# Bridging the gap between knowledge modelling and technical documentation

## *Engage subject-matter experts to contribute to knowledge management and help them write accurate & correct documentation.*

Bert Willems

*FontoXML*

<bert.willems@fontoxml.com>

**Abstract**

*This paper describes an architecture which allows subject-matter experts and the systems to co-create both structured content and knowledge models. The proposed architecture creates a knowledge model from structured content which, in turn, is queried to validate and improve the accuracy and correctness of structured content leveraging the expertise of the subject-matter expert. The proposed architecture effectively describes a feedback loop.*

## 1. Introduction

Writing content is hard. One has to understand the subject and the intended target audience and one must be able to express oneself in written word.

Fortunately, one does not usually stand alone. There is software to support one's writing endeavors. A well-known piece of software integrated into virtually any text editor out there is the spell checker. A spell checker typically works on individual words without looking at the meaning: as long as the word is spelled correctly, it is happy. More advanced are grammar checkers, which typically work by looking at sentences as a whole. They help authors write sentences that are correct from a grammatical perspective as prescribed by a particular language.

Readability checkers help authors write sentences that are easy to read. You don't want to squander your exquisite phrasing if your target audience is 6 years old. Several industries have defined a standardized subset of languages in order to improve readability, like ASD STE-100 Simplified Technical English.

However, none of the above prevents authors from writing complete and utter nonsense as long as it is spelled well, grammatically correct, and easy to read. Although this may seem like a benefit in some cases, in technical documentation it is not. An important use of documentation, whether online or printed, is to help users to do their work as efficiently as possible.

This paper proposes a solution to help authors to write *accurate* and *correct* documentation by bridging the gap between knowledge modelling and technical documentation. The first part of this paper describes a general architecture. In the second part a practical implementation is explored to prove the prosoed architecture can be built. The final part holds the conclusions and future work.

## 2. Structured Content feedback loop architecture

Technical documentation, or more generally speaking body of knowledge, contains valuable information from which knowledge models can be build. There is a lot of research in the area of automated extraction of facts to build knowledge models. Most of them rely on training sets put together by domain experts.

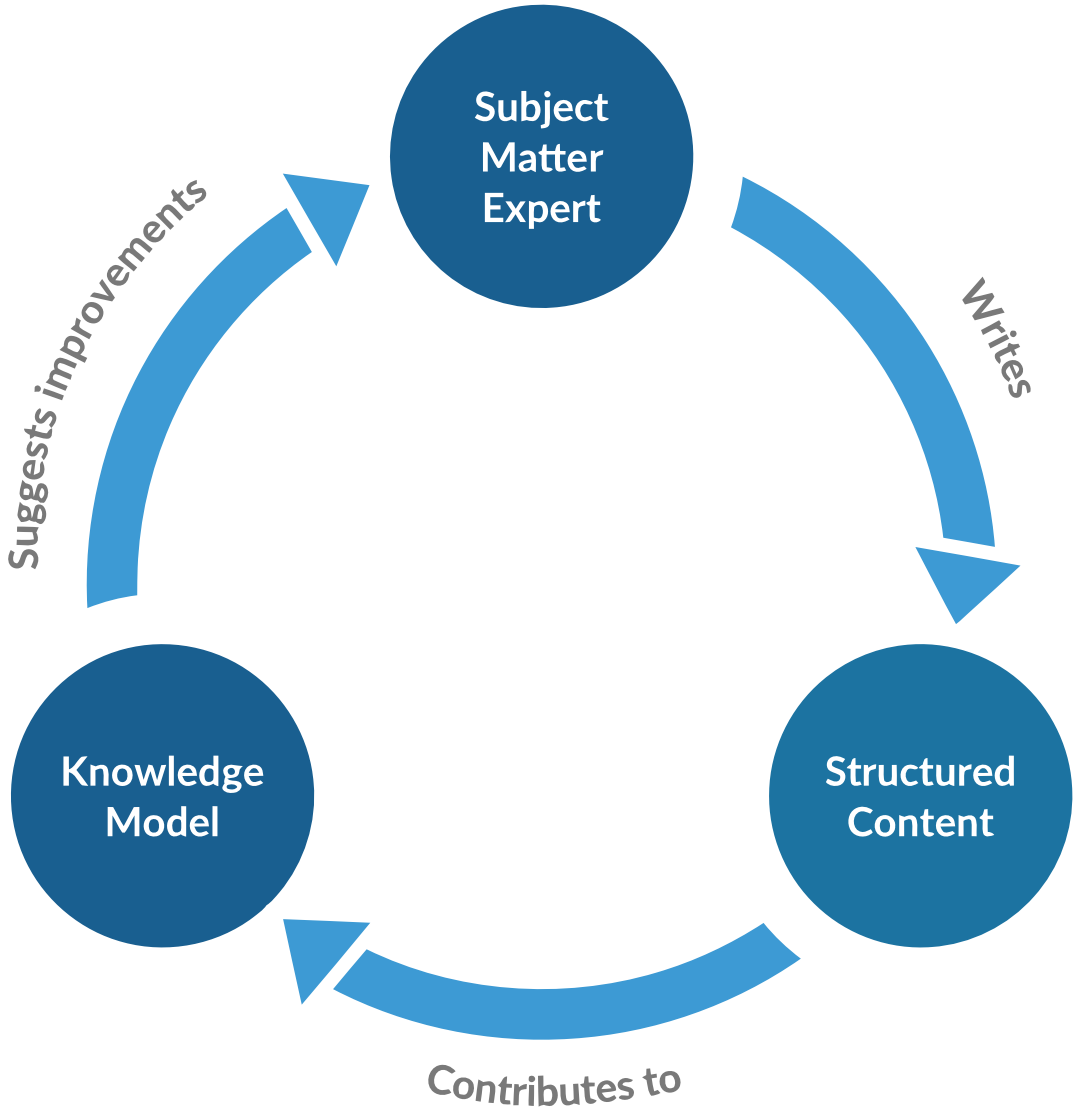
Structured content is usually written by subject-matter experts, who are domain experts themselves or act as proxies to experts and therefore are qualified to contribute to those knowledge models. This means that structured content is an excellent source of knowledge to build knowledge models from. Extracting facts from content, structured or not, is a well studied field.

