Web-based Cooperative Document Understanding

Nicolas Roussel, Oliver Hitz and Rolf Ingold

DIUF, University of Fribourg
Chemin du Musée 3, 1700 Fribourg, Switzerland
{nicolas.roussel, oliver.hitz, rolf.ingold}@unifr.ch

Abstract

This paper presents our ongoing work on the design of a Web-based framework for cooperative document understanding. We begin by exposing our motivations for designing a new document understanding environment. We then describe the different levels of cooperation we intend to support and how Web technologies can help us in this respect. Finally, we present Edelweiss, the framework we are currently developing based on this approach.

1. Introduction

The document understanding research community has traditionally focused much of its efforts on methods and algorithms for automated document segmentation, recognition and classification. However, these methods and algorithms tend to be developed as the focus of highly specific research projects. Few initiatives have tried to combine them into a generic framework. We believe that such a framework would benefit to both researchers and end-users, by supporting the rapid prototyping and incremental development of document processing applications.

In his thesis [1], Bapst proposes an interesting modular approach to document understanding inspired by multi-agent architectures. Yet, his approach still faces the problem of choosing the right implementation platform. We believe that the Web is the right platform for such a modular framework. More and more applications and programming toolkits are able to use URLs for resource and service naming, HTTP for command and data transfer and HTML for content description. Moreover, the growing number of people and organizations adopting Java and moving from HTML to XML for content description suggests that the move to the Internet and interoperable standards will continue.

Fully automatic document processing is still restricted to only a few specific applications, and the cost of post-corrections is often underestimated. Instead of being designed to replace the human, we believe that document understanding systems should be *cooperative* ones, allowing the users and the system to work together. Along with our move toward a generic and modular framework, we believe that this cooperation model can be used as a unifying principle for the design of the internals of the system, the interactions between a single user and the system, and the interactions between users through the system.

This paper presents our ongoing work on the design of a Web-based cooperative document understanding framework. The rest of the paper is organized as follows. In the first section, we describe the three levels of cooperation we intend to support and how Web technologies can help us in this respect. We then present Edelweiss, the framework we are currently developing based on this approach. Finally, we conclude with directions for future research.

2. Web support for cooperative document understanding

Over the last few years, the Web has become a central access point to many applications and services on the Internet, making them easily and permanently accessible. As Web standards and protocols helped people integrating these applications and services in a common framework, we anticipate they can help us create a generic framework for cooperative document understanding. In this section, we describe how Web technologies can be used in such perspective to support cooperation on three levels: inside the system, between a user and the system and between users.

2.1. Intra-system cooperation

The segmentation, recognition and classification of a document is a cooperative process in itself: image segmentation techniques are combined with character and font recognition to get a text version of the document, which is often used for a semantic analysis. As all these different techniques are evolving, we believe that document understanding system design would benefit from a modular approach that would open the system and make it easier to design, develop, maintain and upgrade.

In previous work, we have described the potential of XML technologies for representing and visualizing document recognition results [4]. Like other researchers [10, 8, 6], we think that the use of XML tools and XSLT [2] stylesheets facilitates the exchange of data and supports a modular approach. However, we think that XML could be used to describe not only the results of a document processing, but also the processing itself. In this perspective, we propose to consider XML as a high-level scripting language for gluing together elementary operations implemented in another language. Such a scripting approach has already proven useful in many applications, including document understanding[3], and we think that XML can take it further by mixing parameters, commands and results in a single document.

HTTP is a simple, open-ended protocol for accessing network resources and services. It provides applications with a set of standard methods and status codes corresponding to different semantics, including retrieval or posting of data, success notification or redirection. Applications are not restricted to the standard methods and status codes but can also define their own ones and thus extend the protocol. This flexibility of HTTP makes it suitable for connecting together the different modules of a document understanding system. Using one of the many HTTP-aware languages, such as Java, new modules or wrappers around existing software can be easily added to the system, supporting the rapid extension, modification or replacement of its functionalities.

2.2. Cooperation between the user and the system

As we already stated, we believe that the goal of document understanding systems should not be to *automate* the document processing, but rather to *informate*[7] it: to provide people with the knowledge they need to make informed, intelligent decisions that would ease the processing of one document and allow the system to improve with use.

Determining the right granularity of interaction between a user and the system is difficult and highly subjective. Repeated processing of similar documents, for example, should lead to quasi-automation as both the user and the system learn the specificities of the document class. However, the correct processing of a single document might require interactive validation of every step. Using XML as a scripting language offers a very flexible way to compose such interactive stages with automated operations. User interactions can be added or removed from the "script", depending on the task to achieve.

