Soft validation in an editor environment

Schematron for non-technical users

Martin Middel
FontoXML
<martin.middel@fontoxml.com>

Abstract

To allow the use of Schematron[1] in a quickly changing environment like an editor, understandability is crucial. An author may not be an XML expert, so they must be guided in resolving the messages generated by Schematron.

A good understandability rests on two pillars: performance and user interface. An author needs constant feedback on the current state of the soft validation. The author must know which places in the document need attention, and how to resolve them. The report must then update as soon as possible to enable the author to see the result of a modification they just made.

To ensure good performance, a number of technical problems have been solved, this includes a novel dependency tracking system.

1. Introduction

FontoXML is an editor for XML content, used by non-technical authors. At this moment, FontoXML offers a standard editor for DITA 1.3[1] and can be configured to support any schema including TEI[2], JATS[3], Office Open XML[4] and a number of DITA specializations.

Since non-technical authors have no knowledge of XML or the schema, they will not understand nor be able to fix invalid documents. Therefore we guide the author to prevent them from creating invalid documents; we ensure loaded documents remain valid at all times. Valid in this context means both well-formed (no unclosed tags), but also schema-valid (titles can not contain list items).

However, some restrictions should not always be enforced. Constraining the length of a title will destroy usability, because typing text in it will suddenly be disabled. Additionally, schema restrictions may need to be relaxed in various use cases. For these reasons, we have implemented soft validation in the editor, which essentially are recommendations instead of requirements.

[1] http://docs.oasis-open.org/dita/dita/v1.3/dita-v1.3-part3-all-inclusive.html
[2] http://www.tei-c.org/index.xml
[3] https://jats.nlm.nih.gov/index.html
[4] http://www.ecma-international.org/publications/standards/Ecma-376.htm

2. The case for soft validation

In our experience adapting the FontoXML editor for various clients, we have seen two major cases for soft-validation:

1. Adapting content from a permissive schema used for editing to a strict schema used for publishing content.

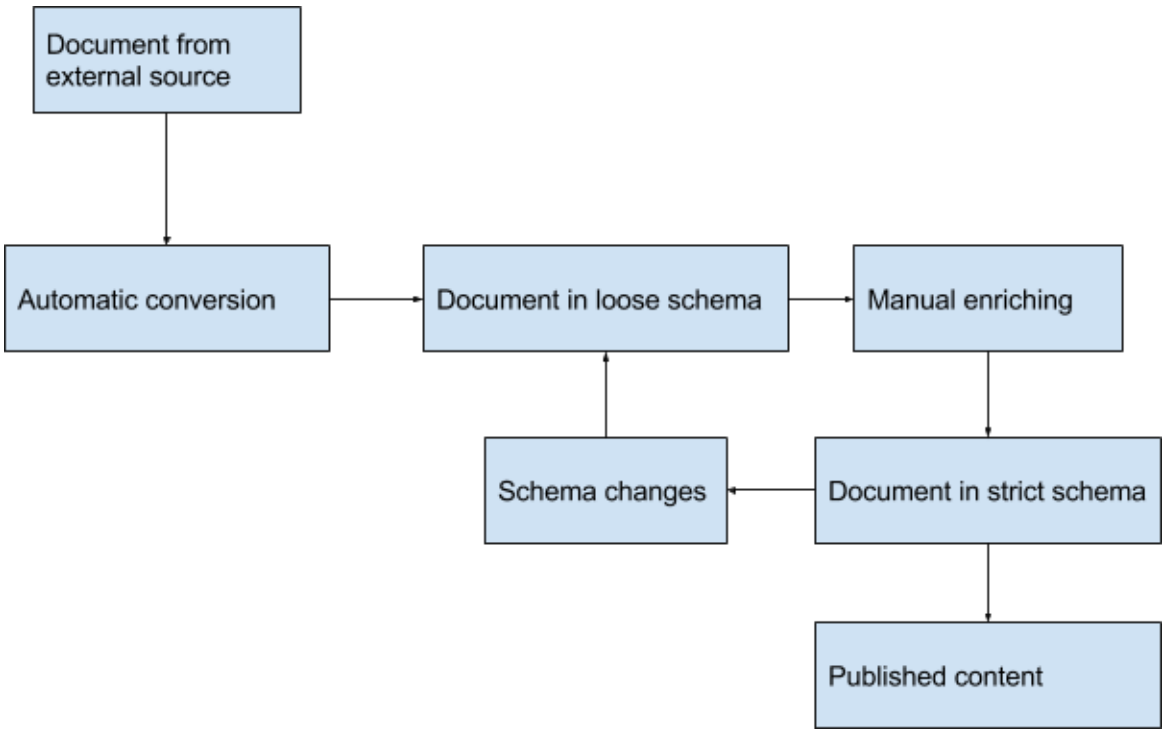
   One of our clients (automatically) imports content from a word processor. They can not automatically tag most elements, apart from distinct paragraphs, so this enrichment is a manual process. After the enriching phase of the workflow, all elements must be tagged: title paragraphs must be title paragraphs, abstract paragraphs must be abstract etc. Using the stricter schema as soft validation on top of the permissive schema provides valuable feedback during the enrichment process.

