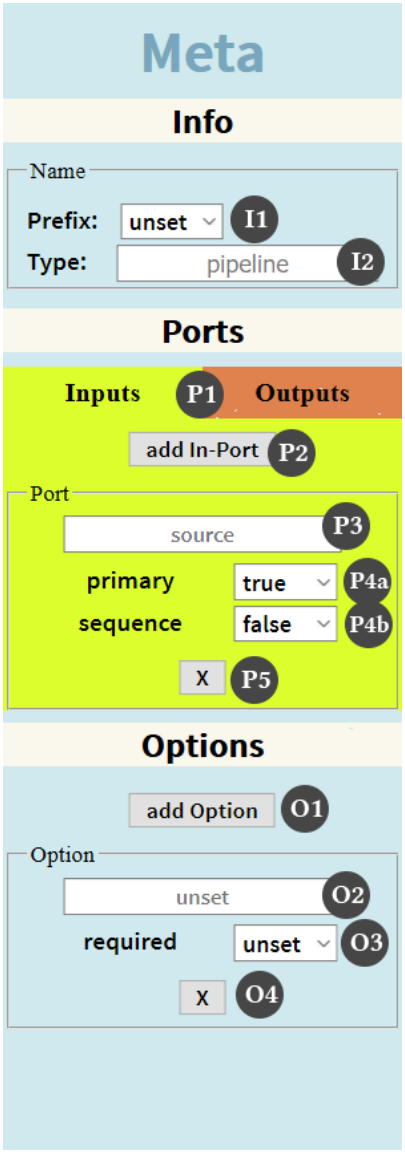


Abb. 35: Screenshot Meta-Panel

(2) Um Ihrer Pipeline einen neuen Port hinzuzufügen, klicken Sie auf den Button „Add In-Port“ bzw. „Add Out-Port“ (P2).

✓ In der grafischen Pipeline erscheint ein neuer Port.

✓ Dem Bereich „Ports“ des Meta-Panels wird ein neues Port-Feld hinzugefügt.

✓ Dem Datenmodell des Steps wird ein neuer Port-Datensatz hinzugefügt.

(3) Um den Namen eines Ports zu verändern, geben Sie einen neuen Namen in das Eingabefeld (P3) ein. Der Name darf für jeden Step nur einmalig vorkommen. Klicken Sie neben das Feld oder drücken Sie die Enter-Taste zum Bestätigen.

✓ Das Label des Ports passt sich an und die Informationen werden gespeichert.

(4) Um den Primary- oder Sequence-Wert eines Ports zu verändern, wählen Sie den gewünschten Wert aus der Auswahlliste (P4a und P4b).

✓ Die Informationen werden gespeichert.

(5) Um einen Port zu löschen, klicken Sie auf den „X"-Button (P5) im entsprechenden Port-Feld.

✓ Der Port verschwindet aus der Pipeline.

✓ Das Port-Feld wird gelöscht.

✓ Der Port-Datensatz wird aus dem Datenmodell des Steps entfernt.

Options

(1) Um Ihrer Pipeline eine neue Option hinzuzufügen, klicken Sie auf den Button „Add Option" (O1).

✓ Dem Options-Bereich des Meta-Panel wird ein neues Option-Feld (O1x) hinzugefügt und die Information wird ins Datenmodell des Steps geschrieben.

(2) Um einer Option einen Namen zu geben, tragen Sie den Namen in das entsprechende Eingabefeld (O2) ein. Klicken Sie neben das Feld oder drücken Sie die Enter-Taste, um zu bestätigen.

✓ Die Information wird ins Datenmodell des Steps geschrieben

(3) Um den Wert „required" zu verändern, wählen Sie den gewünschten Wert aus der Auswahlliste (O3).

✓ Die Information wird ins Datenmodell des Steps geschrieben.

(4) Um eine Option zu löschen, klicken Sie auf den entsprechenden „X"-Button (O4).

✓ Die Option wird aus dem Datenmodell des Steps entfernt.

✗ Das aktuelle JSON-Datenmodell eines Elements im Hauptbereich (Step, eigene Pipeline, Option) können Sie jederzeit einsehen, indem Sie die Entwicklerwerkzeuge Ihres Browsers öffnen und auf das entsprechende Element klicken. Das Datenmodell wird in der Konsole ausgegeben.