The validation of document processing operations usually relies on an interactive representation of the document and the intermediary results. Determining the right presentation and interaction techniques to use is also a difficult task. Trade-offs must be found between simplicity and efficiency for both the user and the programmer, whose interests are often conflicting. The combined use of XML and HTTP offers a wide range of solutions to this problem, from using standard Web browsers and XSLT stylesheets to developing custom client applications.

Letting users interact with the system by simply pointing their usual Web browser to the appropriate URLs favors the integration of the system into their existing work practices. They don't have to learn anything new, or even to start a new application. Naturally, a browser-only approach would be very limited: feedback or direct manipulation of objects is hardly an option. However, Java applets and independent applications can easily overcome these restrictions and supplement the basic HTML-based interfaces with more complex ones.

2.3. Cooperation between users

Documents are often meant to be shared or exchanged between people. When designing a document understanding system, one shouldn't ignore this collective aspect of document use: when submitting a document or a collection of documents to the system, a user might want to be able to share or exchange the results and the meta-information about the processing (e.g., parameter configuration) with other people.

The Web strongly encourages reuse and promotes a culture in which tailoring is the norm. By using standard Web languages, protocols and tools as a software infrastructure for building our system, we make it accessible from a wide range of existing applications such as browsers and editors. Moreover, the use of simple text-based representations facilitates the collaboration between users: anyone can view the source of an HTML document and copy/paste parts of it, sometimes without fully understanding the details of how it works. We anticipate that the use of XML to describe parameters, commands and results of the document processing will allow users to share experience by exchanging XML fragments, like they already exchange HTML ones.

3. The Edelweiss framework

Edelweiss is a cooperative document understanding framework we are developing using the approach described in the previous section. In this section, we describe the architecture of this framework and the basic services it already provides.

3.1. General overview

In its current state, Edelweiss consists of about 7500 lines of Java code and 120 classes. The whole framework relies on the following concepts:

jobs are elementary document processing operations;

eDocuments are structured text documents that use an XML syntax to describe a physical source (a PDF or PostScript file), a succession of jobs and the data sets produced by these jobs;

repositories provide hierarchical storage and retrieval mechanisms for eDocument collections;

schedulers are processing units that can load an eDocument from a repository, execute some of its jobs and save it back on the repository.

The following scenario illustrates the interactions between users, eDocuments, jobs, repositories and schedulers:

> The user creates an eDocument, possibly from an existing template. This eDocument specifies the physical source and a series of jobs to execute.
>
> The user stores the eDocument on some repository and asks a scheduler to process it.
>
> The scheduler retrieves the eDocument and executes all the jobs in sequence. If it cannot handle a particular job for some reason (e.g., due to a lack of resource), it stops. Once the processing is done or stopped, the scheduler updates the eDocument on the repository to reflect the possible changes.
>
> The user is notified of the process termination by the scheduler and can get the final version of the eDocument from the repository.

Edelweiss jobs are separated in two classes: *operators*, that can be executed fully automatically, and *interactors*, that require user interaction. All operators and interactors are implemented as independent code modules that manipulate an eDocument structure. Every job can modify the results produced by previous ones and store their own results in the eDocument structure for the following ones.

Appendix A shows a sample eDocument that has been partially processed. This eDocument is organized in three parts. The first part, enclosed with `datasources` tags, specifies the physical document source (`/test.pdf`). The second part, enclosed with `processing` tags, describes the list of jobs to be executed (the first two jobs are marked as already done). Finally, the third part, enclosed with `data` tags, contains the results that have been produced so far (only one data set in this case).

The simple versioning system used in eDocuments provides us with a very flexible way of validating intermediary results. During interactive jobs for example, a user could choose to go back in history and re-run some previous jobs after manually changing some parameters to correct some errors. She could also simply point out errors and let a learning algorithm try other configurations, this procedure being iterated until she gets satisfying results.

Edelweiss provides several classes and methods to facilitate the development of operators and interactors. Bitmap images of physical document pages, for example, can be obtained from an eDocument at any resolution[1]. Edelweiss also provides methods to store and retrieve data into/from the eDocument structure.

