Flexible Publication Workflows Using Dynamic Dispatch

Sebastian Schick, Holger Meyer, and Andreas Heuer

Database Research Group
University of Rostock
{schick,hme,heuer}@informatik.uni-rostock.de

Abstract. Publication processes within Digital Libraries are seldom supported by a workflow management system (WFMS). Publication workflows are often described within the application logic due to its data dependency — publication processes are data-driven. Though, documents and the processes should not be treated independently of each other. Rather, processes should dynamically react to changes of document structure and content.
We present an approach for flexible, data-centric publication workflows. The approach extends the control-flow perspective of a WFMS with concepts for handling process adaption at run-time, depending on a document's structure and its content.

Keywords: Digital Library; Publication Process; Flexible Workflow Modeling

1 Introduction

Nowadays, Digital Libraries (DLs) are used to manage all kind of document types, e.g. academic articles, handwritings or course materials. A multimedia document encompasses different kinds of media types, e.g. text, video and audio. These media types are associated with lots of specific operations, which have to be described and implemented in the DL system. For example, supporting content based retrieval on each media type requires a complex feature extraction and indexing process. That's why the integration of multimedia documents within DLs is a challenge.

A common solution to control publication processes is the integration of a WFMS within the DL. Usually, the publication process of a certain document type is described by a process model. This model consists of activities which define atomic processing steps within the publication process. Besides this, execution constraints, resources (e.g. authors or IT services) and workflow relevant data have to be defined within the scope of a process model. The effort to maintain these process models increases with each document type added to the document model. Thus, documents and the processes should not be treated independently of each other. Processes have to react to document changes in a flexible way, i.e. often the control flow has to be adapted.

In this paper we begin with the control flow perspective of the workflow and extend it with operations to handle dynamic process changes depending on document structure and content. We introduce a model to define relationships between a document's structure and content and dynamically adapted process structures.

The rest of the paper is organized as follows. In Section 2, we provide a motivating example together with requirements for publication process modeling and discuss related work. In section 3, we give a model for dynamic publication process specification and describe the composition of flexible process parts. The implementation of the approach using the Yawl [1] workflow engine is shown afterwards. Finally, in Section 4 we conclude and give an outlook on further activities.

2 Publication Processes and Digital Libraries

2.1 Running Example

The Publication Process describes the creation of complex structured documents in digital libraries. In DLs authors often lack support during the creation of complex multimedia documents, e.g. when they structure and assemble different media types into one logical document. These systems often assume that this work was done before. Nevertheless, some processes within the DL depend on the content type like indexing, which runs differently for media types like text, image or video. In addition, content creation should not be handled independently of its usage.

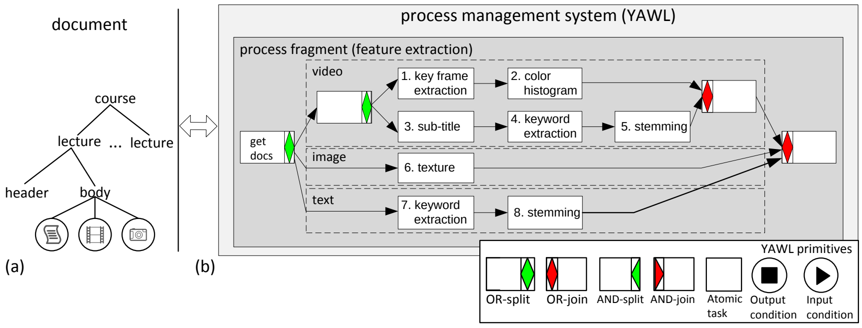

Fig. 1. Workflow for a course management

Fig. 1 (a) shows a simple document for a course management system. A course document consists of several lectures. Each individual lecture can make use of different media types, e.g. a slideshow consisting of a sequence of images, a video shown and discussed in the lecture and the full text of the lecturer's notes. In sub-figure (b) the feature extraction part of the document indexing in the DL is depicted. When the document for a specific lecture contains a video, key-frame extraction takes place first. Then methods for feature extraction for still image are applied to the sequence of key-frames, e.g. the calculation of the color histogram for each image. When sub-titles exist, they are extracted and text indexing takes place afterwards. This can be done in parallel to key-frame extraction and indexing but only if a video is present in the document.

