# JDF File Viewer

Dr. Thomas Hoffmann-Walbeck, Dr. Barbara Dörsam, Mr. Tobias Zeller*

**Keywords:** JDF, java, software, workflow

## Abstract

A software is presented that reads JDF files and shows (part of) the content in an Explorer-like fashion. The main objective of the software is to visualize the most important information inside a JDF file to people that are not JDF specialists. Therefore, the hierarchical structures of JDF-nodes are rearranged to make the content easier to understand for human beings. A database is used to give additional information about the nodes and the resources to the user.

## Introduction

With the help of the Job Definition Format (JDF) (CIP4, 2008) one can define the characteristics of almost every print product as well the production steps that are needed to produce it. This might justify the fact that JDF files are difficult to read even though they can be easily opened with a browser or an XML editor (Figure 1).


***Figure 1:*** *Small part of a typical JDF file, displayed by a browser (left) and a XML-Editor (right)*

*Stuttgart Media University, Nobelstr. 10, 70563 Stuttgart, Germany

There are some reasons for that:

- JDF files consist of highly structured JDF nodes which describe very different things like the product itself, the production processes or some kind of process collections. One might need to evaluate different attributes of the JDF node in order to determine what kind of JDF node is present. Sibling nodes in the JDF-tree can be of different kind, like a description of a product part (e.g. cover or content), a process or a group of processes.

- The dependencies between the production processes are only given indirectly by the input- and output resources that are involved. In particularly, the processes are not ordered by the more common categories of prepress, press and postpress.

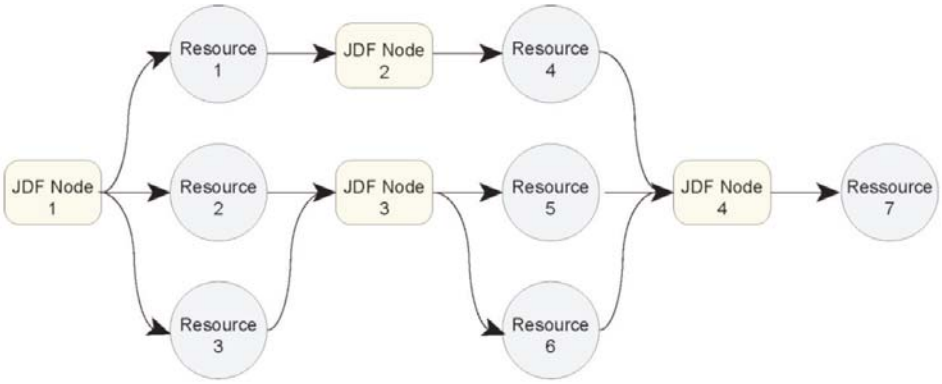


***Figure 2:*** *Example for dependencies between JDF nodes and resources*

- Input and output resources are normally attached to each JDF node. These resources, however, need not to be stored in those nodes that are using them. They are referenced by IDs. Also, resources can reference (or contain) other resources.

- Some of the resources might be partitioned. For example, resources like ink, plates and folding sheets are such structured resources. They can be viewed for the whole job or for specific parts only. The *Ink*-Resource for instance might get partitioned to different signatures, different sheets, front and back of each sheet and to each separation of both sides. The *FoldingParams*-Resource, however, can be only partitioned to different signatures and sheets.

- In general, a JDF file contains a lot of (necessary) technical data concerning modifications of the file, data types and the like. Also, a typical JDF file is quite extensive. JDF files for entire print jobs are normally over 100 A4 pages long.

Thus, for most people JDF is like a black box. This is all right in many cases, but if somebody needs to analyze the JDF-interface between two or more JDF-modules from different vendors it does not suffice. An analysis might also be necessary if one plans to setup a new workflow or tries to resolve a severe error that might occur in an existing workflow.

