ProcessTron: Efficient Semi-Automated Markup Generation for Scientific Documents

Guido Sautter
KIT
Am Fasanengarten 5
76128 Karlsruhe
guido.sautter@kit.edu

Klemens Böhm
KIT
Am Fasanengarten 5
76128 Karlsruhe
klemens.boehm@kit.edu

Conny Kühne
KIT
Am Fasanengarten 5
76128 Karlsruhe
conny.kuehne@kit.edu

Tobias Mathäß
KIT
Am Fasanengarten 5
76128 Karlsruhe

ABSTRACT

Digitizing legacy documents and marking them up with XML is important for many scientific domains. However, creating comprehensive semantic markup of high quality is challenging. Respective processes consist of many steps, with automated markup generation and intermediate manual correction. These corrections are extremely laborious. To reduce this effort, this paper makes two contributions: First, it proposes ProcessTron, a lightweight markup-process-control mechanism. ProcessTron assists users in two ways: It ensures that the steps are executed in the appropriate order, and it points the user to possible errors during manual correction. Second, ProcessTron has been deployed in real-world projects, and this paper reports on our experiences. A core observation is that ProcessTron more than halves the time users need to mark up a document. Results from laboratory experiments, which we have conducted as well, confirm this finding.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing – *Linguistic processing*.

General Terms

Algorithms, Experimentation, Human Factors, Languages.

Keywords

Semantic XML Markup, Data-driven Markup Process Control.

1. INTRODUCTION

Digitizing legacy documents and marking them up according to an XML schema is an important task in the sciences. One objective is to facilitate machine processing, e.g., analyses like information retrieval and data mining. Creating comprehensive markup of high quality, covering both document structure, e.g., sections and paragraphs, and semantically important details, e.g., named entities like names of geographic locations, is challenging. Respective processes consist of many subsequent steps that build on the results of each other.

Example 1. Consider a markup process where a first step P corrects paragraph boundaries or marks up missing paragraphs in OCR output. A second step S marks up sections, as an overlay of the paragraphs. It exploits that section boundaries always coincide with paragraph boundaries. ■

In Example 1, erroneous paragraph boundaries induce erroneous section boundaries, causing additional correction effort. To prevent such error propagation, each step consists of two phases: In the **Automated Phase**, a tool, mostly NLP-based, generates markup. Since NLP tools typically do not yield error-free results – their accuracy is rarely above 95% [8] – the user checks and corrects the auto-generated markup in the **Correction Phase**.

A study accompanying a real-world markup project has confirmed that interleaving automated markup generation and manual corrections reduces the correction effort of the user significantly [11]. Further, the study makes two important observations: (1) Users should run the steps of the process strictly in the intended order to keep the error rate low. (2) The correction phases are tedious and time-consuming, as users spend a lot of time seeking and correcting errors. – Thus, curbing the user effort requires a control mechanism that provides assistance in two ways: (a) To guide users through the markup process step by step, enforcing their appropriate order, and (b) to highlight possible errors, in order to reduce the effort of finding them.

To control complex processes, one usually relies on Workflow Management Systems. The steps of a markup process would be activities of a workflow. In this current context however, it solely depends on the state of the document which step is next, i.e., on the markup created so far and on the errors in this markup. It does not directly depend on the steps that have been executed last, as we will explain. In contrast to workflows, there are no well-defined transitions between the steps. In particular, free text editing can change the state of a document arbitrarily: For instance, users might simply undo the results of previous steps, which then have to be re-executed. This situation would be impossible to model as a workflow in practice.

From the perspective of XML validation, a markup process is complete if the document passes a process-specific validation. With this point of view, a step in a markup process fixes a specific type of error. Thus, it is promising to use XML schema languages to specify the desired outcome of the various steps of a markup process. However, **XML Schemas**, which are essentially grammars, validate a document strictly in a top-down fashion, starting with the root element. Thus it is difficult to impossible to describe the intended outcomes of intermediate steps with the same schema: In Example 1, an XML Schema for Step P would have to allow paragraphs to be top-level elements. This rules out enforcing that paragraphs are nested in sections, as would be the case after Step S. **Schematron** [12] in contrast, another XML schema language, uses XPath-based **rules** to validate individual markup elements. A Schematron schema is a collection of such rules. This point-check approach is well suited to spot specific errors: A Schematron schema for Example 1 can specify one rule to make sure that the paragraph boundaries are correct (Step P) and a second one to enforce paragraphs to be nested in sections (Step S). We have found that Schematron rules are well suited to describe markup processes. However, a problem that XML Schema and Schematron have in common is their execution models: Validation tools report errors as they encounter them, not in the order a markup process intends to fix them. Thus, existing tools are not well suited to control the step-by-step execution of a markup process.

To assist users in performing complex markup processes, we propose ProcessTron. In essence, the idea is to describe a markup process by means of a Schematron schema, with some minor extensions. The execution models of Schematron and ProcessTron differ significantly, however: The ProcessTron execution model applies the rules in sequential order, in line with the order of the markup steps.