So, some of the activities can be carried out in parallel, e.g. key-frame extraction in parallel to sub-title analysis, but others can take place only in a certain order, e.g. sub-title extraction before full text indexing. Some activities can only take place if the related media type is present in the document. There are other sub-tasks omitted here which must be executed in dependence of certain document parts. For example different metadata extractors are used for different file formats.

2.2 Requirements and Publication Process Modeling

There is a strong need for the description of both the content specific and the application-specific parts of the publication process. This is reflected by dependencies between documents or certain document parts (based on their structure and/or content) and instances of workflow tasks. As shown in the example above, different document parts need different indexing techniques, which are performed by sophisticated sub-workflow sometimes. This poses different requirements on the overall process modeling part of a DL. We list some of them which can be tackled by our approach:

Req. 1 *Avoid complex process specifications.* Moreover, the process specification should concentrate on the application-specific parts of the publication process in a comprehensive manner. **Req. 2** *Decouple the application-specific from the content specific part of the process specification.* Ideally, a high level description of the publication process concentrates on the individual, application-specific phases. **Req. 3** *Support the reuse of process parts.* **Req. 4** *Allow for run-time adaption of document specific process parts.* Each process instance should only contain process parts which are directly related to the active documents it processes. **Req. 5** *Document changes have to be associated with process parts.* Editing operations (e.g. insert, delete, . . .) on documents influence process specifications at run-time.

2.3 State of the Art

Within the Digital Library Community different approaches are discussed. Supporting authors by defining flexible interfaces is proposed in [3] and [7]. We propose the use of a WFMS that can control all processes within a digital library, because of supporting authors through flexible user interfaces does not consider technical services such as indexing of different document types.

SCOPE [6] is a framework for generic publication processes. It uses publication components as part of the publication chain which transforms documents from one state to another state. However, the described publication process refers only to publication of documents e.g. MS Word or Latex. Flexible building of collections or complex document structures is not considered.

Greenstone [2] is a Framework for building digital library collections. The architecture is divided into seven parts which represents phases within the publication process. Within these parts different plug-ins are managed to process documents in the right way. Nevertheless, the publication process is restricted to seven phases and a dynamical adaption of publication process is also not possible. Flexibility is supported only by the choice of various plug-ins.

Current WFMS offer different approaches to support flexibility. A common solution to manage different publication processes are process variants [4, 5]. Applied to DLs, process variants are produced by using a reference process model (e.g. publication process) which is configured according to different requirements, e.g. different document types. The intention is to avoid *choices* within process models where decisions can be predicted. Since the publication process is highly dynamic during runtime, the model's configuration has only a small effect. Content-related process parts must still be completely modeled (Req. 2,4,5 are not met).

Pockets of flexibility [8] uses the concept of *open instances.* Within the process model pockets of flexibility are defined within a core process. A pocket is a special build activity which composes activities depending on different constructs (e.g. fork sequence, etc.). This approach is very similar to ours regarding the choice of the late modeling concept [10]. However, the composition is left to the user and only restricted afterwards by conditions. Constraints related to external environment are not considered at all (Req. 5 is violated).

3 Flexible Document Aware Workflows

To illustrate our approach for dynamic publication processes Fig. 2 outlines the newly introduced concepts. On the left side of Fig. 2 the dynamic process model is depicted together with the related logical document model underneath. The two models will be discussed in more detail in Sec. 3.1. The dynamic parts of the approach, the process composition and execution, are shown in the center and right part of Figure 2. The details will be discussed in Sec. 3.2.

3.1 Dynamic Process Specification

The dynamic publication process model (cf. Fig. 2) will be adapted in dependency of the current document instances during runtime using a *dynamic dispatch* of activities. Therefore, our approach observes document changes and specifies where corresponding changes in the workflow should take effect, i.e. where dynamically generated sub-processes are executed. We use a notation of a core process which is extended with special *observer* and *generator* tasks. In order to achieve flexibility within the publication process the full specification of the process model is completed at runtime.