The following example should explain what is meant by that: A JDF file for a folding machine can either contain basic administration data like the job-ID, the customer name and the like or additionally presetting data, which is stored in the JDF resource FoldingParams. With the latter the file might simply contain the imposition scheme number according to the Fold catalogue that is listed in the JDF specification or it might specify the exact folding sequence and positions explicitly. An imposition scheme, however, is not enough to define the folding positions accurately (for example, if binding flaps are involved). Thus, for a workflow implementation in a print shop it makes a big difference what kind of JDF is sent to a device like a folding machine as well as what kind of JDF structure the folding machine can handle. But the vendor of a JDF module will call the software "JDF savvy" in any case and so the workflow managers have a hard time to evaluate the usefulness of a specific JDF interface before it is implemented. And even if after the installation and the presetting one of the devices does not work as expected it is difficult to find out the reason for it. It could be caused by the JDF generating module, that does not support some details, or by the JDF reading device or by both.

This motivated us to write software for integrators, workflow managers and students to give them an idea about the contents of JDF-files without assuming that they know much about the JDF-terminology. Our objective was to write a JDF-viewer, explaining the processes and resources that are involved, resolving the references, re-organizing the processes to the categories prepress, press and postpress, emphasizing import properties of the processes and resources and omitting all less important details in the same time.

## Methods

The software has been written in Java, using the standard JAVA-library "JDFLibJ-2.1.4a.62" provided by the CIP4-organisation. Furthermore, we used a database management system to store information and textual descriptions for all 112 JDF-processes and 226 JDF-resources.

The software application consists of three parts:

- JDF import. Here certain nodes and resources as well as some of the values of their attributes are stored in a internal data structure.

- Evaluation of the internal data structure. Here the information in the database is used to analyze the JDF. In particular all processes are assigned to the categories prepress, press and postpress. For each process and each resource the standard text for describing the elements is retrieved from the data base. Individual properties of certain resources are generated, showing for instance certain properties of some partitioned resources.

- Generation of the graphical output (as well as a textual one), showing the tree structure and some information about the processes and resources. For this the component JTree from the package javax.swing is used.

## Results

The GUI of the JDF File Viewer can be seen in Figure 3. The lower part of the window shows the typical tree structure of the nodes and resources. One can navigate through the tree similarly to Windows Explorer. The top part displays some information concerning the folder that has been selected in the lower window. When starting the Viewer only the print job's name ("Students Brochure") and in the next level "Product Description" and "Production" are shown. The folder "Product Description" contains general information about the JDF file as well as the nodes and resources of the JDF file that describes the projected print job, while "Production" let you see the actual processes and resources concerning the production.

Clicking on the folder "General Description" one gets to the level of "Final Product" and "Partial Products". The "Final Product" represents the end product that the customer ordered and which consists of partial products. In this sample job there are three partial products: "Cover", "Content" and "Foldout 1".

The folder "General Information" has been selected in Figure 3 and some statistical data and other properties of the JDF file in general are displayed. First the number of JDF nodes in the file is listed. This number gives some hint about the complexity of the file. Next the number of Product nodes is given, representing the final product and product parts. The production nodes follow and are classified in prepress, press and postpress. But not every node can get allocated to one of the three production areas. In a typical JDF file there are, in fact, quite a number of private process nodes (11 in our example). Since these are proprietary nodes designed by some vendor they cannot get analyzed automatically. The same holds for private process groups. Furthermore, the JDF file can contain process groups and combined processes which describe processes in at least two areas. A node representing the actions of a digital print press can, for example, contain processes in prepress, press and postpress. These process groups and combined processes are labeled with "non-classified" in the GUI. It should be added that a high number of private nodes points towards a high probability that the JDF file is not designed for an open and interface to another external module.