Contributions. This paper motivates and presents the ProcessTron mechanism. It assesses its benefit both in a controlled laboratory experiment and in the context of a real-word project that has created markup for over 600 document pages. In particular, we have found that (1) ProcessTron is suitable to model and control complex real-world markup processes, and that (2) working with ProcessTron more than halves the time it takes users to mark up documents. The laboratory experiment validates this second finding with a statistical significance beyond 90%. Further we report on our experiences from modeling markup processes with ProcessTron and formulate guidelines that help with modeling.

Paper Outline. The rest of this paper is organized as follows: Section 2 presents the TaxonX Process, which will be a running example. Section 3 discusses related work. Section 4 explains Schematron, the basis of ProcessTron. In Section 5, we present the ProcessTron mechanism. Section 6 features our evaluations, Section 7 concludes.

2. THE TAXONX PROCESS

TaxonX [3] is a dedicated XML Schema for biosystematics documents. We refer to the markup process that generates this markup as the TaxonX Process. In this subsection, we describe this process. It is representative of complex markup processes for scientific documents: It includes OCR cleanup at the structural and word level, markup of the logical document structure, and markup of semantic details. ([11] describes the individual steps of the TaxonX Process in more detail, discusses their dependencies, and explains why they have the specific order listed below.)

The goal of the TaxonX Process is to transform raw, XHTML formatted page-by-page OCR output into XML documents with machine-interpretable semantics. Such a document has the following characteristics: (1) The document is free from artifacts that do not belong to its actual content, e.g., page titles, and consists of flowing text, without line breaks or hyphenation. (2) Named entities are marked up and carry representations that are unambiguous, i.e., their interpretation does not require the surrounding document text. In particular, these named entities are taxon names (scientific names of life forms, e.g., *Drosophila melanogaster*) and locations. The former have Life-Science Identifiers (LSIDs, [1]) as their unique representation, the latter geographical coordinates. (3) The document has markup representing its structure above paragraph level, with an emphasis on so-called treatments, a specific kind of section. A treatment is the part of a document that refers to a specific taxon. – The process now is as follows. (Note that. for brevity, the following description subsumes several steps under one heading, and we do not mention all steps explicitly.)

1. **Layout-Artifact Detection.** Detect captions of figures and tables, page titles, and footnotes: Mark up page boundaries and pages, and then mark up page titles and footnotes next to the page boundaries. Finally, extract page numbers.
2. **Paragraph Correction.** Correct paragraph boundaries, which may be erroneous in OCR output.
3. **Paragraph Normalization.** Remove paragraph-internal line breaks and re-join hyphenated words. Add page-number attributes to the paragraph elements. Remove the page markup, which is not needed any longer.
4. **Structural Normalization.** Clean up layout artifacts. In particular, delete page titles and move footnotes and captions out of paragraphs they disrupt.
5. **MODS Referencing.** Import a metadata header [9] from a web service into the document.
6. **Taxon-Name Markup.** Mark up the taxon names, the most important details in the document. Further steps normalize the taxon names and import their LSIDs into the document.
7. **Treatment Markup.** Mark up the taxonomic *treatments*, the most important structural unit in biosystematics documents.
8. **Location Markup.** Mark up location names in the document text. Add their geographical longitude and latitude.
9. **Structure of Treatments.** Mark up the subsections of the treatments.

3. RELATED WORK

Workflow Management Systems (WfMS) [13, 14] control complex processes, which are, for instance, represented in BPEL [2]. However, in these systems (and their underlying process models) transitions between activities are well-defined and take place within a closed domain of states, e.g., who is next to take action in an editorial process. The activity just executed, together with its result, determines which activity is next. The state of a data item in the workflow therefore depends on the activities completed. In general, WfMS execute workflows in a process driven fashion. In a markup process in turn, the state of a document can change almost arbitrarily, as users can freely edit the document. In particular, they can corrupt or undo the results of steps already completed. Such arbitrary transitions are hard to impossible to model in description languages for workflows. Instead, it is practical to execute a markup process in a purely data driven fashion, as we will show. This is, the state of a document in a markup process solely depends on the markup in the document and on the errors in this markup.

XML Schema [16] is widely employed to validate XML documents. It defines the structure of XML documents as a context free grammar. Measuring the progress of a markup process can be seen as checking if a document is valid for a given step. XML Schema validates a document top-down, starting with its root element. This would require generating the markup in a top-down fashion as well. This is not the optimal order of markup steps, however. For instance, a convenient way to mark up sections is to generate them as an overlay of the paragraphs (see Example 1). In this case, paragraphs have to be marked up first. Consequently, using XML Schema to assess the status of a document in a markup process would require a sequence of different schemas that reflect the step-by-step creation of the markup. As Example 2 shows, XML element nesting can change almost arbitrarily in the course of a markup process.