The following operators and interactors have been implemented:

- a top-down segmentation operator, that uses standard Run Length Smoothing Algorithm (RLSA) and connected components analysis on bitmap images of the document to compute a hierarchical tree of rectangular areas corresponding to blocks, lines, words and signs;
- a character recognition operator, that uses Xerox ScanWorX to extract from bitmap images the text corresponding to a specified level of the hierarchical tree;
- another segmentation and character recognition operator that extracts strings and their bounding box from PostScript or PDF data;
- a simple XML editor, that allows to modify an eDocument by directly editing its XML code;
- an eDocument viewer, a very simple interactor that presents to users the logical elements obtained from the segmentation operators overlaid on the physical image of the document;
- a generic interactor based on XMIllum [5], that we describe in details in section 3.4;
- a Mail gateway operator, that can be used for asynchronous notification of automated jobs completion.

3.2. Edelweiss servers

Edelweiss defines two abstract `Repository` and `Scheduler` classes. Derived from these classes, `LocalRepository` and `LocalScheduler` provide a filesystem-based repository and a basic scheduler implementation. All structured data exchanged with local repositories and schedulers is represented by DOM [11] trees that can be serialized to XML.

[1]Multi-pages TIFF images are generated upon request from the PDF or PostScript file using Ghostscript. Once generated, these images are cached in memory and also saved on the repository for later use.

Repository content listing, for example, is serialized to XML code such as:

```
<container-listing name="/" path="/">
  <entry name="tdseg.edoc" path="/tdseg.edoc"/>
  <entry name="test.pdf" path="/test.pdf"/>
</container-listing>
```

Two classes, `RepositoryMessageServer` and `SchedulerMessageServer`, wrap the local repository and scheduler implementations with servers able to decode HTTP requests and send serialized DOM trees to make their services remotely accessible. `RepositoryMessageClient` and `SchedulerMessageClient`, derived from `Repository` and `Scheduler`, implement the HTTP client code needed to access these remote scheduler and repository services (i.e., the code needed to send requests and rebuild DOM trees from XML). Factory methods are used to create local or remote instances of repositories and schedulers from URLs such as `file:/home/edelweiss/` or `erms://ufps7.unifr.ch/`. Combined with HTTP and XML, this use of URLs supports transparent access from any Edelweiss application to repositories and scheduler, whether local or remote.

A repository server and a scheduler server application have been implemented, the later being restricted to the solely execution of automated jobs (i.e., operators). These two applications make it possible to have multiple repositories and schedulers running on different machines, which can be useful for sharing files between various groups of users and for dealing with hardware or licensed software that are bound to a particular machine.

3.3. Interacting with Edelweiss servers

In addition to the repository and scheduler servers, we have developed Asterix, an application designed to provide users with an interactive graphical interface to these servers. Asterix presents the content of a repository specified on the command line (Figure 1). Double-clicking on repository entries executes a default action, such as opening an custom viewer for TIFF files or launching an external application for PDF and PostScript files. Several “links” on the right side of the view allow users to open an eDocument with one of the interactors available and to submit it to a scheduler.

The state of a chosen scheduler can be displayed in a separate window (Figure 2). This second view gives information about the various eDocuments being processed by showing the list of past and present jobs and allowing users to get more details on a particular job by clicking on it.

As a complement to Asterix, we are also exploring the use of Web browsers as Edelweiss interfaces. By adding a few lines of code in the `RepositoryMessageServer` and `SchedulerMessageServer` classes, we were able to handle standard browsers' requests and to serialize DOM trees to HTML using a simple XSLT stylesheet. These small

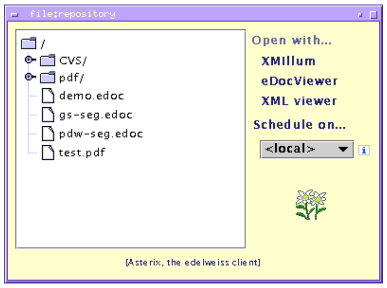
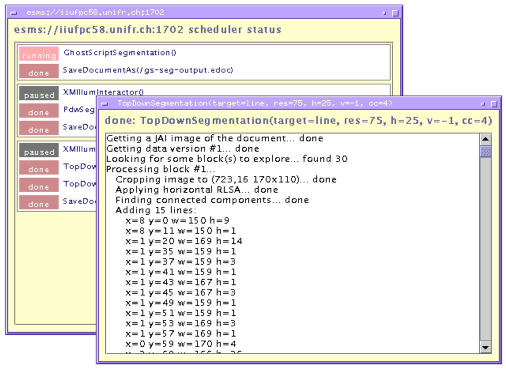