2. Schema with varying publication-specific constraints.

   Another client maintains a very high number of different schemas, each fitting their unique purpose. A schema for writing a book for teaching a language is different from an exercises book for learning math.

   To allow the same XML pipeline to be used for all of these schemas, the actual schemas are the same. Different soft-validation rules are enforced over them, in order to tailor to specific publication needs. This also allows for easy upgrades from one version of the (overlaying) schema to the next: the document is always schema-valid.

Soft-validation can also be used to guide an author into adhering to a style guide, such as writing short titles or paragraphs. These rules can often be expressed as textual constraints on certain elements (such as maximum character count or minimum word count).



3. Schematron

Schematron is a widely used standard for performing 'soft-validation': describing certain structures in XML which are technically valid, but deemed undesired.

In short, Schematron is a declarative format that works like this:

- Define “rules”, selecting nodes which should be validated.
- Per rule, define reports and asserts testing these nodes.
- Both rules and tests are XSLT patterns, which are very similar to and can be transformed to standard XPath queries without problems.

```
<sch:schema xmlns:sch="http://purl.oclc.org/dsdl/schematron">
  <sch:title>An example schema</sch:title>
  <sch:rule context="p">
    <sch:assert test="@class">A paragraph must have a class</sch:assert>
  </sch:rule>
</sch:schema>
```

The rule context and test expressions can be transformed to a single XPath[2] query which can identify nodes that would trigger the assertion or report. For performance, it does not make much sense to split these queries: an assert for a body asserting a title node is present somewhere in the document still introduces a full-document-scan.

The Schematron reference implementation is based on XSLT. As FontoXML is implemented in pure JavaScript and aims to have as few server-side components

as possible, there are not many usable XSLT processors around. Also, because FontoXML is an editor, we expect many, generally very localized changes to happen very frequently. We do not want to fully process a large document multiple times per second. In order to provide a fluid user experience, we need the editor to update with a rate of 60 frames per second, just like a video game. This leaves us with a budget of roughly 16ms (1000ms / 60) per frame. Within this frame budget we need to do all the JavaScript updates and still allow time for the browser to do its layout and paint work. Another concern is the size of both documents and schemas: one of our clients loads documents up to 2MB in size, another client uses up to 3500 (automatically generated) Schematron rules. Fully processing these documents and their rules simply won't scale.

Because of these concerns, we have decided to roll our own implementation, optimized specifically for evaluating Schematron rules in an editor environment.

4. Implementation

4.1. Requirements

We decided to build our own optimized Schematron engine with the following concerns:

- Must be client-side, fully written in JavaScript
- Must be real-time (reactive to changes in the document)
- Must have quick-fix functionality (understandable by non-XML experts)
- Must not hold the UI thread hostage (remember: Javascript is single threaded)
  We aim to only work continuously for 16ms. Any more than this will make the browser framerate drop to below 60FPS.