Example 2. Consider a process marking up sections, subsections, paragraphs, and named entities (listed in top-down order for sake of clarity). Suppose that the best order (i.e., highest degree of automation) of creating this markup is as follows: paragraphs, named entities, sections, subsections. Enforcing this order would require three different XML Schemas: one allowing paragraphs as top level elements, a second one allowing paragraphs solely as children of sections, and finally one that enforces paragraphs to be nested in subsections that are, in turn, nested in sections. In addition, all these XML Schemas would have to contain the named entities, which are marked up before the sections. ■

The effort to create and maintain a series of schema definitions would be extremely high. It would also reduce flexibility by much, as adaptations would induce changes to a series of schema definitions. Altering such a series of schema definitions would be tedious and error prone.

Annotation-Control Mechanisms are software components that coordinate the generation of markup. Such components have been successfully used in corpus-annotation projects like the Penn TreeBank Project [7] or the GENIA Project [5]. However, the scope of the control mechanisms in these projects was limited to individual steps – for instance to highlight words for which annotators do not agree on the part-of-speech tag [7]. They are not intended and not suited to control an entire markup process. In contrast, they usually are special-purpose implementations for specific types of detail-level markup elements. Thus, these plug-ins do not solve the problem addressed here. Furthermore, special-purpose components tend to have a low level of genericity, so changes to the markup process would require changes on the implementation level.

4. SCHEMATRON

Schematron [12] is an XML validation approach that, instead of a grammar-style schema, uses XPath expressions [15] (called **rules**) to validate document markup. Each rule stands by itself, so it is possible to target specific parts of the markup, regardless if other parts have been created or validated. This rule-based approach forms the basis of our markup-process-control mechanism.

A Schematron schema has the following structure: It contains one or more **rules**. A rule validates a specific part of the document markup, the so-called **rule context**, expressed as an XPath expression. Each rule contains one or more **assertions** or **reports**, which perform the actual validity checks. An assertion consists of an **XPath predicate** evaluated against the context of the surrounding rule. If the predicate evaluates to false, the assertion outputs an error message. Reports in turn output an error message if the predicate evaluates to true. To support workflows, Schematron allows for grouping the rules, and one can switch groups on and off. When using this grouping feature, however, the user has to specify which step a given document is in, by means of a parameter to the Schematron validator. Our goal in turn is to identify the current step automatically, based on the state of the document.

In principle, validation of a document against a Schematron rule works as follows: First, the validator uses the XPath query from the **context** attribute to select the elements to check, the so-called **context elements**. In Example 3, these are all **paragraph** elements. Then, the validator evaluates the XPath predicates specified in the **test** attributes of the reports and assertions, for each of the context elements. If the predicate of an assertion evaluates to **false**, the validator outputs the textual content of the **assertion** element. If the predicate of a report evaluates to **true**, the validator outputs the textual content of the **report** element.

Example 3. The following Schematron rule tests if paragraphs have proper boundaries, Step 2 in the TaxonX Process. The assertion uses a regular expression and the **matches()** function from XPath 2.0 to test whether the textual content of a **paragraph** element ends with a punctuation mark. In the same way, the report identifies any **paragraph** elements whose textual content starts with a lower case letter. Expressions evaluated and text output during validation are in bold:

```
<rule context="paragraph">
  <assert test="matches(text(), '.+[\.|\!|\?]')">
    Paragraphs must end with a sentence-ending
      punctuation mark.</assert>
  <report test="matches(text(), '[a-z].+')">
    Paragraphs must not start with a lower case word.
  </report>
</rule>
```

Note that this rule is designed to find every paragraph that might be erroneous, even at the cost of some false positives. ■

The default Schematron execution model is based on XSLT. It first compiles a Schematron schema into an XSLT stylesheet, where each rule becomes an XSL template. Then it applies the stylesheet to the document to be validated. The output is a sequence of error messages. Due to the way XSLT works, the order of these messages corresponds to the document order of (possibly) erroneous markup elements. Further, the output includes all error messages for the entire document markup. In other words, the default Schematron execution model evaluates the rules as if they were an unordered set. However, enforcing the order of the steps requires a well-defined ordering of the rules. In particular, only the error messages of the first rule that has failed are relevant. This calls for an alternative execution model.

5. PROCESSTRON

In this section, we introduce our markup-process-control mechanism. First, we show how to represent a markup process by means of a Schematron schema. Second, we define the ProcessTron execution model (PEM for short), which controls markup processes based on such schemas. As opposed to the default

Schematron execution model (see Section 4), the PEM applies the rules sequentially.

5.1 DESCRIBING MARKUP PROCESSES WITH SCHEMATRON SCHEMAS

We propose to represent each step of a markup process as a Schematron rule. To describe and control markup processes, we require for each step (1) an identifier for the respective automated markup tool and (2) immediate, ID-based access to the possibly erroneous markup elements. We therefore extend the definition of a rule in two points:

- **AMT-ID:** Each rule bears the identifier of the automated markup tool that performs the automated phase of the respective step. This facilitates automated execution of the tool.
- **Element-ID:** For each assertion/report, we require the textual message to specify the ID of each failing markup element. Using the (optional) **name** element of Schematron and its **path** attribute, this is straightforward; we only make it mandatory. This facilitates highlighting suspected errors in the correction phase of the step represented by the rule.