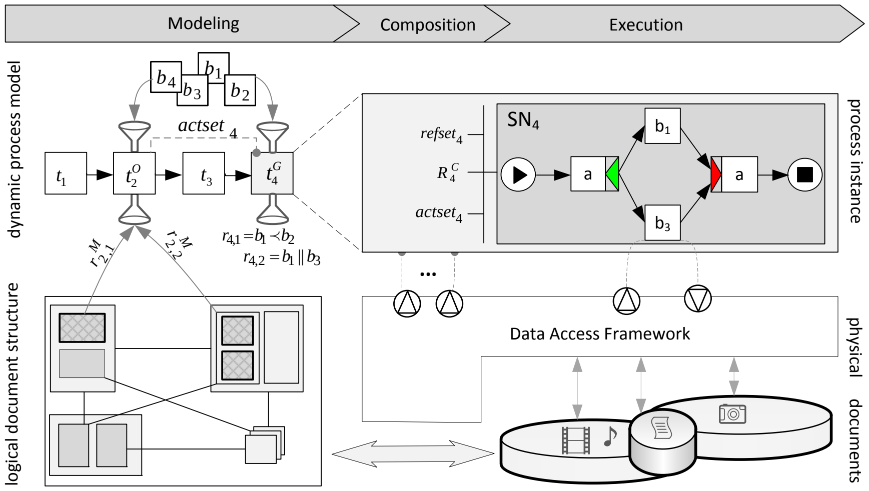

Fig. 2. Supporting Dynamic Publication Processes

Bricklets b_i are the building blocks for re-using specific activities which are bundled into sub-processes (Req. 3). They are not directly part of the process specification but will be executed at well-defined points. The bricklets are a mean for separating the application-specific process from data specific parts which rely on current document structures (Req. 2).

Definition 1 (Bricklet). *Let* $B = \{b_1, b_2, \cdots, b_i\}$ *with* $i \in N$ *be the set of* bricklets. *A bricklet* b_i *is a valid process model, which contains at least one task definition. Bricklets will be assembled into subnets* SN_i*.*

Observer tasks t_i^O define points within the dynamic process model where the status of documents to be published should be investigated. Usually, they are inserted after activities which cause major changes of the document. The task t_2^O in Fig. 2 denotes an observer task which makes use of the *matching rule set* (R_2^M). The rules ($r_{2,1}^M, r_{2,2}^M$) specify which activities should be added or removed from the process if there are certain parts in document or if they are absent.

Definition 2 (Observer Task). *Let* T^O *denote the set of* observer tasks $t_i^O = (R_i^M, match)$ *with* $i \in N$*. Then:*

- $R_i^M = \{r_{i,1}^M, r_{i,2}^M, \cdots, r_{i,j}^M\}$ *with* $j \in N$ *is a set of matching rules.*
- $r_{i,j}^M : match(pexpr_j) \mapsto (op, t_m^G, B_j, pfrag_j)$ *is a matching rule.*
- $pexpr_j$ *is used to specify data parts expected within the data. It is basically a XPath expression.*
- $op = \{add, delete, merge, undo\}$ *is the set of change operations.*
- $t_m^G \in T^G$ *is a generator task.*
- $B_j \subseteq refset_m$ *is a set of predefined bricklets.*

– *$pfrag_j$ contains the resulting XPath 1.0 node-set using $pexpr_j$.*

Whether a document fragment exists or not is determined using the path expression $pexpr_j$ and the function $match(pexpr_j)$. Using our *Data Access Framework* (DAF), each expression is evaluated directly on the documents within the appropriate repositories [9]. If the expression gets evaluated true corresponding bricklets are added to or removed from the construction set $actset_4$ of *generator tasks* t_4^G. Furthermore, the selected fragments are returned as $pfrag_j$.