6. Funktionen der Werkzeugleiste

Die Werkzeugleiste bietet mehrere Buttons, deren Funktionen hier genauer erläutert werden:

Abb. 36: Screenshot Werkzeugleiste

Get JSON: Der JSON-String des aktuellen Graphen (inklusive aller Pipelines und Links) wird ausgelesen und in die Entwicklerkonsole geschrieben.

Clear LS: Der Local Storage des Nutzers wird geleert.

Get LS: Der Local Storage des Nutzers wird ausgelesen und in die Entwicklerkonsole geschrieben.

Clear Pipeline: Die aktuelle Pipeline wird zurückgesetzt.

> ✗ Hierdurch werden alle Inhalte der Pipeline inkl. Ports und Options gelöscht!

Save Pipeline: Die aktuelle Pipeline wird gespeichert und in der Step-Bibliothek „User Steps“ abgelegt.

> ✗ Dies funktioniert nur, wenn für die Pipeline ein Präfix und ein Typ definiert wurde.

Convert Pipeline: Aus der aktuellen Pipeline wird erzeugt.

> ✗ Bisher wird der XProc-Code nur in ein verstecktes Div-Element (id: `xproc_xml`) geladen. Das Div-Element finden Sie über den Inspektor (Firefox) oder die Elements (Chrome) der Entwicklerwerkzeuge.

+ preserve Layout: Wird an dieser Stelle ein Häkchen gesetzt, werden die grafischen Informationen der Pipeline (z. B. Position, Farben) mit an den XProc-Konverter übergeben.

5 Ergebnisse

Aufbauend auf die vorangegangenen Ausführungen werden im Folgenden die wichtigsten Ergebnisse des bisherigen Entwicklungsprozesses zusammengetragen.

Erstellung und Konvertierung von Pipelines

Es konnte eine differenzierte Datenstruktur entwickelt werden, die bereits einige zentrale Aspekte von XProc wiedergibt. Die Repräsentation von Primary-Ports ist die wesentliche Basis, um das Konzept der Default Readable Ports umzusetzen und eine grafische Reihenfolge der Step-Elemente zu ermöglichen. Den grafischen Elementen werden so die wesentlichen Daten übergeben, auf deren Basis die Programmierung von Verknüpfungsregeln möglich ist. Erste Versuche, solche Regeln zu definieren, waren bereits erfolgreich.
Die Dateninteraktion durch das Meta-Panel funktioniert bei einigen Metadaten bereits ohne Einschränkungen. Die aktuelle Struktur der `metaPanel()`-Funktion bietet eine solide Basis zur Ausweitung der Funktionalität auf weitere Daten. Es wurden erste Regeln programmiert, die Nutzer*innen z. B. verpflichten, eine Pipeline vor dessen Export zu benennen. Das dynamische Hinzufügen und Löschen von Ports und Options funktioniert sowohl im Daten-Modell, als auch (im Fall der Ports) in der grafischen Generierung. Auch in weiteren Aspekten sind die Veränderungen der Datenstruktur in der grafischen Darstellung nachvollziehbar.
Durch die Funktionalität des Paper-Switchs wurde das Konzept der Kapselung von XProc-Steps grafisch umgesetzt. Um die Intuitivität in der Bearbeitung von Pipelines zu erhöhen und das Verständnis von Datenfluss und Step-Verknüpfungen zu erleichtern, wäre die Darstellung verschachtelter Steps innerhalb der Haupt-Pipeline hilfreich. Auf das diesbezügliche Platzproblem wird weiter unten nochmals eingegangen.
Es ist mittlerweile möglich, eine zusammengestellte Pipeline als wiederverwendbaren Step (über den Button „Save Pipeline“) in einer eigenen Step-Library abzulegen. Nutzer*innen können eigene Pipelines nunmehr in neuen Kontexten nutzen.
Kurz vor der erstmaligen öffentlichen Vorstellung des XProc-Editors konnte aus einer grafischen Pipeline ein erstes XProc-Dokument serialisiert werden. Dies wird zwar bisher nur in ein verstecktes div-Element geschrieben, erfüllt jedoch die wesentlichen Anforderungen an eine Pipeline.