4.2. Written in JavaScript, running client-side

The Schematron engine should run client-side so that it can provide feedback as fast as possible and does not introduce a server-side dependency. The editor should be able to run off-line: network latency and enterprise firewalls are largely not in our control. Besides this, the battery cost of a GSM or WiFi modem on mobile devices has to be taken into account.

The best XSLT processor in JavaScript at the time or writing is Saxon-JS[5]. For us, using this approach is not an option because of scalability. The naive XSLT-based approach would require the entire document to be processed after every change. As some of our clients work with XML documents ranging in the megabytes, this approach becomes infeasible due to performance constraints. The

[5] http://www.saxonica.com/saxon-js/index.xml

XSLT is not in our control, and we can not say anything about it. Ideally, we'd want to regard it as a black box.

A Schematron engine works using XSLT expressions, which are an extension of XPath queries. There are a few JavaScript XPath implementations available:

- Google's wicked-good-xpath[6], which is an XPath 1.0 implementation.
- XPath-JS[7], also an XPath 1.0 implementation.
- XPath.js[8], an XPath 2.0 implementation.

We also had the choice of building our own XPath engine, which would allow for anything we'd ever want. Because of our requirements stated earlier, we want to have tight control over the performance characteristics of the XPath implementation. We also wanted to use the same XPath implementation in other parts of the FontoXML editor and related products. We therefore decided to invest the time and build our own. This ensures we get the control we want, and also enables us to more easily implement any further optimizations or extensions in the future. For this implementation, we decided to implement an XPath 3.1[5] engine. This is the latest iteration of the XPath standard, which is currently in the candidate recommendation phase.

4.3. Real-time updates

Any change to the XML document could cause any of the Schematron rules to have different results. Because the size of the document is variable and could be very large, re-evaluating every rule on the entire document after every change would take too much time. We need a way to reduce the amount of Schematron rules that need to be to re-evaluated after each edit.

Let's take the XPath query `@someAttribute eq 'value'`. Given the element `<element someAttribute="value"/>`, we can see it will evaluate to `true()`.

If we change the value of this attribute to `<element someAttribute="some other value"/>`, we can see it will evaluate to `false()`.

If, instead, we change the node by adding an attribute to create `<element someOtherAttribute="meep" someAttribute="value"/>`, we can see the XPath will not change its' original value: `true()`.

We could say that the result of an XPath query is determined by the parts of the DOM it 'looks at'. These are its dependencies. These dependencies can be invalidated by changes on nodes. For instance, to be able to determine the result of the ancestor axis, the parent relation of all ancestors will be evaluated, this will introduce a dependency on the parent-child relation of these nodes.

[6] https://github.com/google/wicked-good-xpath
[7] https://github.com/andrejpavlovic/xpathjs
[8] https://github.com/ilinsky/xpath.js

The DOM standard defines the MutationObserver interface to allow a consuming API to react to changes in the DOM. It does this by recording each mutation as an object, called a MutationRecord[3], and exposing these objects for further processing after the mutation completes:

```
{
  type: 'childList' | 'attribute' | 'characterData',
  target: Node,
  addedNodes: Node[],
  removedNodes: Node[],
  nextSibling: Node?,
  previousSibling: Node?,
  attributeName: String?,
  oldValue: String?
}
```

The same interface is implemented by our editor to track changes made in an XML document. These records are used throughout the editor, for instance, for implementing the undo/redo functionality.

By tracking the dependencies of queries and using MutationRecords, we can make it so that for any given edit, only the affected Schematron rules (i.e. XPath queries) has to be re-evaluated, instead of all of them. For example, any rules on the presence of certain elements do not have to be evaluated if we're only working on an attribute. This removes the bulk of the processing needed to keep the Schematron results up to date.

We store the dependencies for our XPath queries in a two-level Map data structure:

```
node -> type -> query[]
```

This way, given a MutationRecord, we can look up the possibly affected queries in constant time (`O(1)`).

We update this data structure whenever we run an XPath query. First, we transform the XPath string to an abstract syntax tree using a parser generated with the wonderful peg.js parser generator[9]. We then compile this syntax tree to a set of DOM traversals.