Matching rules R_i^M associate parts of a document (based on content and/or structure) with a set of bricklets (activities) and define points in the control flow where the activities should be scheduled (Req. 2 & 4). As an consequence, an XPath expression will describe the parts within document instances which should (not) match and trigger activities in the subsequent workflow. Sometimes the existence or absence of a document fragment will not only add but remove also scheduled activities depending whatever the default behavior may be. If for example sub-titles are added to a video then their fulltext can be indexed and content based analysis of key-frames can be omitted.

Each $pexpr_j$ within a matching rule $r_{i,j}^M$ is closely associated with an operation *op*. Where an operation *op* may only use bricklets from set $refset_i$.

Definition 3 (Change Operation). *Let $op = \{add, delete, merge, undo\}$ denote the set of possible operations to manipulate the construction set $actset_i$ (hereinafter actset) of generator tasks t_i^G.*

The add *operation appends the set of activated bricklets B_j to the activation set actset. The* merge *operation appends a set of predefined bricklets B_j only into actset if they are not member of it. And the* delete *operation removes bricklets from actset. B_j^{-1} is the compensation of B_j and the* undo *operation appends a set of compensating bricklets B_j^{-1} to the activation set actset to rollback operations B_j after a data fragment was removed from the document.*

Generator tasks[1] t_i^G specify points within the flow of control where bricklets are combined at run-time to build up a subnet of activities, e.g. SN_4 in Fig. 2. The resulting subnet is then deployed and executed. Essentially, the generator tasks are responsible for the dynamic dispatching of the activities/bricklets like selecting and executing method calls in object-oriented systems (Req. 4 & 5). To build up the subnets a set of composition rules (R_i^C) and set of scheduled activities ($actset_4$) is used. The scheduled activities must belong to a set of allowed activities ($refset_4$) per distinct generator task. If a generator task is executed within the process, it has to compose a valid execution order for the activated bricklets. The generator tasks provide flexibility into the process instance using a late modeling, descriptive approach and under-specification [10].

Definition 4 (Generator Task). *Let T^G be the set of* generator tasks $t_i^G : (R_i^C, actset_i, refset_i) \mapsto SN_i$ *with $i \in N$. Then:*

– *SN_i is a valid subnet executed if t_i^G is processed within the control flow.*

[1] It resembles the idea of *pockets of flexibility* introduced in [8].

- $R_i^C = \{r_{i,1}^C, r_{i,2}^C, \cdots, r_{i,j}^C\}$ *with* $j \in N$ *defines a set of construction rules.*
- *$actset_i$ is a set of active bricklets (b_k) and corresponding number of data fragments ($f(b_k)$) chosen by different* observer tasks t_m^O.
- *$refset_i$ defines all bricklets allowed for t_i^G.*

Construction rules R_i^C define relationships between bricklets and how they are combined into a resulting control flow. If a bricklet is a pre-requisite for another, a sub-sequent order can be specified. Further, a bricklet can be executed sequential or parallel n-times. If not stated otherwise, bricklets can be executed arbitrarily and in parallel.

Definition 5 (Construction Rules). *Let R_i^C be the set of construction rules $r_{i,j}^C$. A construction rule $r_{i,j}^C \in \{b_k \prec b_l, b_k \overset{n}{\prec}, b_k \overset{n}{||}\}$ defines how a* bricklet b_k *is inserted into the subnet SN_i iff $b_k \in actset_i$.*

- *$b_k \prec b_l$: Iff bricklet $b_l \in actset_i$, b_l is immediately executed after b_k.*
- *$b_k \overset{n}{\prec}$: The bricklet b_k will be inserted sequentially n times.*
- *$b_k \overset{n}{||}$: The bricklet b_k will be inserted n times in parallel.*

By using these concepts, we avoid complex process structures (Req. 1). The primary process specification is a model of the application's point of view. Wherever message-specific activities have to be carried out, they are hidden by generator tasks and descriptive matching and composition rules. These rules determine the dynamic execution of a re-usable set of message-specific activities.

3.2 Composing Sub-processes

After the activation of bricklets, which is done by the observer tasks, the composition of a valid sub-process has to be controlled by the generator task (cf. center and right part of Fig. 2) . Therefore, we provide an algorithm for combining bricklets b_k into a sub-process using the construction rule set R_i^C. The composition is done during runtime to offer a flexible generation of sub-processes.