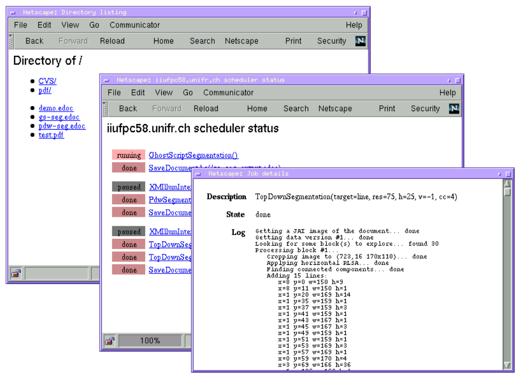
Figure 1. Asterix, repository view


esms://iiufpc58.unifr.ch:1702
esms://iiufpc58.unifr.ch:1702 scheduler status
running
GhostScriptSegmentation()
done
SaveDocumentAs(/gs-seg-output.edoc)
paused
XMIllumInteractor()
done
PdwSeg
done
SaveDo
paused
XMIllum
done
TopDow
done
TopDow
done
SaveDo
TopDownSegmentation(target=line, res=75, h=25, v=-1, cc=4)
done: TopDownSegmentation(target=line, res=75, h=25, v=-1, cc=4)

```
Getting a JAI image of the document... done
Getting data version #1... done
Looking for some block(s) to explore... found 30
Processing block #1...
  Cropping image to (723,16 170x110)... done
  Applying horizontal RLSA... done
  Finding connected components... done
  Adding 15 lines:
    x=8 y=0 w=150 h=9
    x=8 y=11 w=150 h=1
    x=1 y=20 w=169 h=14
    x=1 y=35 w=159 h=1
    x=1 y=37 w=159 h=3
    x=1 y=41 w=159 h=1
    x=1 y=43 w=167 h=1
    x=1 y=45 w=167 h=3
    x=1 y=49 w=159 h=1
    x=1 y=51 w=159 h=1
    x=1 y=53 w=169 h=3
    x=1 y=57 w=169 h=1
    x=0 y=59 w=170 h=4
```

Figure 2. Asterix, scheduler view

changes allow us to access the content of a repository and to get information about jobs processed by a scheduler from any place, using a traditional Web browser (Figure 3).


Netscape: Directory listing
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security
Directory of /
CVS/
pdf/
demo.edoc
gs-seg.edoc
pdw-seg.edoc
test.pdf
Netscape: iiufpc58.unifr.ch scheduler status
File Edit View Go Communicator Help
Back Forward Reload Home Search Netscape Print Security
iiufpc58.unifr.ch scheduler status
running
GhostScriptSegmentation()
done
SaveDocume
paused
XMIllumInte
done
PdwSegment
done
SaveDocume
paused
XMIllumInte
done
TopDownSeg
done
TopDownSeg
done
SaveDocume
100%
Netscape: Job details

Description TopDownSegmentation(target=line, res=75, h=25, v=-1, cc=4)

State done

Log

```
Getting a JAI image of the document... done
Getting data version #1... done
Looking for some block(s) to explore... found 30
Processing block #1...
  Cropping image to (723,16 170x110)... done
  Applying horizontal RLSA... done
  Finding connected components... done
  Adding 15 lines:
    x=8 y=0 w=150 h=9
    x=8 y=11 w=150 h=1
    x=1 y=20 w=169 h=14
    x=1 y=35 w=159 h=1
    x=1 y=37 w=159 h=3
    x=1 y=41 w=159 h=1
    x=1 y=43 w=167 h=1
    x=1 y=45 w=167 h=3
    x=1 y=49 w=159 h=1
    x=1 y=51 w=159 h=1
    x=1 y=53 w=169 h=3
    x=1 y=57 w=169 h=1
    x=0 y=59 w=170 h=4
    x=3 y=69 w=166 h=36
```

Figure 3. Browser-based interfaces

3.4. Toward new interactors with XMIllum

Interactors are essential to the success of our cooperative document understanding framework. As we mentioned already, determining the right presentation and interaction techniques to use is a difficult task. This task is made even more complex by the diversity of data structures used by document understanding algorithms. Although our operators and interactors use a common XML syntax for describing blocks, lines, etc., other softwares usually have their own syntax which can be based on XML, as in TrueViz [6], or on another language such as DAFS [9].