By using a facade for accessing all DOM relations, we can record the traversed relations as dependencies of a specific operation such as the evaluation of an XPath query. As a bonus, this facade makes the engine independent of the interface of any specific DOM implementation, as it can be used as a translation layer.

The facade simply consists of functions such as the following:

```
class DependencyTrackingDomFacade {
  getChildNodes (node) {
```

[9] https://pegjs.org/

```
      registerDependency(
        this._dependenciesByNodeIdAndKind,
        node,
       'childList');
      return this._dom.getChildNodes(node);
    }
    getParentNode (node) {
      var parentNode = this._dom.getParentNode(node);
      if (parentNode) {
        registerDependency(
          this._dependenciesByNodeIdAndKind,
          parentNode,
          'childList');
      }
      return parentNode;
    }
    getAttribute (node, attributeName) {
      registerDependency(
        this._dependenciesByNodeIdAndKind,
        node,
        'attribute');
      return this._dom.getAttribute(node, attributeName);
    }
    getData (node) {
      registerDependency(
        this._dependenciesByNodeIdAndKind,
        node,
       'characterData');
      return this._dom.getData(node);
    }
  }
```

This dynamic analysis of an XPath query allows us to regard the XPath as a black box. We do not impose any additional requirements to the structure of XPath expressions, determining for instance streamability. It also does not block further improvements such as static analysis [4]. This approach for dynamic analysis is simple to implement. Although it does not make hard XPath queries easier to evaluate, it does provide a base for memoization[10]. In the future, we will want to use static analysis to provide partial queries, indices or query simplification.

4.4. Putting everything together

Given the following Schematron snippet:

[10]Memoization is a technique used to reuse the result of a function if its' parameters are the same as a previous execution.

```
<sch:rule context="/html/body/div/p">
  <sch:assert test="not(@class) or @class=('title', 'intro')">
    A paragraph should have either no class or the title or intro class.
  </sch:assert>
</sch:rule/>
```

Combining the context and test will result in the following query for reportable nodes:

```
/html/body/div/p[not(not(@class) or @class=("title", "intro"))]
```

Running this query will make the XPath engine 'look at' a number of properties of DOM nodes. This gives us the following dependencies:

- html -> 'childList'
- body -> 'childList'
- Every div in the body -> 'childList'
- Every p in these divs -> 'attribute'

This query will be triggered for re-evaluation by some of the following changes in the DOM:

- Adding a new body to the html element
  This changes the childList of the html element, and can possibly introduce new div elements with new p elements.
- Adding or removing divs
- Adding or removing paragraphs in these div elements
- Changing the class attribute on the paragraphs

This query will not be affected by changes like editing the contents of a paragraph or adding a title element in the head element of the html element.

Note that if the dependencies of a query have changed, it does not mean the result of the query has changed: `@someAttribute => string() => starts-with('abc')` can evaluate to true for many different values of `@someAttribute`: `'abc', 'abcd', 'abcX', etc`.

This causes a number of false positives. In the example given above, the following changes will also trigger a re-evaluation:

- Adding a head element to the html element (the childList of html changes)
- Moving around div elements in the body (the childList of body changes)

These will never change the result of this query, causing a needless re-evaluation. However, as these actions do not happen as frequently as typing text in a paragraph, the impact on performance of this limitation is minimal. The dependency tracking approach prevents re-evaluation of most queries, most of the time. This is sufficient to provide acceptable performance for our requirements. In the future we plan to investigate memoizing intermediate results in order to prevent full re-

evaluations, stopping evaluation if a part of the query resolves to the same result as before.

This also works in a subtly different way: the set of dependencies of a query may change, while the result remains the same. For example, take a query containing the conditional expression: `@A or @B`. Because the `or` expression may never evaluate `@B` if attribute A is present, removing this attribute forces it to also look at the other. The end result may remain the same, but the DOM has been traversed in a different way, causing a different set of dependencies.

4.5. Quick fix

Now that we can generate reports, and re-evaluate them quickly, we can move on to enabling users to fix any issues that have been detected.

The FontoXML editor uses the concept of 'operations', small units of functionality describing things like opening a modal, setting an attribute, wrapping a range of text, inserting a new element or any combination thereof.