Die grafische und datenbasierte Vorbereitung der XProc-Serialisierung ist trotz der erreichten Möglichkeiten weiterhin die zentrale Baustelle der Entwicklung. Um den Aufwand der Konvertierungen zu reduzieren, sollten möglichst viele XProc-Fehlermeldungen bereits durch Regeln in der grafischen Zusammenstellung der Pipeline abgefangen werden. Hierzu bedarf es längerer Entwicklungszeit.

Die Generierung einer grafischen Pipeline aus einem XProc-Dokument wurde bisher noch nicht ermöglicht und entsprechende Schnittstellen wurden noch nicht entwickelt. Die aktuelle Datenstruktur erscheint geeignet, um dieses Vorhaben umzusetzen. Es wird dennoch eine Herausforderung sein, eine JSON-Struktur zu generieren, mit der sich eine zusammenhängende Pipeline erzeugen lässt, die sich passend und gut gelayoutet in den Graph einfügt.

Layout und Responsivität

Der XProc-Editor reagiert derzeit sowohl auf horizontale als auch vertikale Größenveränderungen. Wird das Browserfenster verkleinert, reduzieren sich gleichzeitig die Größenverhältnisse von Paper und Haupt-Pipeline.
Neben diesem äußeren responsiven Rahmen, bleibt die ‚innere Responsivität' des Editors eine Herausforderung. Da XProc-Pipelines sehr umfangreich werden können, gibt es im Hauptbereich des Editors ein generelles Platzproblem. Es gab bereits erste Versuche, eine Funktion für das Laden neuer Steps zu programmieren. Der noch verfügbare Platz innerhalb einer Haupt-Pipeline sollte ermittelt und die Größe aller geladenen JointJS-Elemente dynamisch hieran anpasst werden. Die Funktion erschien jedoch zu aufwändig, um sie weiterzuentwickeln. Außerdem gibt es aktuell eine über das Maus-Scrollrad funktionierende Zoom-Funktion. Diese ermöglicht zwar je nach Bedarf eine Verkleinerung der grafischen Elemente, jedoch wird die Haupt-Pipeline ebenso verkleinert oder vergrößert. Das Platzproblem hierdurch nicht gelöst.
JointJS bringt einige Funktionselemente mit, die eine dynamische Größenanpassung der grafischen Elemente ermöglichen bzw. unterstützen könnten, wie z. B. die Funktionen paper.fitToContent() und paper.scaleContentToFit(). Ebenso enthält die JointJS-API einen Bereich „Layout".[79] Unter Einbeziehung der zusätzlichen Frameworks Dagre und Graphlib sollen sich automatische Layouts für gerichtete Graphen erstellen lassen. Eine genauere Recherche der genannten Funktionen sowie erste Entwicklungsversuche sind zu empfehlen.

Die Bereiche Step- und Meta-Panel enthalten bereits einige Elemente, welche eine Sortierung und platzsparende Aufbereitung von Informationen und Elementen ermöglichen. Das Layout des Step-Panels lässt es jedoch derzeit nicht zu, weitere Libraries zu integrieren. Ebenso stößt das Meta-Panels bei der Darstellung komplexer Steps und Pipelines an Grenzen. Diesbezüglich sollte erörtert werden, inwieweit die Dateninteraktion

79 Vgl. [JointJS – Docs #Layout]

in eine modulare Struktur überführt werden könnte und bestimmte Metadaten von Steps und Ports grafisch im Hauptbereich anpassbar wären.

Im Kontext der zunehmenden Nutzung mobiler Endgeräte wie z. B. Tablets ist es möglicherweise lohnenswert für den XProc-Editor, zukünftig über Touch-optimierte Events und ein entsprechendes Layout nachzudenken. Es wäre zu recherchieren, inwieweit JointJS eine mobile Nutzung unterstützt. Außerdem sollte eine genauere Zielgruppenanalyse zunächst untersuchen, ob eine solche Weiterentwicklung von Interesse wäre.