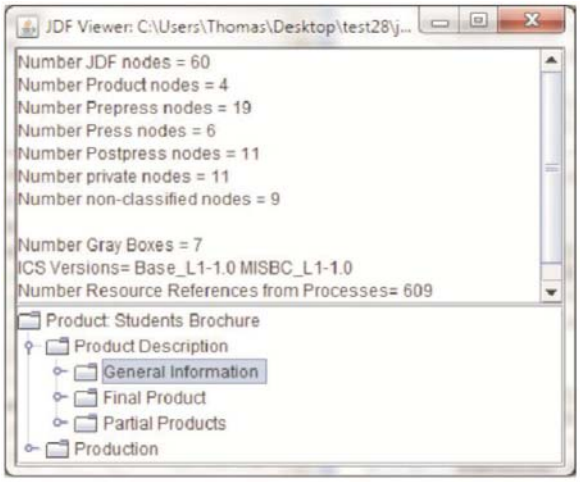


***Figure 3:*** *General Description of the JDF File*

Gray Boxes are containers of several processes with a specific goal. They are mostly written by a MIS, which does not know all technical details that are necessary to run processes. That is, Gray Boxes, in general, do not contain or reference all their resources and thus they can not be executed and must be transformed to processes before.

ICS stands for *Interoperability Conformance Specification*". These papers describe different sets of properties that a JDF writer and a JDF reader for a certain class of devices are supposed to support. A system integrator should check these entries first if he or she analyzes the compatibility between two JDF modules. Finally, the number of references of processes to resources gives another hint to the complexity of a file.

Figure 4 shows the situation when the folder "Final Product" and Partial Products" are opened: they contain input- and output resources. Most of these resources are called "intent resources" in the JDF terminology, because they define the intentions how a product (part) is planned. In the information window one can read two different entries concerning the resource LayoutIntent. The sentence "This Resource records the size of the finished pages for the product component as well as the number of pages for a product (part)" comes from the above mentioned database. For each process and each resource there is as short text in the database explaining the element (see Figure 8). The software connects the names of the processes and resources in the JDF file with the appropriate data base entries. The number of pages (96), however, is directly taken from the JDF file. The software reads properties of some attribute values or sub-elements to describe a resource more precisely. This has to be done individually for a specific type of resource.

Not all input resources are intent resources, especially not those from the "Final Product". Here we see three Components as input resources, which are, in fact, the output resources of the three partial products. That is, for the final products one needs the cover, the content and foldout.