These operations can be used to encode the mutations required to fix content issues, and can be mixed into the Schematron XML notation as follows:

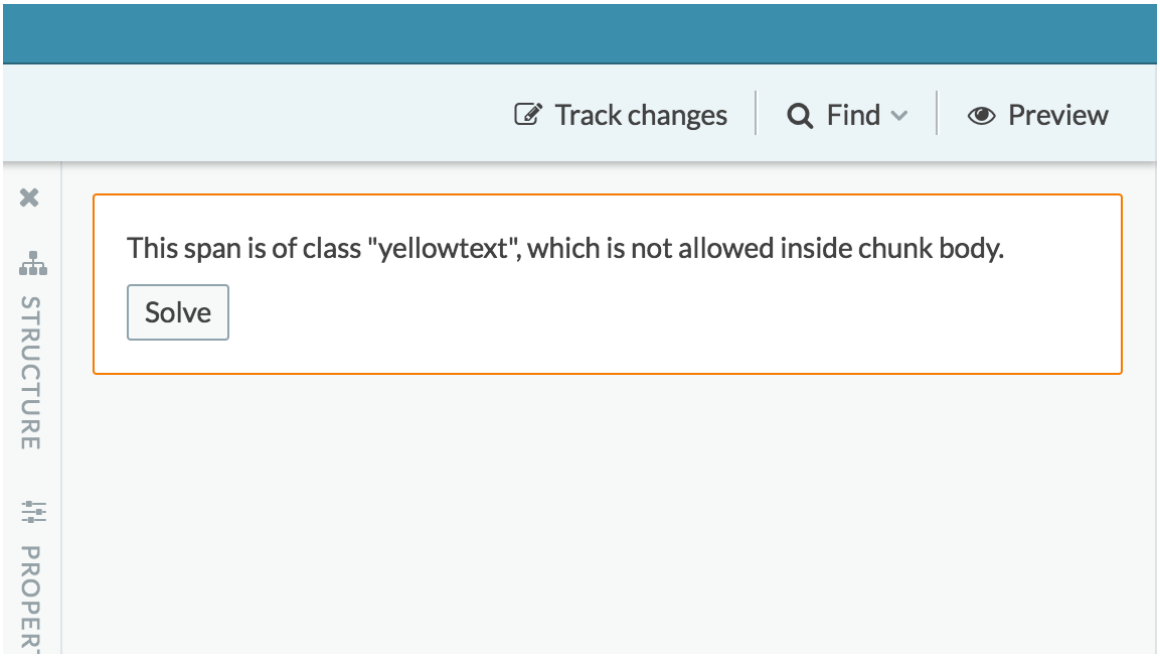
```
<sch:rule context="//span">
  <sch:assert test="@class = ('strong', 'italic')">
    <fonto:message>
      A
      <sch:value select="fonto:friendly-name(.)"/>
      must have the class 'strong' or 'italic'. It has the class
      <sch:value select="if (@class) then @class else 'No class'"/>
      . This is wrong.
    </fonto:message>
    <fonto:fix name="set-attribute" value="strong"
               label="Convert to Strong">
    <fonto:fix name="set-attribute" value="italic"
               label="Convert to Italic">
    <fonto:fix name="unwrap-node" label="Unwrap this">
  </sch:assert>
</sch:rule>
```

To make the report easier to parse, we have placed the message in an additional element.