Since $actset_i$ changes during runtime, SN_i only needs to be generated when t_i^G is activated. This is a two-step procedure. First a directed acyclic graph is created with all activated bricklets $b_n \in actset_i$. In the second step we transform the graph into a valid sub-process (YAWL subnet).

Definition 6. *Let $G_i^C = (V, E)$ be a digraph. V is the set of vertices and E is the set of directed edges. The graph G_i^C contains only one starting vertex "start" $\in V$ and one ending vertex "end" $\in V$.*

Definition 7. *The indegree $deg^-(b_k)$ is the number of head endpoints for bricklet b_k. The outdegree $deg^+(b_k)$ is the number of tail endpoints for b_k.*

Listing 1 shows a simple algorithm to calculate SN_i. Since the subnet is composed during runtime no deferred choice is needed and no composition rule is mapped to OR-splits, either. We avoid cycles in the constructed subnet graph by enforcing acyclicity of the construction rule set R_i^C at modeling time[2].

Listing 1. Algorithm to calculate subnets

```
initialize $G$ with $G.V = \{start, end\} \cup actset_i$ and $G.E = \{\}$
foreach $r_{i,j}^C \in R_i^C$ {
  if $r_{i,j}^C$ equals $b_k \prec b_l$ and $\{b_k, b_l\} \subseteq actset_i$ {
    add directed edge $(b_k, b_l)$ }}
foreach $r_{i,j}^C \in R_i^C$ {
  //expand replaces $b_k$ by $n$ nodes $b_{k.m}$ of type $b_k \wedge m \in \{1, \ldots, n\}$
  if $r_{i,j}^C$ equals $b_k \overset{n}{\prec}$ and $b_k \in actset_i$ {
    $n$ = $actset_i.f(b_k)$; expand($b_k$,$n$);
    add directed edge between successive $b_{k.m}$ }
  if $r_{i,j}^C$ equals $b_k \overset{n}{||}$ and $b_k \in actset_i$ {
    $n$ = $actset_i.f(b_k)$; expand($b_k$,$n$)
    if $\exists r_{i,m}^C \in R_i^C \wedge r_{i,m}$ equals $b_k \prec b_l \wedge b_l \in actset_i$ {
      foreach $b_{k.o} \in \{b_{k.1}, \ldots, b_{k.n}\}$ { add directed edge $(b_{k.o}, b_l)$ }}
    if $\exists r_{i,m}^C \in R_i^C \wedge r_{i,m}$ equals $b_l \prec b_k \wedge b_l \in actset_i$ {
      foreach $b_{k.o} \in \{b_{k.1}, \ldots, b_{k.n}\}$ { add directed edge $(b_l, b_{k.o})$ }}}}
foreach $b_k \in G.V$ {
  if $deg^-(b_k) = 0$ { add directed edge $(start, b_k)$ }
  if $deg^+(b_k) = 0$ { add directed edge $(b_k, end)$ }}
```

The digraph for subnet SN_i is not executable within YAWL, therefore it is transformed into a valid YAWL net. The transformation is based on rules $R_{1 \ldots 6}$:

R_1 maps the *start* node to a YAWL *Input Condition* where the process starts.
R_2 maps the *end* note to an *Output Condition* where the process ends.
R_3 maps a sequential path from node b_i and b_j to corresponding tasks b_i, b_j.
R_4 maps the split of node b_k to a *And-Split task* b_k. Nodes b_i, b_j are mapped to corresponding tasks b_i, b_j. Node b_n is mapped to a *And-Join task* b_n.
R_5 if the start node *start* is part of a parallel execution, *start* is mapped to an *Input Condition* together with an *And-Split task* a. The dummy task a will be connected with the following tasks $b_i \ldots b_j$.
R_6 if the end node *end* is part of a parallel execution, *end* is mapped to a *Output Condition* together with a dummy *And-Join task* a. The dummy task a will be connected with the incoming tasks $b_i \ldots b_j$.