For clarity, we refer to a Schematron schema that provides these extensions as a **ProcessTron schema**. Example 4 is a rule representing the step which checks and corrects paragraphs boundaries.

Example 4. The following ProcessTron rule represents the step that corrects paragraph boundaries (parts specific to ProcessTron in bold). The **automatedMarkupTool** element specifies in its **id** attribute which markup tool to apply if a paragraph in a document does not comply with the rule. This is the case if the paragraph fails the test of an assertion or passes the one of a report. Both indicate that the paragraph boundaries might be erroneous. The **path** attributes of the **name** elements in the assertions and reports include the IDs of the affected paragraphs in the error messages. The actual test is exactly the same as in Example 3; the difference is that the ProcessTron specific parts (in bold) have been added:

```
<rule context="paragraph" id="1">
  <automatedMarkupTool
   id="#paragraphBoundaryCorrector"/>
  <assert test="matches(text(), '.+[\.|\!|\?]')">
   <name path="@id"/>: Paragraphs must end with a
    sentence-ending punctuation mark. </assert>
  <report test="matches(text(), '[a-z].+')">
   <name path="@id"/>: Paragraphs must not start
    with a lower case word.</report>
</rule> ■
```

The **design of the XPath tests** requires special attention: To reliably highlight all potential errors in the correction phase, the XPath tests have to be designed for 100% recall, i.e., to make sure that every possible error is indeed highlighted. A certain number of false positives are acceptable, i.e., markup elements that are actually correct, but fail a given XPath test: A user can quickly recognize that they are not erroneous and mark them as correct. Example 5 illustrates this.

Example 5. The XPath test of the assertion in Example 3, for instance, would recognize every section heading in this paper as a potential error – because headings do not end with a punctuation mark. This is necessary, however, in order to not miss any paragraphs that do have erroneous boundaries. However, marking the relatively few section headings as correct is little effort for the user, compared to checking all paragraph boundaries in a document. ■

5.2 PROCESSTRON EXECUTION MODEL

While Schematron schemas require only marginal extensions to describe markup processes, the picture is different for the execution model. The XSLT-based one of Schematron is not well suited to control a markup process. In particular, its output does not reflect the order of the rules, but the document order of the markup elements the error messages refer to. However, the order of the rules reflects the order of the steps of the markup process. Thus, it is essential to enforce this order. Consequently, we introduce a new execution model for ProcessTron schemas, the **ProcessTron Execution Model (PEM)**.

```
01 // functions for evaluating rules on a document
02 boolean fails(Test T, Document D) :=
03   true if D contains any markup elements that do not match
       (for assertions) or match (for reports) the XPath test of T,
04   false otherwise
05 boolean fails(Rule Φ, Document D)
06   for (Test T in Φ) // apply individual tests
                         (assertions & reports) of Φ
07     if (fails(T, D) // D fails T, and thus Φ
08       return true
09   return false // D did not fail any test,
                     thus does not fail Φ
10 // functions for performing individual
      steps in a markup process
11 void executeAutomatedPhase(Rule Φ, Document D) :=
12   apply the automated markup tool for
      the step represented by Φ
13 void executeCorrectionPhase(Rule Φ, Document D) :=
14   display D for manual correction, using Φ to
     highlight potential errors
15 void executeStep(Rule Φ, Document D) // execute the
                                    step represented by Φ
16   executeAutomatedPhase(Φ, D)
17   while (fails(Φ, D)) // stay in correction phase
                             until D passes Φ
18     executeCorrectionPhase(Φ, D)
19 // main rule evaluation functions
20 Rule getCurrentStep(PT-Schema P, Document D)
21   for (Rule Φ in P) // treats schema as
                          ordered sequence of rules
22     if (fails(Φ, D)) // D fails Φ
23       return Φ
24   return nil // no failing rule found
25 // main function
26 void executeProcess(PT-Schema P, Document D)
27   while (true)
28     Rule Φ = getCurrentStep(P, D) // find current step
29     if (Φ == nil) // D did not fail any rule
                       ➔ markup process complete for D
30       return
31     else executeStep(Φ, D) // execute step
                                 represented by Φ
```

Figure 1. The ProcessTron execution model

Figure 1 visualizes PEM as pseudo code. PEM applies the individual rules sequentially, one by one (Line 21). Keep in mind that the order of the rules reflects the order of the steps in the markup process described by the ProcessTron schema. The loop goes through the rules in this order. As soon as a rule Φ fails (Line 22), i.e., it reports potentially erroneous markup elements, rule application stops. Φ corresponds to the first step of the markup process that is not yet complete, i.e., the next step to execute, referred to as S in the following. PEM then executes S (Line 31): First, it applies the automated markup tool that belongs to S (Line 16). Then execution remains in the correction phase of S until the user has handled all potential errors reported by Φ (Lines 17 and 18). The user has two ways of doing so: (1) He can correct the error. (2) He can approve the markup element in question, i.e., stating that it is not an error. – To curb the user effort, our implementation of PEM (described in Section 6.1) uses the XPath tests of Φ to highlight all markup elements in doubt. When S is complete, i.e., Φ reports no more errors, execution starts again by applying the first rule in the markup-process definition (Line 28). This is necessary because a user might have introduced new errors in the correction phase. When no rule reports an error any more, the markup process is complete.