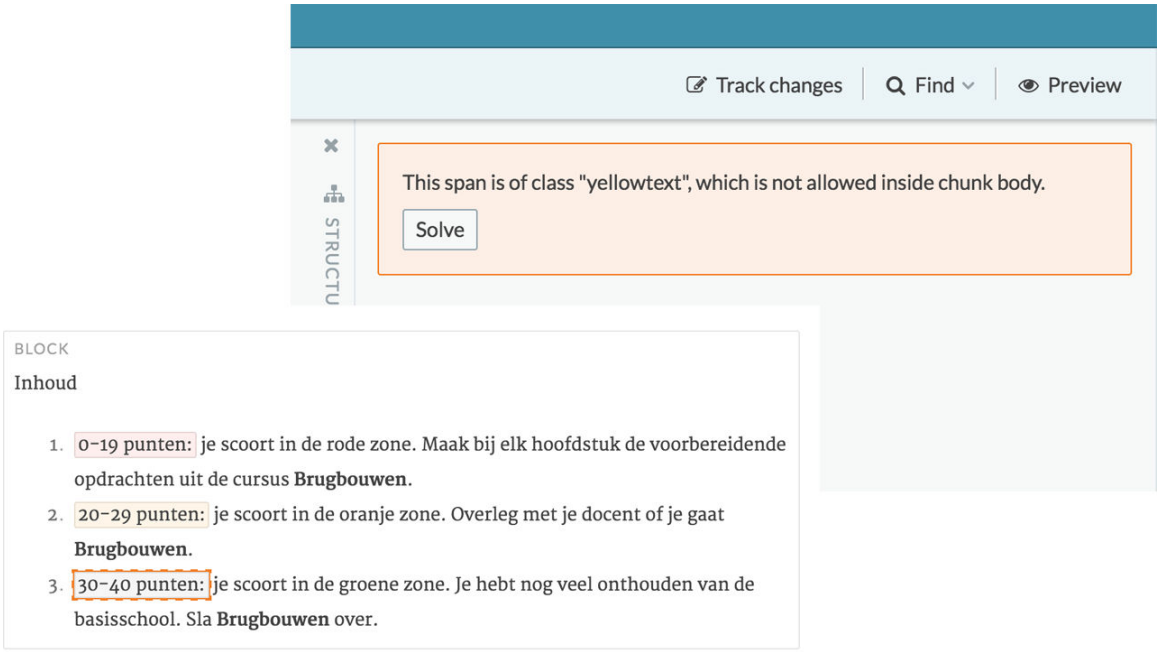
4.6. UI

The soft-validation reports can be used as a list of tasks that an author should process. As our authors are not all XML-experts, it is important to visualize as much context as possible. To do this, we did the following:

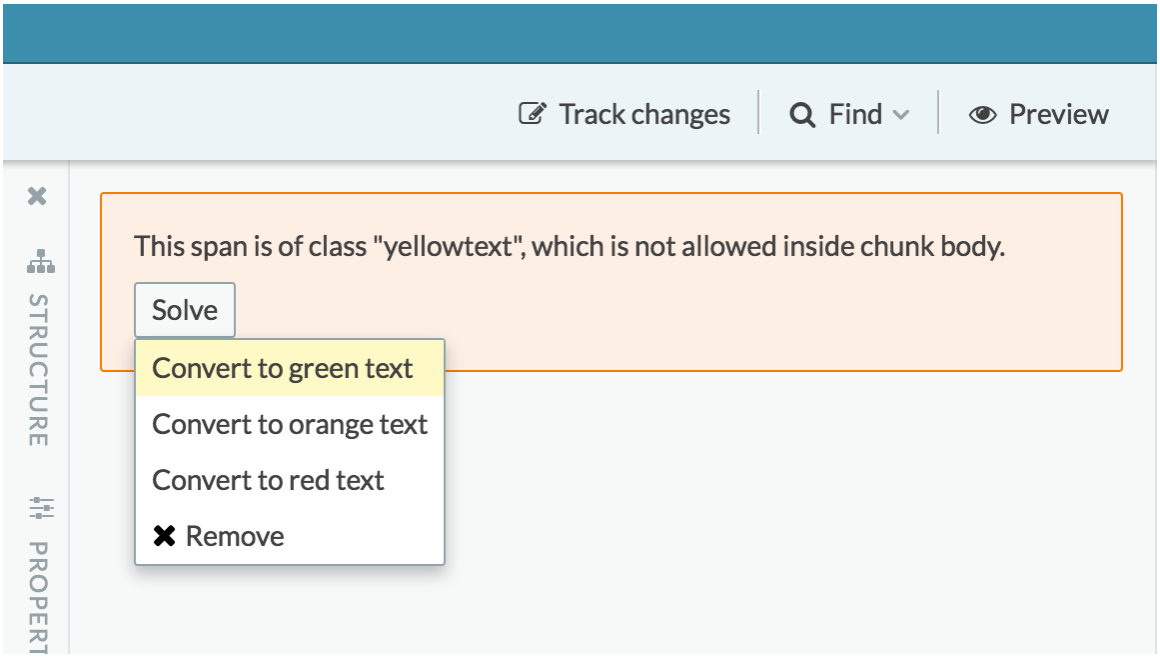
1. The reports are visualized as cards in a side panel, allowing them to be visible alongside the editor.