Example 1. In Fig. 3 (a) the process from Fig. 1 is realized using our approach. Task t_1 (*get docs*) is modeled as an observer task which controls the dynamic dispatch of activities in generator task t_2 (*feature extraction*). Task t_2 provides the set of selectable bricklets $refset_2 = \{b_1, b_2, b_3, b_4, b_5, b_6\}$, which correspond to the tasks (1. . . . 6.) in Fig. 1. The observer task t_1 uses a set of matching rules $R_1^M = \{r_{1,1}^M, r_{1,2}^M, r_{1,3}^M\}$ with $r_{1,1}^M = add(t_2, b_6,$"`//course[../doctype='image' ]`"), $r_{1,2}^M = add(t_2, \{b_1, b_2, b_3, b_4, b_5\},$"`//course[../doctype='video']`") and

[2] This can be done by applying R_i^C on $refset_i$

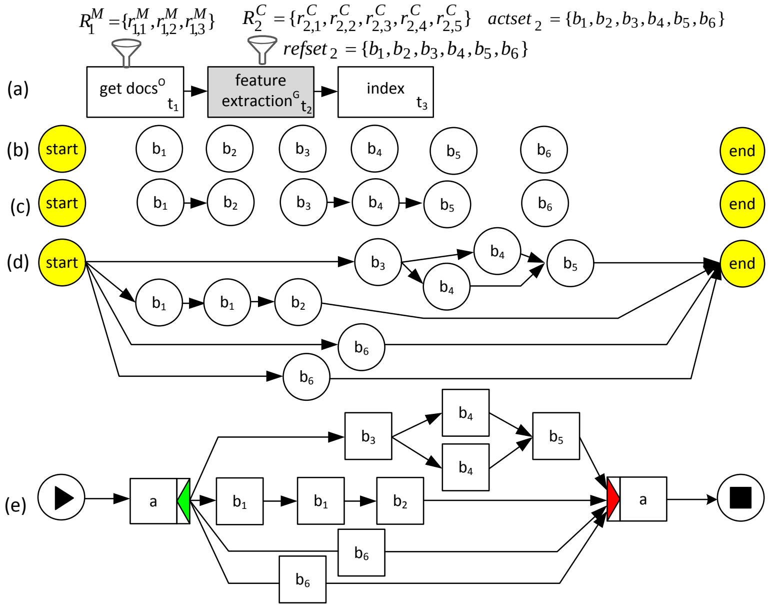
$R_1^M = \{r_{1,1}^M, r_{1,2}^M, r_{1,3}^M\}$
$R_2^C = \{r_{2,1}^C, r_{2,2}^C, r_{2,3}^C, r_{2,4}^C, r_{2,5}^C\}$
$actset_2 = \{b_1, b_2, b_3, b_4, b_5, b_6\}$
$refset_2 = \{b_1, b_2, b_3, b_4, b_5, b_6\}$
(a)
get docs^O t_1
feature extraction^G t_2
index t_3
(b)
start
b_1
b_2
b_3
b_4
b_5
b_6
end
(c)
(d)
(e)
a

Fig. 3. Digraph construction and transformation into a Yawl subnet

$r_{1,3}^M = add(t_2, \{b_4, b_5\},$”`//course[../doctype='text']`”$)$[3]. For the sub-process construction in t_2, the rule set $R_2^C = \{b_1 \prec b_2, b_3 \prec b_4, b_4 \prec b_5, b_1 \overset{2}{\prec}, b_4 \overset{2}{\|}, b_6 \overset{2}{\|}\}$ is used, which reflects the order of the activities in Fig. 1. The observer task t_1 activates the bricklets in $actset_2 = \{b_1, b_2, b_3, b_4, b_5, b_6\}$ due to matching rules $r_{1,1}^M$, $r_{1,2}^M$, $r_{1,3}^M$.