XMIllum [5] was designed to address these problems. It combines XSLT stylesheets and Java classes to create interactive presentations of document understanding results described in any XML-based syntax. XSLT stylesheets are used to transform these results into an intermediary format that describes the graphical elements to display and the editing operations that can be applied to them. The following fragment, for example:

```
<block x="36" y="184" w="306" h="59">
 <line x="36" y="184" w="306" h="59">
  <sign x="36" y="186" w="35" h="57"/>
  <sign x="76" y="202" w="38" h="41"/>
 </line>
</block>
```

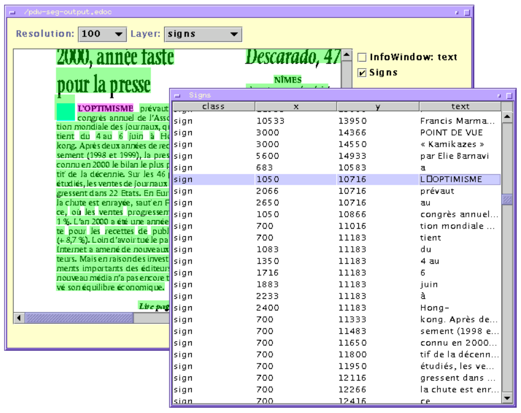
might be transformed into something like:

```
<object name="basic-block" class="visualize.Rectangle">
 <param name="color" value="yellow"/>
</object>

<object name="active-line" class="visualize.Rectangle">
 <param name="color" value="red"/>
 <param name="onClick" value="edit.Attributes"/>
</object>

<basic-block x="36" y="184" w="306" h="59">
 <active-line x="36" y="184" w="306" h="59"/>
</basic-block>
```

In this example, two types of graphical objects are defined, `basic-block` and `active-line`, that use the same Java class for rendering (`Rectangle`). A second class is associated to `active-lines` so that clicking on them will open an attribute editor. During the XSL transformation, `blocks` were replaced by `basic-blocks`, `lines` by `active-lines`, and the `signs` were simply filtered out.

XSLT supports transformations much more complex than the ones presented in the previous example. By creating the appropriate stylesheet and possibly a few Java classes for rendering and edition, one can easily adapt XMIllum to a wide variety of input formats and graphical presentation and interaction techniques. TrueViz files, for example, can be viewed with XMIllum using a very simple stylesheet. Multiple synchronized views can also be easily implemented. Figure 4 shows two such views: a graphical one showing the document and some segmented regions, and a list of these regions that allows the modification of their attributes.

We have implemented an Edelweiss interactor based on XMIllum. Although we might still develop some other interactors in the future, we intend to create a collection of stylesheets, rendering and editing classes for this particular



Resolution: 100 Layer: signs

☐ InfoWindow: text
☑ Signs

class	x	y	text
sign	10533	13950	Francis Marma...
sign	3000	14366	POINT DE VUE
sign	3000	14550	« Kamikazes »
sign	5600	14933	par Elie Barnavi
sign	683	10583	a
sign	1050	10716	L□OPTIMISME
sign	2066	10716	prévaut
sign	2650	10716	au
sign	1050	10866	congrès annuel...
sign	700	11016	tion mondiale ...
sign	700	11183	tient
sign	1083	11183	du
sign	1350	11183	4 au
sign	1716	11183	6
sign	1883	11183	juin
sign	2233	11183	à
sign	2400	11183	Hong-
sign	700	11333	kong. Après de...
sign	700	11483	sement (1998 e...
sign	700	11650	connu en 2000...
sign	700	11800	tif de la décenn...
sign	700	11950	étudiés, les ve...
sign	700	12116	gressent dans ...
sign	700	12266	la chute est enr...
sign	700	12416	ce

Figure 4. The XMIllum interactor

one that will bring the modularity of the XSLT approach to the cooperation between the users and the system.

4. Conclusion and future work

In this paper, we have presented a new approach to document understanding. We have shown how a generic and modular software platform would benefit to the community. We have shown the importance of the notion of *cooperation* for the design of such platform, and how Web technologies can be used to support it. Finally, we have presented Edelweiss, a framework we are currently developing based on this analysis.

The current implementation of Edelweiss already shows that our approach is sound and that the technologies we use are adequate. It provides us with a unifying platform that we intend to use for developing new document understanding techniques in the future, these techniques being integrated as new operators and interactors.

In addition to new interactors based on XMIllum, short-term developments of Edelweiss include an interactive eDocument builder for specifying a physical source and a job sequence to apply, as well as font identification and logical structure analysis operators. Long-term developments might include WebDAV-based repositories and user-management services.