Note that PEM is lightweight and easy to implement in common XML editors. It only requires an IO facility to read the process definition and an XPath engine to evaluate the rules.

6. EXPERIENCES FROM EXPERIMENTS AND REAL-WORLD DEPLOYMENT

In this section, we first report on a controlled laboratory experiment we have conducted with ProcessTron. With a statistical significance of over 90%, it shows that working with ProcessTron yields a speedup of over 50%. Second, we report on the experiences we have gained with ProcessTron in a real-world markup project, the ZooTaxa Project. This project has used the TaxonX Process (Section 2) to generate TaxonX markup for all ant-related documents from the ZooTaxa[1] collection, i.e., 30 documents with over 600 pages in total. – Our core observation, which is fully in line with the laboratory experiment, is that ProcessTron significantly curbs user effort: It more than halves the time a user must work on a document page. We further report on the insights we have gained when modeling the markup process of the ZooTaxa Project with ProcessTron. The main finding is that ProcessTron is well suited to model markup processes. Further, based on our observations, we propose some general process-modeling guidelines, which we deem helpful when designing and modeling a given markup process. – The focus of our evaluation lies on the laboratory experiment and the ZooTaxa Project rather than on the process modeling itself. This is because, in any markup project, there is only one markup process to model. This happens at a central instance. On the other hand, we envision many users working with ProcessTron on many documents. Thus, reducing the user effort is far more important; it outweighs the effort for modeling the markup process by much.

6.1 EXPERIMENTAL SETUP

Software. As the platform for our experiments, we have used the GoldenGATE Editor [10][2]. Its purpose is to assist users creating and correcting semantic markup. It supports the separation of each step into an automated phase and a correction phase. To facilitate deployment of automated markup tools, it provides interfaces for their integration. To simplify correction for users who are not XML experts, manual editing works on the element level in GoldenGATE, as opposed to the character level in other XML editors. This means, for instance, that users do not have to bother with escaping the values of attributes, e.g., replacing '<' with '&lt;', as this happens automatically. Likewise, creating, removing, and renaming XML elements are atomic operations, as opposed to editing XML tags at the character level. In addition, GoldenGATE offers specialized document views that let the user sift through specific markup elements very quickly, e.g., a list view for location names. For our experiments, we have implemented ProcessTron as a plug-in for the GoldenGATE editor. Besides the process-control mechanism, the plug-in provides editing facilities for ProcessTron schemas: an editor for individual rules, with a test function for the XPath expressions, and a selector for the automated markup tool assigned to the step the rule corresponds to. On execution, the ProcessTron plug-in uses a specialized list view to display potentially erroneous markup elements. The rationale is that the user does not have to inspect the whole document.

Measures Used. In both the laboratory experiment and the real-world markup project, we use the following measures to quantify user effort: d is the number of documents, t_i the time it took to mark up document i, subsequently referred to as working time for the document, and p_i the number of pages in document i. Given this, the measures are as follows:

1. The **average working time per page** (AWT) is based on the working times for the individual documents, regardless of document size:
$$AWT := \frac{1}{d}\sum_{i=1}^{d}\frac{t_i}{p_i}$$
2. The **weighted average working time per page** (WAWT) weights the times for each document relative to the document size, i.e., the overall average working time per document page:
$$WAWT := \sum_{i=1}^{d} t_i \Big/ \sum_{i=1}^{d} p_i$$

Measurement. In all experiments and studies, we have measured the time it took users to complete the markup of a document, starting with the OCR output. From these numbers, we have then computed AWT and WAWT. In the laboratory experiment, we consider the markup of a document to be complete if it matches that of a reference document. In all our markup efforts, the markup of a document is complete if the TaxonX Process is complete, i.e., the document is properly marked up and valid according to the TaxonX schema.

6.2 LABORATORY EXPERIMENT

To assess the benefit of ProcessTron under controlled conditions, we have conducted a laboratory experiment, with 8 participants. From preliminary experiments, we knew that we could expect a speedup of around 2. According to [4], with 8 participants and a speedup of 2 we can expect a statistical significance below 10% in a one-sided t-test, with about 80% power [4]. – We have not used biosystematics documents for this experiment. This is because

[1] ZooTaxa (http://www.mapress.com/zootaxa/) is a biology journal.