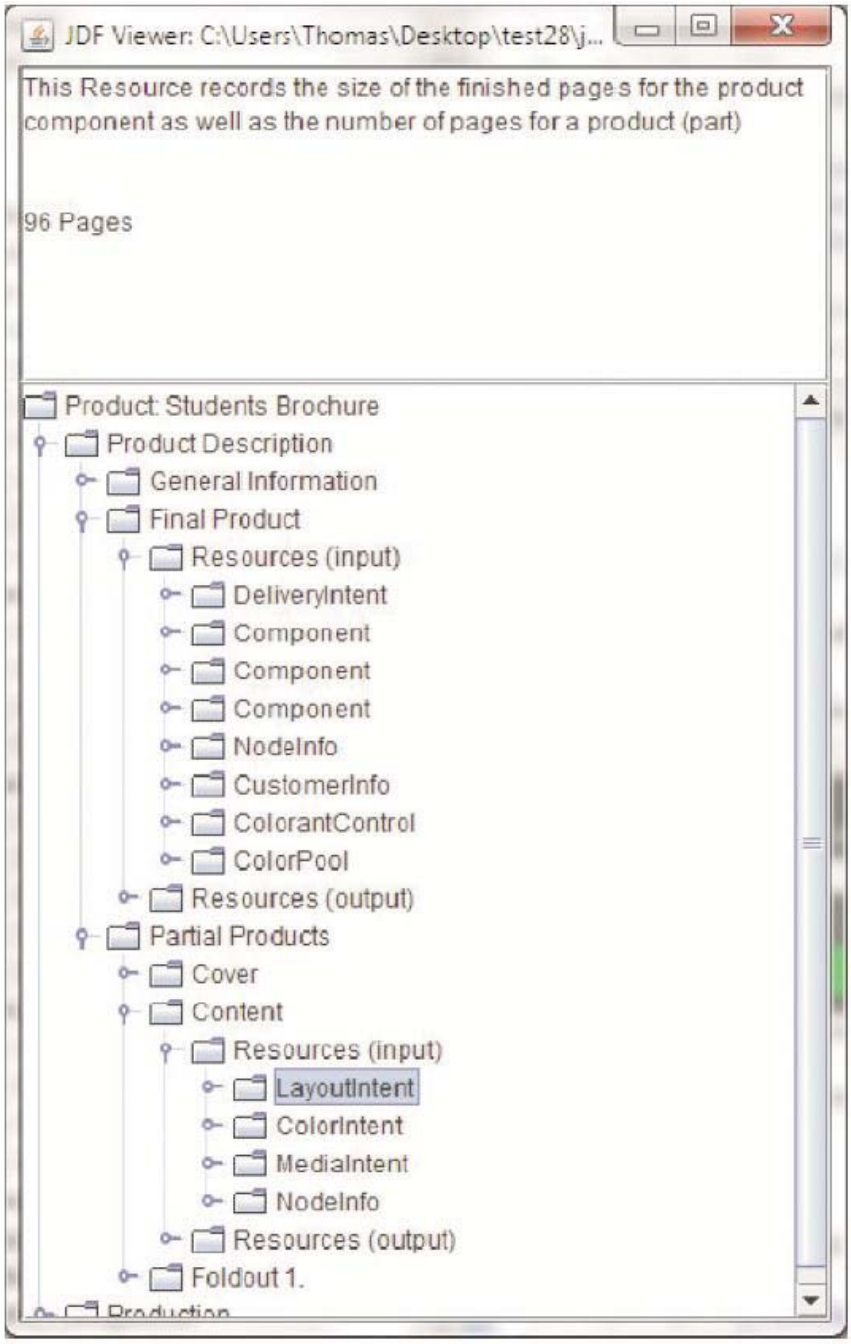


***Figure 4:*** *Input resources of the finished product and the partial products*

Opening the production folder, one gets the folders "Final Product" and the "Partial Products" in the next level once again (see Figure 6). Stepping one level further down, however, one does not see the resources right away like it was the case with the folder "Product Description", but rather has an intermediate layer, namely the production areas "Prepress", "Press" and "Postpress". These categories are very familiar to everybody in the graphical industry and they make it easier for the user to navigate. Since these categories, however, are not explicitly stated in the JDF-notation, our software has to assign to each node one of the three during the analysis of the data. Figure 5 shows the idea.

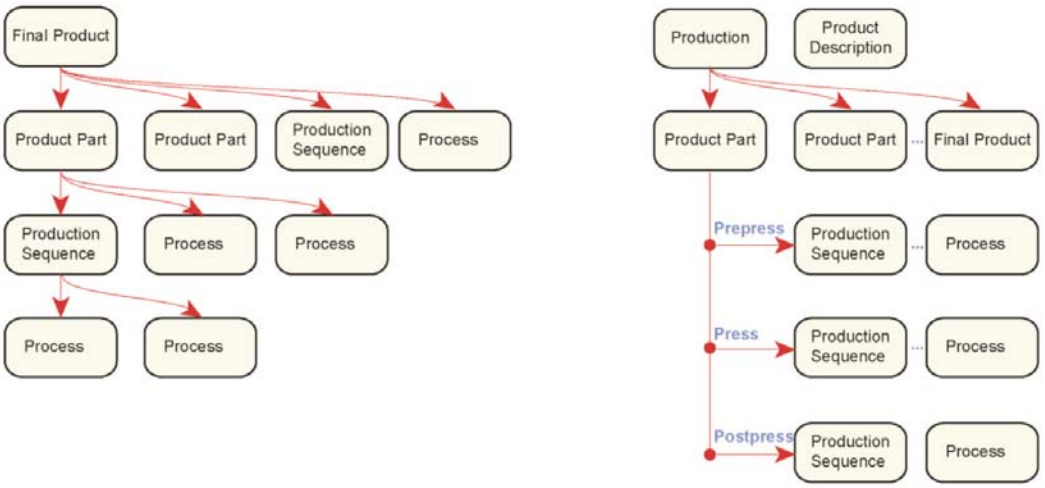


***Figure 5:*** *The JDF structure (left) is rearranged in our software (right)*

Partial products typically need prepress, press and postpress activities in order to produce them. In the very end the final product is assembled with all partial products. Thus, the production of a final product might also need postpress activities for the assembling task. In addition it might incorporate some prepress processes to prepare the PDL-pages for all product parts.