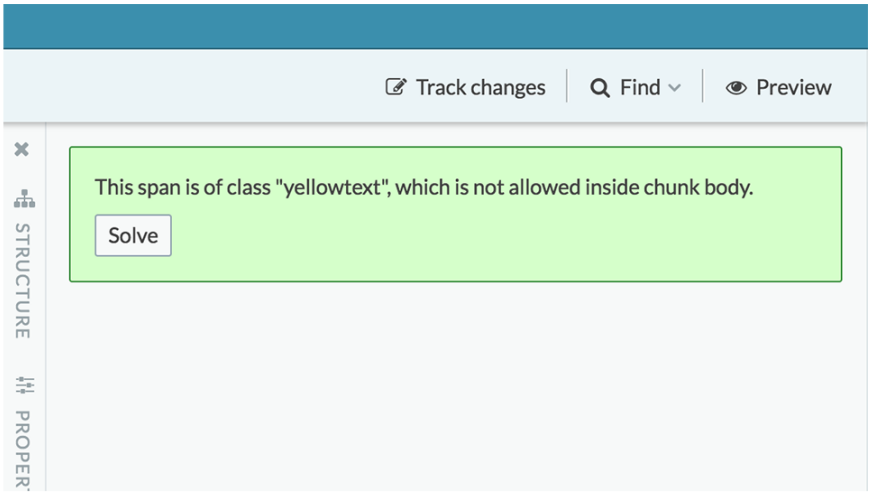
2. Clicking on a report highlights the element causing the report and brings it into view in the editor.



3. All reports having one or more quick-fix operations available display a ‘solve’ button, which opens the quick-fix menu.



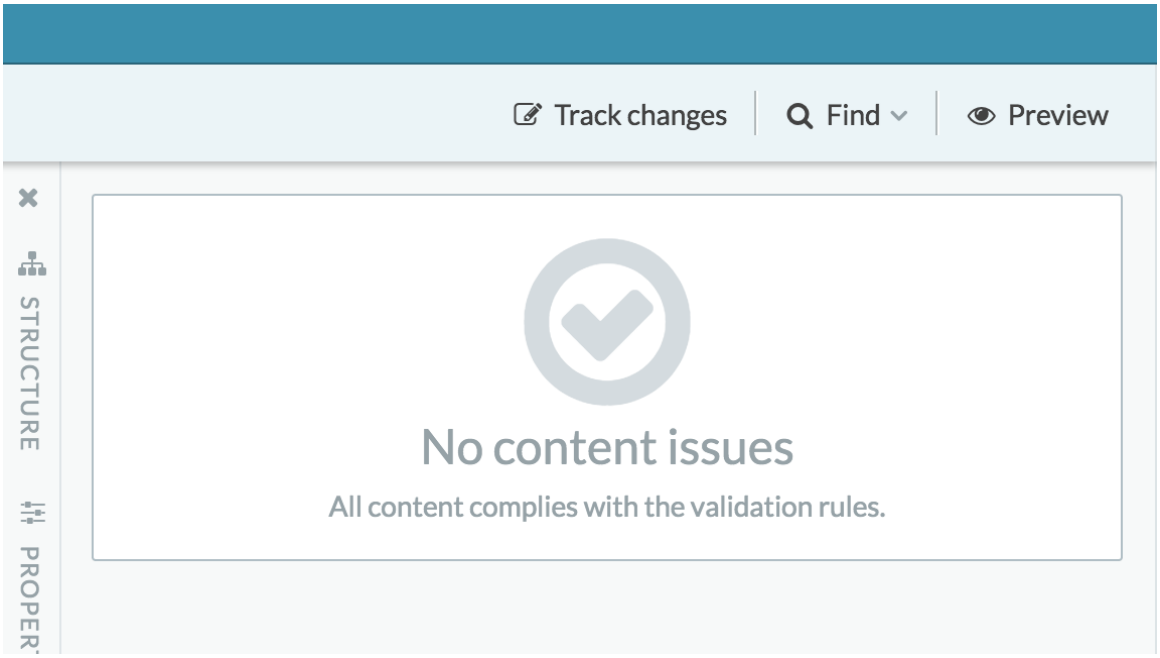
4. After resolving the error (using either the quick-fix menu or any other action), the card shows a brief animation and is removed.