[2] Software and documentation available for download at http://idaho.ipd.uka.de/GoldenGATE

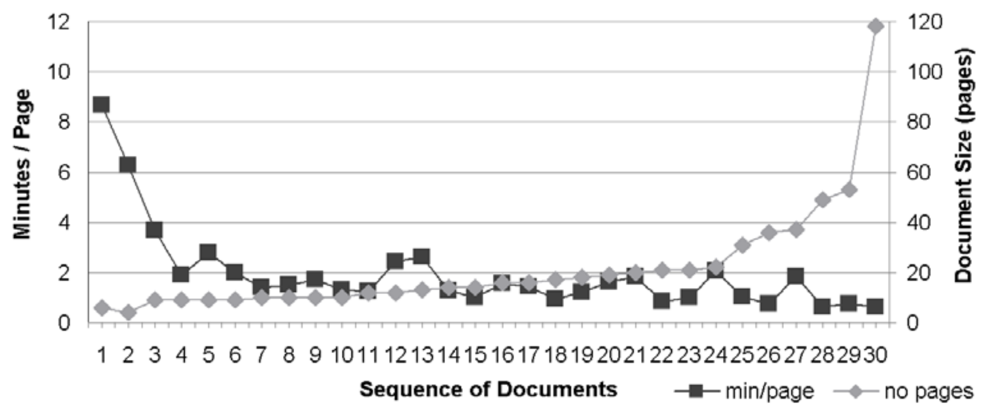

Figure 2. Document size in pages & working time per page in minutes

marking them up requires domain knowledge. When recruiting participants for the experiment, we did not want to be restricted in this way. We have used documents that are (a) commonly understandable and (b) comparable to biosystematics documents in structure and markup complexity. The documents used here are recipes for pasta dishes. The markup covers both details (ingredients and cooking tools) and document structure (recipes, and within these, titles, ingredient lists, and step-by-step preparation instructions).

We have defined and modeled a markup process for cooking recipes, consisting of the following steps:

1. **Layout-Artifact Detection**, same as in Section 2.
2. **Paragraph Correction**, same as in Section 2.
3. **Paragraph Normalization**, same as in Section 2.
4. **Structural Normalization**, same as in Section 2.
5. **Ingredient Markup.** Mark up the ingredients. This helps to identify recipe titles and ingredient lists in subsequent steps.
6. **Recipe Markup.** Mark up individual recipes.
7. **Cooking Tool Markup.** Mark up cooking tools. This helps to distinguish between 'ingredient list' and 'preparation' in the next step.
8. **Structure of Recipes.** Mark up the subsections of the recipes, namely title, ingredient list and preparation, plus (if present) background information, advanced tips and recipe variations.

This means that we could re-use the rules for almost all the normalization steps fom the TaxonX Process model, which represent about half of the process, namely Steps 1 through 4. The only part we had to adjust and partly model anew was the detail and structural markup that follows thereafter. For instance, we had to adjust the configuration of the tool that marks up the inner structure of treatments to mark up the inner structure of recipes, which involves other sub section types, and other categorization rules. – The 8 participants in this experiment received a brief training with the GoldenGATE Editor and ProcessTron, neither of which they had used before.

Table 1 displays the average time it took the participants to mark up a document with and without the support of ProcessTron. We have obtained the baseline numbers in a previous experiment with GoldenGATE [10] where users marked up the same documents, but without the support of ProcessTron. The numbers prove that ProcessTron yields a considerable speedup. In particular, the time it took the participating users to mark up a document with ProcessTron is less than half the time it took without. The post-hoc statistical significance of this result is below 1% in the paired t-test, at over 90% power. This result by far exceeds the expected strength, emphasizing the usefulness of ProcessTron. The quality of the resulting markup was equally high in both cases; we verified this through comparison with reference documents.

Table 1. Results of laboratory experiment

	ProcessTron	Baseline
Average working time in minutes (minutes/page)	29.50 (2.46)	79.1 (6.59)
Standard Deviation	14.22	18.30
Speedup (over baseline)	62.71%	N/A

6.3 THE ZOOTAXA PROJECT

After the favorable laboratory experiment, we have successfully deployed ProcessTron in the ZooTaxa Project. This project was a real-world markup project in the biosystematics domain. Using the GoldenGATE Editor and ProcessTron, a biologist has created TaxonX markup for all ant-related documents from the ZooTaxa collection, i.e., 30 documents with over 600 pages in total. We have measured how the average working time per page has evolved during the project.

We compare the ZooTaxa Project to another markup project that took place in the biosystematics domain as well, the so-called Madagascar Project [11]. Using the GoldenGATE Editor, the Madagascar Project has generated TaxonX markup for the complete literature on the ant fauna of Madagascar, comprising over 100 documents with a total of over 2,500 pages. Note that the sets of documents marked up in the two projects are mutually disjoint. The only difference between the two projects, apart from the fact that the documents are different, is that ProcessTron was used in the ZooTaxa Project but not in the Madagascar Project. The biologist who participated in the ZooTaxa Project had participated in the Madagascar Project before. Thus, she was proficient with the TaxonX Process and the GoldenGATE Editor before the start of the ZooTaxa Project, and we can rule out any learning effects in this respect. Thus, using or not using ProcessTron is the only variable. Its effect is easy to measure. Our measurements from the

Madagascar Project are well suited to serve as the reference point for the ZooTaxa Project. It took users about 3-5 minutes to mark up a document page in the Madagascar Project. This number will be our reference point.