In Figure 6 the tree of the folder "Cover" is further opened and one can observe that the category "Press" contains only one process, i.e. *ConventionalPrinting*. For this process the input resources are visible.

The explanations to the processes and resources can be read again in the top part of the window. The information that is displayed is fed by three different sources, where the first two have been discussed above with Figure 4.

- the database,
- individual properties of the elements to describe them more precisely.
- certain JDF internal, optional attributes like "DescriptiveName",

Concerning the resource "Device", the sentence "Information about a…." comes from the database, the MIS has assign "Speedmaster CD 74-6L (HdM)" to the value of the attribute *DescripveName* and the term "CD 74 C 59.5*33.5" has been extracted specifically for the *Device* resource from the JDF-file.

Nodes are tree-like structured. The structures of the resources are also not flat but can contain or reference other ones. Partitioned resources, in particular, are hierarchically structured resources. For example, the *FoldingParams* resource defines the folding parameters and may be partitioned for different signatures and sheets. Thus the "overall" *FoldingParams* resource can contain several

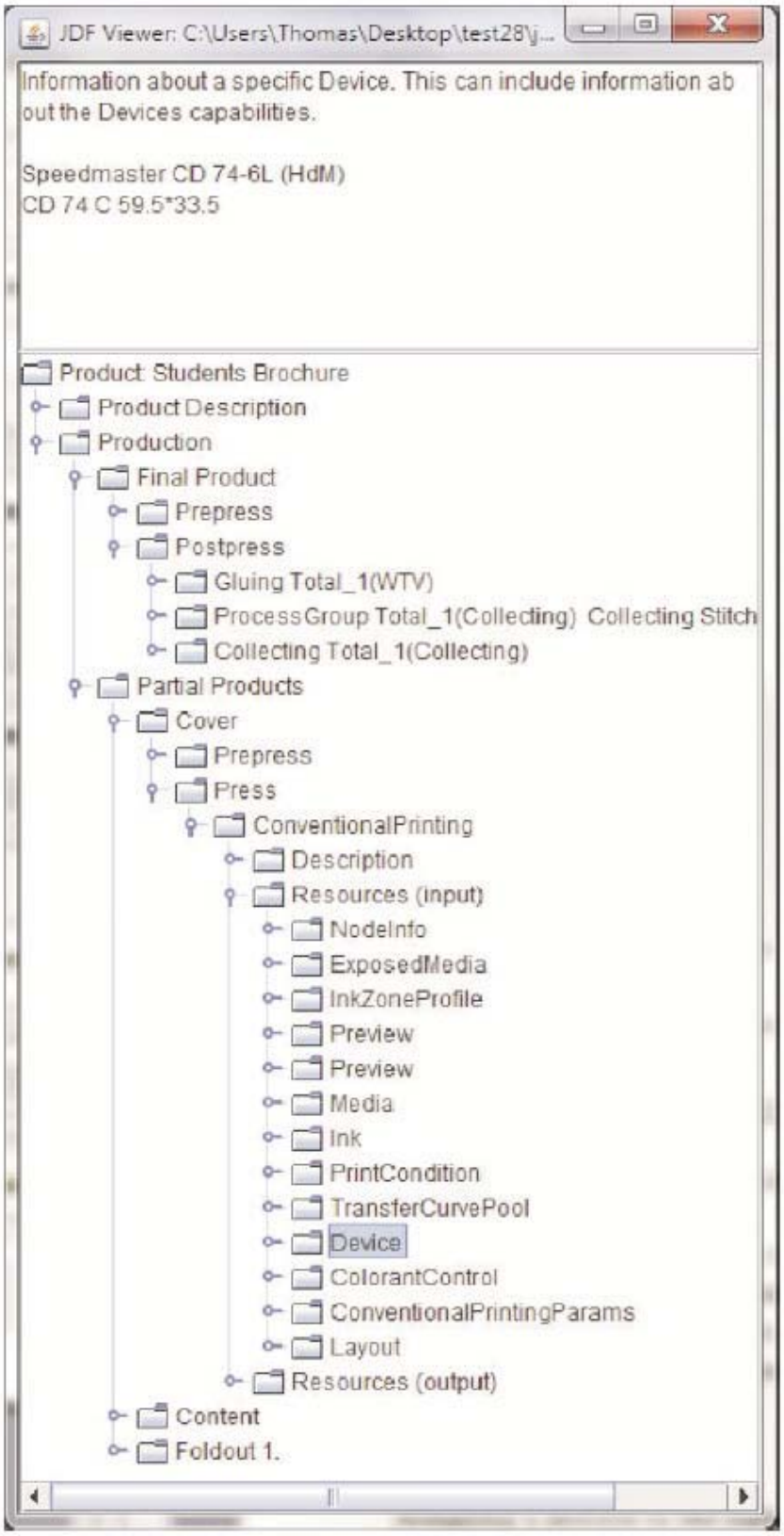


***Figure 6:*** *Resources for the production are ordered by the common notions of prepress, press and postpress*

*FoldingParams* resources for all signatures, and a *FoldingParams* resource for each signature can again include several *FoldingParams* resources for each sheet. Thus if one wants to read the folding schemes (i.e. the *Folding Catalog* number) for each sheet, one has to descent to the lowest hierarchy level. The JDF Filer Viewer flattens this structure and lists all these schemes one by one. Figure 7 is an example of the partitioned resource *FoldingParams*, showing *Folding Catalogue* number F2-1 for the first two signatures.

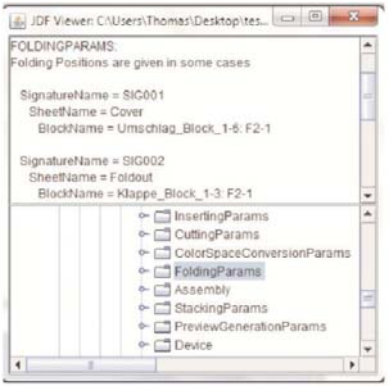