Fig. 3 (b)–(d) depict the construction of digraph G. First, all activated bricklets $(b_1, b_2, b_3, b_4, b_5, b_6)$ will be inserted into $G.V$ (Fig. 3 (b)). The application of rules $b_1 \prec b_2$ (edge from b_1 to b_2), $b_3 \prec b_4$ (edge from b_3 to b_4) and $b_4 \prec b_5$ (edge from b_4 to b_5) is shown in (c). The application of the other rules is shown in (d): $b_1 \overset{2}{\prec}$ (edge from first b_1 to second b_1), $b_4 \overset{2}{\|}$ and $b_6 \overset{2}{\|}$ (both b_4 and b_6 are duplicated). The final step (d) connects all vertices with $deg^-(b_i) = 0$ or $deg^+(b_i) = 0$ with the ”start” and ”end” nodes. The Yawl subnet (e) results from transforming digraph G into Yawl. This subnet gets deployed and executed at runtime by generator task t_2.

The approach presented above was exemplified using Yawl [1] and the corresponding Wfms. Two *Custom Component Services* for the observer and generator task types were implemented. We have extended the Yawl editor to describe the matching and construction rule sets. This allows for modeling everything within the standard Yawl environment. The bricklets are implemented as Yawl nets which contain always a start and an end condition.

[3] Tasks for keyword extraction(4.) and stemming (5.) are reusable for tasks 7. and 8.

4 Conclusion and Future Work

The publication of multimedia documents in DLs is a field that bears much practical relevance. We provided an approach for a flexible publication process of multimedia documents in DLs. In contrast to other approaches, we achieve the flexibility by monitoring document changes and specifying at which part in the control flow corresponding changes should take effect. This is done by extending a Wfms with observer tasks to monitor the document changes. Generator tasks allow for flexible construction and execution of process instances. Our technique presented here can be described as a late modeling, descriptive approach using under-specification [10]. The main contribution of our approach is that both context conditions on external data and the resulting changes of the process instances are described within one process model. Additionally, all components of our approach are provided and executed by extensions of the Wfms Yawl. The prototype implementation finally allows for a detailed case study and evaluation.

Future work will extend the construction rule mechanism to provide more freedom in combining bricklets (process fragments) into sub-processes. The construction of sub-processes should take data dependencies between bricklets into account, e.g. based on data flow analysis. Additionally a library of bricklets is built, which offers a set of generic operations for different media types, e.g. extracting metadata, for converting file formats, thumbnail generation, and others.

References

1. van der Aalst, W.M.P., ter Hofstede, A.H.M.: Yawl: yet another workflow language. Information Systems 30(4), 245–275 (2005)
2. Buchanan, G., Bainbridge, D., Don, K.J., Witten, I.H.: A new framework for building digital library collections. In: JCDL. pp. 23–31 (2005)
3. Davis, S., II, P.L.B., Cifuentes, L., Francisco-Revilla, L., Furuta, R., Hubbard, T., Karadkar, U., Pogue, D., III, F.M.S.: Template-based authoring of educational artifacts. In: JCDL. pp. 242–243 (2006)
4. Gottschalk, F., van der Aalst, W.M.P., Jansen-Vullers, M.H., Rosa, M.L.: Configurable workflow models. Int. J. Cooperative Inf. Syst. 17(2), 177–221 (2008)
5. Hallerbach, A., Bauer, T., Reichert, M.: Managing process variants in the process life cycle. In: ICEIS (3-2). pp. 154–161 (2008)
6. Müller, U., Klatt, M.: Scope — a generic framework for xml based publishing processes. In: ECDL. pp. 104–115 (2005)
7. Park, Y., Karadkar, U., Furuta, R.: Component-based authoring of complex, petri net-based digital library infrastructure. In: ECDL. pp. 22–29 (2010)
8. Sadiq, S.W., Orlowska, M.E., Sadiq, W.: Specification and validation of process constraints for flexible workflows. Inf. Syst. 30(5), 349–378 (2005)
9. Schick, S., Meyer, H., Heuer, A.: Enhancing workflow data interaction patterns by a transaction model. In: ADBIS Research Communications (2011), (Accepted for Publication)
10. Weber, B., Sadiq, S., Reichert, M.: Beyond rigidity — dynamic process lifecycle support. Computer Science — Research and Development 23, 47–65 (2009)