Verfügbarkeit und Verwendbarkeit

In aktuellen Versionen von Firefox sind alle implementierten Funktionen nutzbar und das Layout entspricht den Vorstellungen. In Chrome und Opera weicht die Gestaltung leicht ab. Außerdem ist hier darauf zu achten, den XProc-Editor von einem Server und nicht lokal zu starten. Bei der lokalen Nutzung werden aktuell z. B. die Step-Libraries nicht geladen. Wie bereits beschrieben, wurden IE, Edge und Safari in der Entwicklung bisher nicht berücksichtigt.

Ausblick

Der XProc-Editor wurde am 8. Februar 2019 beim DemoJam der *XML-Prague* zum ersten Mal öffentlich präsentiert.[80] In einem kurzen Vortrag wurden dem anwesenden Fachpublikum die wesentlichen Funktionen des Editors vorgestellt und aus einer grafisch zusammengestellten Pipeline wurde ein wohlgeformtes XProc-Dokument erzeugt. Die Reaktionen auf den XProc-Editor waren äußerst positiv und von einigen Teilnehmenden der Tagung wurde Interesse an weiteren Informationen, auch bezüglich einer gemeinsamen Weiterentwicklung des Projekts, geäußert.
Auch beim Praxispartner dieser Arbeit besteht großes Interesse, die Entwicklung weiterzuführen. Seit November 2019 gibt es für den XProc-Editor ein GitHub-Repository, über das seitdem die gemeinsame Bearbeitung des Quellcodes verwaltet wird.[81] Sollte der XProc-Editor durch den Praxispartner weiterentwickelt werden, stünde in der nächsten Entwicklungsphase eine intensivierte Arbeit an der XProc-Serialisierung an. Dies umfasst den Bereich der XSLT-Konvertierung mit SaxonJS sowie die Weiterentwicklung der Datenausgabe durch den XProc-Editor.

80 Vgl. [XML Prague – 2019]
81 Vgl. [GitHub – xprocedit]

6 Fazit

Nach dem bisherigen Entwicklungsprozess und den Ergebnissen dieser Abschlussarbeit kann JointJS als ein geeignetes Framework bezeichnet werden, um einen lauffähigen grafischen XProc-Editor zu entwickeln. Die grafische Struktur von XProc ist aufgrund der Kapselung von Pipelines, implizit verbundenen DRPs sowie expliziten Verknüpfungen äußerst komplex. Diese Struktur in einem zweidimensionalen Editor abzubilden, ist eine Herausforderung. Die Grundstruktur von JointJS scheint wie für die Visualisierung von XProc gemacht. In der umfangreichen API finden sich immer wieder neue Elemente und Funktionen, die zukünftige Probleme bei der Pipeline-Erstellung und -Konvertierung lösen könnten. Die einfachen Möglichkeiten, vordefinierte Datenmodelle an eigene Erfordernisse anzupassen und eigene Prototypen zu entwickeln, konnten überzeugen. JointJS bietet eine solide Basis um XProc-Pipelines systematisch auf eine Serialisierung vorzubereiten. Die gute Dokumentation hat den Einstieg in die Entwicklung mit JointJS leicht gemacht. Das Framework wird stetig weiterentwickelt und gewartet und es gibt eine wachsende Community, die JointJS nutzt und ihre Erfahrungen und Lösungswege in Internetforen zur Verfügung stellt.
Bei der Gestaltung eines intuitiven GUI ist JointJS dagegen wenig hilfreich. Die wesentlichen Arbeitsbereiche, Schaltflächen und die Schnittstellen zum Datenaustausch müssen eigenständig entwickelt und programmiert werden. Die flexible Datenstruktur des Frameworks macht den Entwicklungsaufwand jedoch überschaubar.
Im Bereich von Responsivität und automatischer Layouts bietet JointJS mehrere unterstützende Funktionen, deren Qualität noch zu untersuchen ist. Dass die gesamte grafische Darstellung von JointJS auf SVG basiert, hat für eine responsive Gestaltung große Vorteile.