Acknowledgements

This work is supported by the Swiss National Science Foundation (ref. 20-59354.99 and 21-61754.00) and ERCIM, the European Research Consortium for Informatics and Mathematics.

References

[1] F. Bapst. *Reconnaissance de documents assistée : architecture logicielle et intégration de savoir-faire*. PhD thesis, Université de Fribourg (Switzerland), Oct. 1998.

[2] J. Clark. XSL Transformations (XSLT) Version 1.0. W3C Recommendation, Nov. 1999. http://www.w3.org/TR/xslt/.

[3] C. Cracknell and A. Downton. A handwriting understanding environment (hue) for rapid prototyping in handwriting and document analysis research. In *Proc. of ICDAR99, the 5th International Conference on Document Analysis and Recognition*, pages 362–365. IEEE Computer Society Press, Sept. 1999.

[4] O. Hitz and R. Ingold. Visualization of Document Recognition Results using XML Technology. In *Proc. of CIDE 2000, 3ème Colloque International sur le Document Electronique*, pages 207–215, July 2000.

[5] O. Hitz, L. Robadey, and R. Ingold. An Architecture for Editing Document Recognition Results Using XML Technology. In *Proc. of DAS2000, the 4th Internation Workshop on Document Analysis Systems*, pages 385–396, Dec. 2000.

[6] T. Kanungo, C. H. Lee, J. Czorapinski, and I. Bella. TRUEVIZ: a groundtruth/metadata editing and visualizing toolkit for OCR. In *Proc. of SPIE Conference on Document Recognition and Retrieval*, Jan. 2001.

[7] D. Norman. *Things That Make Us Smart : Defending Human Attributes in the Age of the Machine*. Perseus Press, May 1994.

[8] O.Altamura, F. Esposito, and D. Malerba. Transforming Paper Documents into XML Format with WISDOM++. *International Journal of Document Analysis and Recognition*, 3(2):175–198, 2000.

[9] RAF Technology, Inc. *DAFS Library, Programmer's Guide and Reference, Release 2.0*, 2000. http://www.dafs.org/.

[10] S. Simske. The use of XML and XML-Data to provide document understanding at the physical, logical and presentational levels. In *Proc. of the ICDAR99 workshop on Document Layout Interpretation and its Applications*, Sept. 1999.

[11] W3C Architecture Domain. *Document Object Model (DOM)*. http://www.w3.org/DOM/.

Appendix A - eDocument sample

```
<edoc>

  <datasources>
    <document src="/test.pdf"/>
  </datasources>

  <processing>

    <job class="iuf.edelweiss.operators.TopDownSegmentation"
         title="Block finder" state="done" dataversion="0">
      <param name="resolution" value="150"/>
      <param name="target" value="block"/>
      <param name="hrlsa" value="20"/>
      <param name="vrlsa" value="20"/>
      <param name="cc" value="8"/>
    </job>

    <job class="iuf.edelweiss.interactors.eDocumentViewer"
         title="Interactive viewer" state="done" dataversion="1"/>

    <job class="iuf.edelweiss.operators.ScanWorX"
         title="ScanWorX-based OCR" state="waiting">
      <param name="resolution" value="400" />
      <param name="level" value="block" />
      <param name="language" value="english" />
      <param name="server" value="iiuf00.unifr.ch" />
    </job>

    <job class="iuf.edelweiss.operators.SendMail"
        title="Mail gateway" state="waiting">
      <param name="to" value="nicolas.roussel@unifr.ch" />
      <param name="subject" value="test.pdf processed" />
    </job>

  </processing>

  <data>

    <version id="1">
      <block height="528" width="4720" x="2608" y="1328"/>
      <block height="144" width="1296" x="4328" y="2024"/>
      <block height="936" width="3120" x="3408" y="2352"/>
      <block height="2552" width="3856" x="952" y="3552"/>
      <block height="2160" width="3856" x="5152" y="3560"/>
      <block height="3128" width="3856" x="5152" y="5960"/>
      <block height="160" width="3712" x="960" y="6344"/>
      <block height="2352" width="3864" x="952" y="6736"/>
      <block height="3160" width="3856" x="952" y="9328"/>
      <block height="1760" width="3848" x="5152" y="9360"/>
      <block height="1160" width="3864" x="5152" y="11360"/>
    </version>

  </data>

</edoc>
```