Brugbouwen.

3. 30–40 punten: je scoort in de groene zone. Je hebt nog veel onthouden van de basisschool. Sla **Brugbouwen** over.

5. When all issues have been resolved, the editor reports this fact to the author.



5. Future work

5.1. Performance

At the moment, performance is highly dependent on the way the Schematron rules and tests are written. A constraint such as preventing referencing unknown ids can be written as this:

```
<rule context="@id-ref">
  <let name="idref" value="."/>
  <assert test="//*[@id=$idref]/>
</rule>
```

This rule will be rewritten to this XPath: `boolean(// *[@id-ref][let $idref := . return //*[@id = $idref]])`.

The XPath will cause us to evaluate the comparison `O(N2)` times: we will scan the whole document for id attributes once for every idref. Also, this query will introduce childList dependencies on all of the elements in the document. Building and maintaining a lookup table for all nodes with an id attribute can speed up these queries significantly.

5.2. Preventing worsening the document

In the current implementation, soft-validation rules never prevent any editing operation. However, we might want to influence a number of actions (like copy/paste or inserting new nodes) to prevent the document from getting less valid

according to these rules. An operation could be blocked if it ends up making the document fail more soft-validation tests. This needs to be thoroughly user-tested to identify the conditions where it would be useful to temporarily make the document less valid (such as when pasting content from somewhere else, or when splitting an 'invalid' paragraph using the enter key in order to convert the individual parts to a valid structure).

5.3. Using Schematron quickfix

A small schema has been whipped up to provide references to quick fix operations. In the future, these quick fix descriptions should be described using the Schematron Quickfix schema[6].

5.4. Open sourcing

We are looking into making the XPath engine open source, possibly including the dependency tracking mechanism. We expect it will have many applications outside of Schematron, like writing modern JavaScript apps that manipulate XML documents, or any other data representing a DOM.

6. Conclusions

The requirements of a soft-validation engine operating in a dynamic environment such as an editor are different from one reasoning over static documents.

Because changes in an editor are usually localized, we should not need to reprocess a full document every single keystroke. Using mutation records and dynamic dependency tracking as a way to mark queries on the DOM as dirty, we can use the edits to determine a smaller set of possibly affected queries.

Soft validation is a great help in guiding authors in writing "good" content.

By considering the soft-validation report as a "to do list", an author can keep track of their progress. By providing quick-fixes, the author always knows what to do and can efficiently resolve most content issues.

Bibliography

[1] Information technology — Document Schema Definition Languages (DSDL) — Part 3: Rule-based validation, Schematron, International Standard ISO/IEC 19757-3, Geneva, Switzerland : ISO

[2] XML Path Language (XPath) 1.0. W3C Recommendation, 16 November 1999. Ed. James Clark and Steve DeRose https://www.w3.org/TR/xpath/

[3] DOM living standard, 19 Januari 2017 on MutationRecord. https://dom.spec.whatwg.org/#interface-mutationrecord

[4] XSLT and XPath Optimization, March 2001, Ed. Michael Kay. http://www.saxonica.com/papers/xslt_xpath.pdf

[5] XML Path Language (XPath) 3.1. W3C Candidate Recommendation, 16 December 2016. Ed. Jonathan Robie, Michael Dyck, and Josh Spiegel. https://www.w3.org/TR/xpath-31/

[6] Schematron quickfix. Ed. Nico Kutscherauer. http://www.schematron-quickfix.com/