Die positive Resonanz auf den Editor in der XProc- und XML-Community bestätigt die Annahme, dass das Interesse an XProc generell hoch ist, die diese XML-Technologie jedoch für viele schwer zugänglich ist.
Auch im Kontext bereits vorhandener Dokumentationen scheint ein visueller XProc-Editor eine gute Ergänzung zu sein, um theoretische Inhalte visuell erfahrbar zu machen. So besteht zudem die Hoffnung mit einem solchen Editor einen Beitrag zum fachlichen Austausch über XProc zu leisten und bestenfalls neue Entwickler*innen für diese Technologie zu gewinnen. Denn eine größere Nutzerbasis könnte die Entwicklung von und mit XProc sehr beleben. Der Austausch von *best practices* bzw. gut funktionierenden XProc-Pipelines für verschiedene XML-Verarbeitungsschritte, wäre ein Mehrwert für die Community und könnte ein effizienteres Single-Source-Publishing ermöglichen.

Literaturverzeichnis

Ackermann, Philip (2015): Professionell entwickeln mit JavaScript : Design, Patterns, Praxistipps. Bonn, Rheinwerk-Verlag.

Bondy, J. A und Murty, U. S. R (1976): Graph theory with applications. New York, North Holland.

Bongers, Frank (2008): XSLT 2.0 & XPath 2.0: Das umfassende Handbuch. Bonn, Galileo Press.

data2Type GmbH (2018): Übersicht zu XProc: XProc-Einführung. Heidelberg. <https://www.data2type.de/xml-xslt-xslfo/xproc/> [zuletzt abgerufen am 18.03.2019].

Eisenberg, J. David (2002): SVG essentials: Producing Scalable Vector Graphics with XML. Beijing und Köln, O'Reilly.

Flanagan, David (2014): JavaScript: Das umfassende Referenzwerk. Beijing und Köln, O'Reilly.

Grupe, Wilfried (2018): XProc. In: *XML: Grundlagen, Technologien, Validierung, Auswertung*. Frechen, mitp. S. 581–585.

Hartmann, Peter (2012): Graphentheorie. In: *Mathematik für Informatiker*. Wiesbaden, Vieweg+Teubner. S. 225–249.

Haverbeke, Marijn (2018): Eloquent JavaScript. San Francisco, No Starch Press.

Krumke, Sven und Noltemeier, Hartmut (2012): Graphentheoretische Konzepte und Algorithmen. 3. Aufl. Aufl., Wiesbaden, Springer Vieweg.

Meinike, Thomas (2017): Interaktive XML-Verarbeitung im Browser mit Saxon-JS und XSLT 3.0. In: tekom, Gesellschaft für technische Kommunikation e. V., Tagungsband zur Jahrestagung 2017 (Oktober) in Stuttgart. Stuttgart. S.177-180. <https://datenverdrahten.de/PDF/Meinike_tekom_2017.pdf> [zuletzt abgerufen am 18.03.2019].

Nitzsche, Manfred (2009): Graphen für Einsteiger: Rund um das Haus vom Nikolaus. Wiesbaden, Vieweg + Teubner.

Polyzotis, Neoklis und Garofalakis, Minos (2002): Structure and Value Synopses for XML Data Graphs. In: *VLDB '02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier. S. 466–477. <https://linkinghub.elsevier.com/retrieve/pii/B9 781 558 608 696 500 482> [zuletzt abgerufen am 18.03.2019].

Pomaska, Günter (2012): Webseiten-Programmierung: Sprachen, Werkzeuge, Entwicklung. Wiesbaden, Springer Vieweg.

Ruohonen, Keijo (2013): Graph Theory. Tampere University of Technology. <http://math.tut.fi/~ruohonen/GT_English.pdf> [zuletzt abgerufen am 18.03.2019].

Siegel, Erik (vorr. 2019): XProc 3.0 Programmer Reference. XML Press.

Steyer, Ralph (2017): Webanwendungen mit ASP.NET MVC und Razor: Ein kompakter und praxisnaher Einstieg. Wiesbaden, Springer Vieweg.

Tittmann, Peter (2011): Graphentheorie : eine anwendungsorientierte Einführung. München, Fachbuchverlag.

Turau, Volker und Weyer, Christoph (2015): Algorithmische Graphentheorie. Berlin, De Gruyter.

Vonhoegen, Helmut (2015): Einstieg in XML : Grundlagen, Praxis, Referenz. Rheinwerk Computing.