Figure 2 graphs the evolvement of the working time per document page over all documents in the ZooTaxa Project. The graph implies that it took the user only the first 3 documents to get used to working with ProcessTron, as the working time per page declines significantly with these initial documents. From Document 4 on, the working time levels off around 2 minutes per page. It keeps slightly decreasing over the remaining documents, towards around 1 minute per page. We attribute the oscillations in the graph to the peculiarities of the individual documents, requiring more or less manual corrections.

Table 2 shows the average and weighted average working time per page. Note that the number of documents d = 30 in the ZooTaxa Project. Due to the familiarization phase that spans the initial 3 documents, we also give both averages without these documents, labeled '**after familiarization**', with i starting at 4 instead of 1 in the above formulas. The numbers clearly show the benefit of ProcessTron: The working time per page is slightly more than 1 minute. Compared to the 3-5 minutes per page measured during the Madagascar Project without ProcessTron, this represents a speedup of around 2.5, even higher than in the laboratory experiment.

Table 2. Measurements from ZooTaxa Project

Measured	**Working Time** (minutes / page)
Average working time per page over all documents	1:56
Average working time per page after familiarization	1:19
Weighted average working time per page over all documents	1:27
Weighted average working time per page after familiarization	1:11

6.4 PROCESS-MODELING EXPERIENCES

To deploy ProcessTron in the ZooTaxa Project and elsewhere, we have modeled the TaxonX Process in ProcessTron[3]. The process consists of over 20 steps, and the resulting ProcessTron schema contains the same number of rules. We encountered some interesting issues, to be discussed below, but no major difficulties. The individual rules, and the tests (report or assertion) in particular, are not very complex: Most of the rules require only one test. Only few rules – mostly those with regular expression patterns – are easier to model with two or three tests. This is because otherwise the regular expressions become highly complex.

[3] The ProcessTron markup process definition is available from http://idaho.ipd.uka.de/ProcessTron/TaxonX-Process.xml. We deem the rules for steps Normalization.Paragraphs.ParagraphBoundaries and CollectionData.MaterialsCitations.MarkUpGeoCoordinates good examples for the use of regular expression patterns.

As a general result of the ZooTaxa Project, ProcessTron is well suited to model complex markup processes like the TaxonX Process. This outcome is somewhat expected, considering the high expressiveness of XPath, which serves as the basis for ProcessTron.

In our modeling effort, we have observed several interesting issues, which we now report on, namely the modeling of steps that (a) mark up details that may or may not be present in a document, (b) work with temporary markup or mark up artifacts that will be deleted later on. – Suppose a given Step S creates markup of a specific type, referred to as M in the following. An apparently straightforward approach to model S is to design an XPath test that checks whether or not markup of type M is present in a document: If markup of type M is present, S has been executed, otherwise not. However, this simple approach works in neither Case (a) nor Case (b), as we will describe in the next paragraphs. Logging which steps have been executed is not an option either: Since users can simply undo entire steps by hand, as explained earlier, ProcessTron is – and has to be – completely data driven.

(a) Checking the results of the steps that generate the markup for important details is not trivial: If no detail markup of a specific type (e.g., location) is present in the document, this can either mean that the step creating this markup is not yet complete, or that the automated markup tool intended to create this markup did not do so. The reason for the latter may well be that no such details are present in the document at all. In this case, we have two options: First, if the markup tool performing a given step leaves specific traces besides the markup it creates, we can rely on these traces to check if the step is complete, as illustrated in Example 6.

Example 6. Page numbers are removed from the document later in the markup process because they are layout artifacts. The tool that extracts the page numbers, however, does not only mark them up, but also adds them as attributes to the page and paragraph elements. Thus, the page-number attributes of the paragraphs are evidence whether or not the page-number-extraction step is complete. ■

Second, to check if a step that marks up distinctively structured parts of the text is complete, we can use regular expression patterns: The respective XPath test can check if a piece of the document text matches a specific pattern, but is not marked up accordingly. Example 7 illustrates this for geo-coordinates; we have used the same approach for dates.

Example 7. Geo-coordinates are not always given in older documents. Thus, it is not sufficient to check for the presence of respective markup elements to find out if the respective markup step is complete or not. We use a regular expression pattern to test if the document text contains parts that might be geo-coordinates, but are not marked up accordingly. ■

(b) Temporary markup is markup that is created in a specific step of a markup process, but is removed again in a later step. Its purpose is to act as a helper in the steps between its creation and removal. It is challenging to decide if temporary markup is yet to create, or if it has already been removed. As we cannot rely on logging, we have to rely either on markup created in steps related to the one that creates the temporary markup, or on evidence from the document text. Example 8 illustrates this.