***Figure 7:*** *Partitioned Resource FoldingParams*

JDF nodes and resources can contain other elements. The precise specification of the folding sequence and folding positions can be found, for example, in the optional *Fold* element inside the *FoldingParams* resource. The JDF File Viewer software looks for these elements and informs the user, if such an element has been found in one of the resources. In the information field of Figure 7 the sentence "Folding Positions are given in some cases" is displayed, if such Fold elements have been found.

## Implementation

Assigning a process to the right production area is done with the help of a database. In the database table there simply is one column specifying the "Production Area" for each node. Figure 8 shows some part of this table. The numbers 1, 2 and 3 denote the areas prepress, press and postpress respectively. The column "kind" distinguishes between the different kind of JDF-nodes, like process nodes, product nodes, process group nodes or combined process nodes. The values of these columns are filled in manually. The column "Type" contains the value of the *Type*-Attribute of JDF-nodes. All processes, for example, have different entries for this attribute. Finally the entries in the column "Description" give a short explanation to each process.

There is similar table for the 218 resources that are specified in the JDF specification. This table, however, does not have the columns "Production Area" and "kind". Both data base tables are read in and stored into internal structure, when the software starts. After a JDF file has been selected, the file gets parsed and the root node gets determined using the standard CIP4-library. How this works can be looked up at (Hoffmann-Walbeck & Riegel, 2012). Next all nodes and the references resources are parsed and certain attributes are stored into linear lists (like *Type, Types,*

*DescriptiveName*, but also technical things like the node level in the tree hierarchy, the id and so on). These entries in the list are evaluated to find out the Production area of a node (which is easy for simple processes by checking the data base, but harder for *Combined Processes* or *Process Groups*). Furthermore, *Partitioned Resources* get handled as well for some specific resource types in order to find out particular attributes. In the example above (see Figure 4) the number of pages in the product part "Content" is found in the sub-element *Pages*, which has the attribute *Preferred*. Finally, the graphical output gets initiated.