Walsh, Norman; Milowski, Alex und S. Thompson, Henry (2010): XProc: An XML Pipeline Language: W3C Recommendation 11 May 2010. <https://www.w3.org/TR/xproc/> [zuletzt abgerufen am 18.03.2019].

Internet-Quellen

Alle folgenden Internet-Links wurden zuletzt am 17.03.2019 abgerufen.

[BackboneJS]: https://backbonejs.org/#Getting-started

[Can I use – let]: https://caniuse.com/#search=let

[D3js]: https://d3js.org/

[DAISY Pipeline]: https://daisy.github.io/pipeline/

[Data2Type – XProc]: https://www.data2type.de/xml-xslt-xslfo/xproc/

[DellEMC – XProc Designer]: https://community.emc.com/docs/DOC-4382

[FlowHub]: https://flowhub.io/

[Gephi – About]: https://gephi.org/about/

[Gephi – Download]: https://gephi.org/users/download/

[Gephi -Features]: https://gephi.org/features/

[GitHub – The Graph]: https://github.com/flowhub/the-graph

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

[GitHub – XProc Workshop] https://github.com/xproc/Workshop-2016-09-26/wiki/Minutes

[Greensock]: https://greensock.com/gsap

[GWT Project]: http://www.gwtproject.org/overview.html

[GoJS]: https://gojs.net/latest/index.html

[GoJS – API]: https://gojs.net/latest/api/index.html

[GoJS – Samples]: https://gojs.net/latest/samples/index.html

[Heise – PapDesigner]: https://www.heise.de/download/product/papdesigner-51 889

[JointJS – Demos]: https://resources.jointjs.com/demos

[JointJS – Docs]: https://resources.jointjs.com/docs/jointjs/v2.2/joint.html

[JointJS – OpenSource]: https://www.jointjs.com/opensource

[JointJS – MindMap]: https://resources.jointjs.com/mmap/joint.html

[JointJS – Rappid]: https://www.jointjs.com/rappid-comparison

[JointJS – Tutorial]: https://resources.jointjs.com/tutorial

[jQuery – API]: https://api.jquery.com/

[Launchpad.net – Gephi]: https://launchpad.net/gephi/0.6/0.6alpha1

[Lodash]: https://lodash.com/

[MorganaXProc]: https://www.xml-project.com/

[NoFloJS – Doku]: https://noflojs.org/documentation/

[Oxygen – XProc]: https://www.oxygenxml.com/xml_editor/xproc.html

[Saxonica – SaxonJS]: http://www.saxonica.com/saxon-js/index.xml

[SelfHTML – SMIL]: https://wiki.selfhtml.org/wiki/SVG/SMIL

[SnapSVG]: http://snapsvg.io/

[Transpect]: http://transpect.github.io/

[VelocityJS]: http://velocityjs.org/

[UnderscoreJS]: https://underscorejs.org/

[VisJS – 2017]: http://visjs.org/index.html

[W3C – 1998 – XML]: https://www.w3.org/TR/1998/REC-xml-19 980 210

[W3C – 2001 – SVG]: https://www.w3.org/TR/SVG10/

[W3C – 2005 – XML – Datamodel]: https://www.w3.org/XML/Datamodel.html

[W3C – 2006 – XProc]: https://www.w3.org/TR/2006/WD-xproc-20 060 928

[W3C – 2010 – XProc]: https://www.w3.org/TR/xproc/

[W3C – 2011 – SVG]: https://www.w3.org/TR/SVG11/

[W3C – 2017 – XSLT]: https://www.w3.org/TR/2009/PER-xslt20-20 090 421/

[yWorks]: https://www.yworks.com/

[yWorks – yFiles for HTML]: https://www.yworks.com/products/yfiles-for-html

[XML Calabash]: http://xmlcalabash.com/

[XML Prague – 2019]: http://www.xmlprague.cz/

[XProc.org]: http://xproc.org

[XProc.org – XProc 3.0]: http://spec.xproc.org/lastcall-2019-02/head/xproc/

[XProcBook – 2010]: https://xprocbook.com/book/chapter-7.html

[Xfront – XProc Tutorial]: http://www.xfront.com/xproc/index.html

Glossar

JavaScript

Durch JavaScript, die „Programmiersprache des Webs“ (Flanagan 2014: 1), können statische Webseiten um Dynamik, Nutzerinteraktion und Bewegung erweitert werden. Die objektbasierte Skriptsprache macht es möglich, mit Hilfe des *Document Object Model* (DOM)[82] HTML-Elemente anzusprechen und sie unter bestimmten Bedingungen in Bewegung zu versetzen. Inhalte und Struktur von HTML-Elementen können ver-ändert werden oder ihr Aussehen über eine dynamische Anpassung von CSS-Klassen manipuliert werden. (vgl. Pomaska 2013: 79)