Example 8. In an early step of the TaxonX Process, we temporarily mark up pages to help detecting footnotes and print artifacts. After artifact detection is complete, the page markup is removed. Thus, checking for the presence of pages alone is insufficient to tell if pages are yet to be marked up. We additionally rely on the page boundaries, which mark the border between two pages in the OCR output: The first step marks up the page borders, the second one the actual pages between these borders. Both page borders and pages are removed after the structural normalization. Exploiting this dependency, we check if the document contains both page borders **and** pages. If the former are present, but the latter are not, pages are to be marked up. If neither is present, pages have been removed. ■

Process-Modeling Guidelines. While modeling the TaxonX process, we have found that it is very helpful to study the automated markup tools that perform the individual steps: Knowing the way they work and the evidence they rely on (see Example 9), as well as the output and the errors they might generate is extremely helpful when designing the XPath tests for the rules. Furthermore, it is helpful to test the rules for the individual steps in isolation on specific example documents, as this shows possible errors early on. This is similar to unit tests [6] in software engineering. Only after these individual tests it makes sense to compose the actual ProcessTron schema. This is, to arrange the rules according to the order of the markup process steps they represent.

Example 9. The markup tools used in the different steps of a markup process may rely on different evidence. NER components, for instance, might use word structure (by means of regular expression patterns) or lexicons. Tools that create structural markup may rely on statistical models or rules referring to detail markup. As these different techniques are susceptible to different errors in the document, it is sensible to use rules whose design reflects these differences: If the tool for the automated phase of a given step uses regular expressions, for instance, it is often sensible to use regular expressions in the rule that represents this step as well. The following two instances illustrate this:

1. Think of a tool that marks up dates. Further, suppose that this tool uses regular expression patterns to recognize dates based on their distinctive syntactical structure. Then an XPath with a regular expression that tests if all text snippets with this particular structure are marked up as dates is a suitable means to test whether or not the step that marks up dates has been executed.

2. Think of a tool that marks up figure captions. Further, suppose that this tool relies on caption paragraphs to start with 'Figure X:', where X is the figure number. Then an XPath expression that tests if all paragraphs starting with the word 'Figure' followed by a number and a colon are marked up as captions is a suitable means to test whether or not this tool has been executed. ■

Summary. Modeling a markup process is not as straightforward as it might seem at first glance. To handle temporary markup, one has to carefully study the interdependence of individual markup steps. To model steps that mark up semantic details of documents, it has helped us to put much attention to the functioning of the automated markup tools. Our guidelines from this section should facilitate the deployment of ProcessTron in a wide variety of markup processes.

7. CONCLUSIONS

In this paper, we have presented ProcessTron, a lightweight mechanism for controlling semi-automated markup processes. It guides users through markup processes and assists them in correcting errors left by automated markup tools. We expect ProcessTron to be easy to implement in other existing markup environments, as it relies completely on the on-board facilities of common XML editors. Both a laboratory experiment and observations from a successful real-world deployment show that ProcessTron yields considerable benefit: It more than halves the time it takes users to mark up a document.

8. REFERENCES

[1] Brazma, A. et al. Standards for systems biology. Nature Reviews Genetics 7, pp. 593–605, 2006.

[2] Business Process Execution Language (BPEL) http://www.bpelsource.com/bpel_info/spec.html

[3] Catapano, T. et al. TaxonX: A Lightweight and Flexible XML Schema for Mark-up of Taxonomic Treatments. In Proceedings of TDWG 2006, St. Louis, MO, USA, 2006.

[4] J. Cohen, Statistical Power Analysis for the Behavioral Sciences, 2nd ed., Erlbaum, Hillsdale, NJ, USA, 1988, ISBN 0-8058-0283-5

[5] Kim, J.-D. et al. GENIA corpus – a semantically annotated corpus for bio-text-mining. Bioinformatics, pp. i180-i182, Oxford University Press, 2003.

[6] Kolawa, A.; Huizinga, D. Automated Defect Prevention: Best Practices in Software Management. Wiley-IEEE Computer Society Press, 2007

[7] Marcus, M. P. et al. A. Building a Large Annotated Corpus of English: The Penn Treebank. Computational Linguistics, Vol. 19, No. 2, pp. 313-330, 1994.

[8] Mikheev, A. et al. Named Entity Recognition without Gazetteers, in Proceedings of EACL, Bergen, Norway, 1999

[9] Metadata Object Description Schema. http://www.loc.gov/standards/mods/

[10] Sautter, G. et al. Empirical Evaluation of Semi-Automated XML Annotation of Text Documents with the GoldenGATE Editor. In Proceedings of European Conference on Research and Advances in Digital Libraries, Budapest, Hungary, 2007.

[11] Sautter, G. et al. Creating Digital Resources from Legacy Documents - an Experience Report from the Biosystematics Domain, in Proceedings of ESWC, Heraklion, Greece, 2009

[12] The Schematron Assertion Language http:// xml.ascc.net/resource/schematron/Schematron2000.html

[13] Van der Aalst, W. M. et al. Workflow Patterns, Distributed and Parallel Databases 14(1): pp. 5-51, 2003

[14] Van der Aalst, W. M., van Hee, K. Workflow Management: Models, Methods, and Systems. The MIT Press, Cambridge, Massachusetts, 2004

[15] XML Path Language. http://www.w3.org/TR/xpath

[16] XML Schema http://www.w3.org/XML/Schema