In the following the handling of *Partitioned Resources* is explained on the source code level: It has been mentioned above, that *Partitioned Resources* are hierarchically structured, i.e. the top level of the resource contain the next level and so on. Figure 9 shows these sub-resources for a very simple *FoldingParams* in a graphical manner. Each *Signature* could have, in fact, more than one Sheets, of course, and each *Sheet* could have more than one *BlockNames*.

| id | Type | Description | Production Area | Kind |
|---|---|---|---|---|
| ... | ... | ... | ... | ... |
| 48 | Stripping | The Stripping Process specifies the Process of translating a high level structured description of the imposition of one or multiple Job Parts or part versions to a specific sheet layout | 1 | 1 |
| 49 | Tiling | The Tiling Process allows the contents of Surfaces to be imaged onto separate pieces of media. | 1 | 1 |
| 50 | Trapping | This Process modifies PDL files to compensate misregistration on presses, i.e colorant edges are made overlap slightly | 1 | 1 |
| 51 | Conventional-Printing | This Process covers several conventional printing tasks, including Sheet-Fed printing, Web Printing, Web/ribbon coating, converting and varnishing. | 2 | 1 |
| 52 | DigitalPrinting | DigitalPrinting is a direct printing Process; the data to be printed are not stored on an extra medium (e.g., a printing plate or a printing foil), but instead are stored digitally. | 2 | 1 |
| ... | ... | ... | ... | ... |
| 73 | Folding | Buckle folders or knife folders are used for Folding Sheets. | 3 | 1 |
| 74 | Gathering | In the Gathering Process, ribbons, Sheets or other Component Resources are accumulated on a pile that will eventually be stitched or glued in some way to create an individual Component | 3 | 1 |
| 75 | Gluing | Gluing describes arbitrary methods of applying glue to a Component. | 3 | 1 |
| ... | ... | ... | ... | ... |

***Figure 8:*** *Database table concerning the JDF nodes*

Because resources can be partitioned in different ways and down to different depths, there is an attribute in the top resource, defining to which extend the resource can be partitioned. This attribute is called *PartIDKeys*. The value

*"SignatureName SheetName Side Separation"*

is an appropriate partition concerning the resource *Ink*, for example. On the other hand the value

*"SignatureName SheetName BlockName"*

is a correct partition for the resource *FoldingParams*. Each folding sheet of a print sheet is identified by *BlockName*. These sub-resources are implemented in JDF by XML-elements.

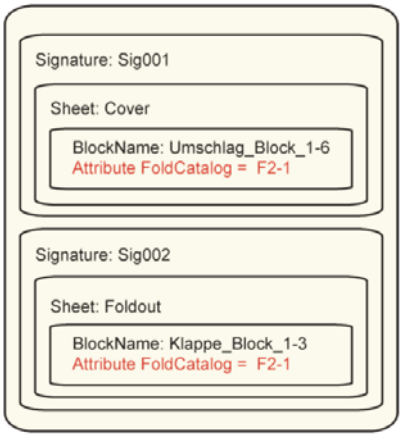


***Figure 9:*** *Simple Partitioned Resource*

If one wants to read a certain attribute value of each partition one has to browse through all sub-elements. For example, if we like to find the attribute *FoldCatalog* (=folding scheme according to the *Folding Catalogue*) of a *FoldingParams* resource for each BlockName, one has to follow the branches of the resource-tree and read the *FoldCatalog* each time for the deepest resources (the leaves). Moreover, since attributes can be inherited from other resources higher up the tree, one actually has to consider the attribute of the other resources, that is, those that are not leaves. In some cases, one is not interested in any particular attribute of a *Partitioned Resource* at all, but rather only in the structure of the *Partitioned Resource*. This is the case, for example, for the resource *Ink*.