JavaScript wurde 1995 erstmals vom Softwareunternehmen *Netscape* veröffentlicht und entspricht heute der, von der *European Computer Manfacturers Association* (*ECMA*) standardisierten, Spezifikation ECMAScript, die seit Juni 2018 in ihrer neunten Version, ECMAScript 2018, vorliegt. Heute ist JavaScript in allen modernen Browsern verfügbar, wobei nicht alle Funktionen neuerer JavaScript bzw. ECMAScript-Versionen in allen Browsern unterstützt werden. (vgl. Haverbeke 2018: 6f.)

Neben den bereits genannten Aspekten, ist ein weiterer Vorteil bei der Verwendung von JavaScript das Einbinden und Laden mitunter sehr komplexer, externer Datenbanken. Über die Verwendung von Befehlen in AJAX (*Asynchronous JavaScript and XML*) kann beispielsweise auf XML-Daten zugegriffen werden, oder es kann eine Abfrage an eine im JSON-Format (siehe 2.2.2) abgelegte Datenstruktur erfolgen. (vgl. Pomaska 2013: 79) Der große Vorteil dieser Vorgehensweise ist, dass die Daten nicht mehr statisch an den entsprechenden Stellen der HTML-Datei eingetragen werden müssen, sondern bei Bedarf dynamisch nachgeladen werden können. Dies erhöht insbesondere die Geschwindigkeit des Webseitenaufbaus enorm.

JSON

Die *JavaScript Object Notation* (JSON) ist ein Format zur einfachen Darstellung und Strukturierung von Daten. Die Syntax wurde von der JavaScript-Schreibweise abgeleitet. Ein JSON-Eintrag besteht immer aus einem name und einem value. Als value können die Datentypen *string, array, number, boolean,* ein *JSON-Object* und der Wert *null* verwendet werden.[83] JSON wurde speziell zur Serialisierung von Daten und als Datenaustauschsformat für die Kommunikation zwischen verschiedenen Web-Technologien entwickelt. (Vgl. Haverbeke 2018: 77f.)

82 Eine plattformübergreifende Programmierschnittstelle.

83 Vgl. [W3Schools – JSON]

SVG

SVG steht für *Scalable Vector Graphics* und ist eine XML-basierte Sprache zur Auszeichnung von Vektorgrafiken. Mit SVG lassen sich durch die Erzeugung von Elementen mit entsprechenden Attributen komplexe Grafiken erstellen, die von Web-Browsern und vielen Grafik-Programmen dargestellt werden können. Wie bei allen XML-Dokumenten ist auch bei SVG eine hierarchische Verschachtelung des Codes und somit eine Gruppierung von einzelnen grafischen Elementen vorgesehen. Im September 2001 wurde SVG 1.0 als *W3C*-Empfehlung veröffentlicht[84]. Seit 2003 hat sich SVG 1.1 als stabile Version etabliert, die von Webbrowsern am besten unterstützt wird.[85] SVG kann entweder als Code geschrieben werden, oder man nutzt einen der vielen grafischen Online- und Offline-Editoren. Auch in professioneller Software wie dem Adobe Illustrator gehört der Export von SVG-Daten zu den Standardfunktionen.
Ein großer Vorteil von SVG-Grafiken ist, dass sie wie alle Vektorgrafiken auf eine beliebige Größe skaliert werden können, ohne dabei, anders als bei Rastergrafiken, Qualitätsverluste zu erleiden. (vgl. Eisenberg 2002: 1ff.) Die Interaktion von SVG mit Web-Technologien wie JavaScript, HTML und CSS ermöglicht darüber hinaus eine sehr vielfältige Anwendung. Momentan wird SVG in der visuellen Darstellung komplexer Daten immer beliebter, wie z. B. das JavaScript-Framework *Data Driven Documents* (D3js[86]) zeigt. Auch im Bereich allgemeiner Animationen haben sich in den letzten Jahren einige Frameworks entwickelt, die mit SVG arbeiten, wie z. B. Snap.svg[87], GSAP[88] oder Velocity.js[89]. Dabei besteht bei SVG mit der Synchronized Multimedia Inte*gration Language* (SMIL) auch die Möglichkeit Animationen ohne externe Technologien zu erstellen (vgl. Eisenberg 2002: 183ff.), jedoch wird SMIL immer mehr von den genannten JavaScript-Frameworks verdrängt. Zudem hat das *W3C* die Weiterentwicklung von SMIL im April 2012 eingestellt.[90]

XML

Die *Extensible Markup Language* (XML) ist, wie der Name verrät, eine erweiterbare Auszeichnungssprache. Sie dient der strukturierten und hierarchischen Erstellung von Datensätzen. Die Daten werden als sogenannte Elemente gruppiert abgelegt und können dabei in beliebige Tiefe verschachtelt werden. Den Elementen können zudem Attribute zugeordnet werden, die zusätzliche (Meta-)Informationen beinhalten können oder

84 Vgl. [W3C – 2001 – SVG]
85 Vgl. [W3C – 2011 – SVG]
86 Vgl. [D3js]
87 Vgl. [SnapSVG]
88 Vgl. [Greensock]
89 Vgl. [VelocityJS]
90 Vgl. [SelfHTML – SMIL]

beispielsweise auf externe Ressourcen verweisen. (vgl. Grupe 2018: 12f.) Das wichtigste Prinzip von XML ist die *Wohlgeformtheit* eines XML-Dokuments. Diese ist erfüllt, wenn keine XML-Regeln verletzt werden. Dies sind unter anderem:

- Jedes XML-Dokument hat nur ein einziges sogenanntes Wurzelelement.
- Attribute dürfen für jedes Element nur einmal vergeben werden.
- Jedes Element hat einen sogenannten Start-*Tag* und einen End-*Tag*, z. B. <vorname> und </vorname>.

Im Februar 1998 wurde XML 1.0 als *W3C*-Empfehlung veröffentlicht.[91] Im Laufe der Jahre hat sich eine Vielzahl aufbauender XML-Technologien entwickelt, die z. B. der Generierung und Transformation von Dokumenten (XSLT) oder der Strukturierung und Validierung (DTD/ XML-Schema) dienen (vgl. Grupe 2018). Im Jahr 2000 wurde außerdem XHTML 1.0 als eine Neustrukturierung von HTML 4.01 veröffentlich. (Vgl. Vonhoegen 2015: 317) XHTML verpflichtet Entwickler*innen sich strikt an die XML-Regeln zu halten und ist so ein wichtiges Austauschformat im Bereich der XML-Entwicklung geworden.

XSLT

Die *Extensible Stylesheet Language Transformations* (XSLT) ist eine XML-Sprache zur Generierung verschiedener (XML-)Dokumententypen auf Grundlage von XML-Daten. (Vgl. Vonhoegen 2015: 36) In einem Transformationsszenario können aus einem Datensatz mehrere Dokumente gleichzeitig generiert werden. So ist es beispielsweise möglich anhand eines XSLT-Stylesheets mehrere HTML-Dateien mit gleicher Struktur aber abweichenden Inhalten zu generieren. Die XML-Daten können hierbei vor der Ausgabe verschiedene Transformationen durchlaufen, wie z. B. eine Sortierung oder Filterung. (Vgl. ebd.: 255) XSLT wurde 1999 in der ersten standardisierten Version als *W3C*-Empfehlung veröffentlicht. Seit Juni 2017 ist XSLT 3.0 der neue Standard, der von Michael Kay gewartet und weiterentwickelt wird.[92]

91 Vgl. [W3C – 1998 – XML]
92 Vgl. [W3C – 2017 – XSLT]