Figure 10 lists the source code that handles the *Partition Resource*. The method processPartitionedResource (line 7) generates the sort of text that is shown in Figure 7. For doing so, a partitioned JDFResource named *"Resource"* is needed as input as well as the desired attribute *"attr"*. If the attribute is an empty String, only the structure of the *Partitioned Resource* is relevant. Otherwise, the attribute value is attached to the output text, which is presented in Figure 7. Furthermore *"PartIdKeys"* is input and – for technical reasons – the actual depth of the resource-tree.

Lines 2 and 5 in the Figure 10 show the calls for the *FoldingParams* and *Ink* (neglecting the return values).

The final level of partition can (theoretically) be any non-negative integer. We therefore use the method recursively. Since we want to collect the appropriate text from each stage, we need to have the *StringBuffer* text not only as an output of the method but also as an input.

In line 15 the partition names are looked for. In line 18 the value of the desired attribute is appended to the text. In lines 27 to 37, finally, we are looking for children-nodes with the same name and apply the method to them too (line 38).

## Discussion

This presented software tool is a prototype only. It might be enhanced in the future. One option would be to check more attributes of individual processes or resources. One other option would be to pop up the JDF source code, whenever clicking a process or a resource. Finally, it would be worthwhile to make the software more robust and usable for everyone.

The design and the programming have been carried out within a research project at the University of Media, Stuttgart.

## Selected Bibliography


CIP4, (Cooperation for Integration of Processes in Prepress, Press and Postpress), 2008, "JDF Specification, Release 1.4", www.cip4.org

Hoffmann-Walbeck T. and Riegel, S., 2010, "Analysis of JDF files", Advances in Printing and Media Technology, Vol. XXXVIII (iarigai, 2010), pages 387-393

Hoffmann-Walbeck T. and Riegel, S., 2012, "JDF Workflow", published by Printing Industries of America, ISBN ISBN:9780883627181


```
1  //Call for Resource FoldingParams and Attribute FoldCatalog
2  processPartitionedResource(Resource,"FoldCatalog",sb,PartIDKeys,0);
3
4  //Call for Resource Ink and no Attribute
5  processPartitionedResource(Resource,"",sb1,PartIDKeys,0;
6  …
7  private StringBuffer processPartitionedResource(JDFResource Resource,String attr1,
8    StringBuffer Text, String PartIDKeys, int PartitionDepth){
9    String Name = Resource.getNodeName();
10   String Partition;
11
12   //Top resource is only container for sub-resources
13   if (PartitionDepth > 0){ // not top resource?
14
15     Partition = WordUtils.findWord(PartIDKeys,PartitionDepth-1);
16     Text.append("    "+Partition+ " = ");
17     if (Resource.hasAttribute(Partition,null, false)){ //append attribute, if it exists
18       Text.append(Resource.getAttribute(Partition));
19     }
20   }
21   if (!attr1.isEmpty()){ // empty if only structure of resource relevant
22     if (Resource.hasAttribute(attr1,null, false)){
23       Text.append(": "+Resource.getAttribute(attr1)); // output the desired attribute
24     }
25   }
26
27   if (Resource.hasChildElements()){ // Do sub-resource exists?
28     KElement[] KElementArray = Resource.getChildElementArray();
29
30     for (int k = 0; k < KElementArray.length; k++){ // for all sub-resource ...
31       KElement Ke = KElementArray[k];
32       String ChildName = Ke.getNodeName();
33
34       if (Name.equals(ChildName)){ //sub-resource has same name as parent resource?
35         JDFResource ChildResource = (JDFResource)Ke;
36         Text.append("\n");
37           //recurive call to find desired attribute in sub-resource
38         processPartitionedResource(ChildResource,attr1, Text,PartIDKeys, PartitionDepth+1);
39       }
40     }
41   }
42   else
43     Text.append("\n");
44
45   return(Text);
46 }
```

***Figure 10:*** *Java Source Code for processing Partitioned Resources*