There are numerous papers published which describe how information can be mined from content and how that information can be queried. However, none of those approaches are a 100% accurate, just like humans.



doi:10.14337/XMLLondon17.Willems01

**Figure 1. Loop overview**




Although the lines are blurring, in general computers are better at repetitive tasks while humans are better at unstructured problem-solving and empathy. This creates an interesting opportunity: *allow the subject-matter expert and the system to co-create both structured content and the knowledge model at the same time*.

Figure 1, "Loop overview" illustrates how ideas and knowledge are exchanged between subject-matter experts and systems:

We propose an architecture where the system ingests structured content in the form of XML from which a knowledge model is created. From this created knowledge model the system starts to suggest improvements to the subject-matter expert. The subject-matter expert evaluates the provided suggestions and, once accepted, improves the structured content. The improved structured content will in turn improve the created knowledge model, effectively closing a feedback loop.

The type of suggestions given by the system, dependents on the structure of the knowledge model and the algorithms used. In the problem space of technical documentation suggestions may include missing

prerequisites, opportunities for reuse and of course missing markup.

Even if the subject-matter expert decides to reject a suggestion, it is valuable. Consider the subject-matter expert rejecting a suggestion for a spelling correction: it might be the case that the word is missing from the dictionary or it should've been in the taxonomy. Some algorithms take counter examples as input to optimize their output. This means it is worthwhile to ask the subject-matter expert to provide feedback and update the knowledge graph accordingly, hence a secondary feedback loop.

# 3. Example implementation

This section describes an example implementation of the feedback architecture proposed in the previous section. The implementation is intentionally simple and straightforward but proves the loop can be built.

## 3.1. Problem example

Have a look at the following abbreviated example, taken from a procedure in a manual of our fictional ACME router:

```
Procedure: List all the files in the current
           working directory
Step 1: Execute the command 'dir'.
Result: An enumeration of all the files in the
        current directory.
```

An IT professional can, even without intimate knowledge of the ACME router, name at least a few (potential) errors in the seemingly simple snippet:

1. The procedure requires a terminal to be opened, which should have been encoded as the first step or as prerequisite.
2. It is unlikely the command is called 'dir' since that is Windows specific, it is more likely to be called 'ls' since it is more likely that the router is based on Linux.

In order to reason in the same way an IT pro can, the system requires the following knowledge to be available:

1. The `dir` command requires a terminal to be opened.
2. The `dir' and `ls` commands are directory listing commands.
3. The `dir` command is Windows specific.
4. The `ls` command is Linux specific.
5. The ACME routers run Linux.

## 3.2. Implementation

In order to create the feedback loop, the implementation works in two stages. The first stage creates the knowledge model from reference content. The second stage validates task-based content against the reference content. Where XML is given it is based on OASIS DITA (Oasis DITA 1.3).

### 3.2.1. Input reference documents

The following documents are used as reference documents from which the knowledge graph will be created. The areas that are relevant for creating the domain model are italicized.

```
<reference id="router">
  <title>ACME Router</title>
  <prolog>
    <prodinfo>
      <emphasis><prodname>ACME Router</prodname>
      <platform>Linux</platform></emphasis>
    </prodinfo>
  </prolog>
</reference>

<reference id="ls">
  <title>
    <emphasis><cmdname>ls</cmdname></emphasis>
  </title>
  <prolog>
    <prodinfo>
      <emphasis><prodname>ACME Router</prodname>
      <platform>Linux</platform></emphasis>
    </prodinfo>
  </prolog>
  <refbody>
    <p>The
        <emphasis><cmdname>ls</cmdname></emphasis>
      is a command used for <emphasis>
        <systemoutput>directory
          listing</systemoutput></emphasis>.
      It must run in the <emphasis>
        <uicontrol>terminal</uicontrol>
      </emphasis>.</p>
  </refbody>
</reference>

<reference id="dir">
  <title>
```

```
    <emphasis><cmdname>dir</cmdname></emphasis>
  </title>
  <prolog>
    <prodinfo>
      <emphasis><prodname>ACME Router</prodname>
      <platform>Windows</platform></emphasis>
    </prodinfo>
  </prolog>
  <refbody>
    <p>The
      <emphasis>
        <cmdname>dir</cmdname>
      </emphasis> is a command used
      for
      <emphasis>
        <systemoutput>directory
                        listing</systemoutput>
      </emphasis>.
      It must run in the
      <emphasis>
        <uicontrol>terminal</uicontrol>
      </emphasis>.
    </p>
  </refbody>
</reference>
```

**Note**

Some elements were removed from the documents for brevity.

**Note**

The examples given do not explicitly encode the "must run in a terminal" relation. This cannot be expressed well using standard DITA. Either specialization is needed or advanced text analysis software.

### 3.2.2. Knowledge graph

In order to validate the correctness according to the case described in the example section, the following knowledge must be captured by the model:

1. product to platform (used to determine the platform)
2. command to platform (used to determine whether a command is available)
3. command to result (used to infer the correct command based on platform)
4. command to required uicontrol (used to infer that a terminal is needed in order to perform the command)

The knowledge model is defined as an RDF graph. The graph is stored in a triple store, which allows semantic queries.

#### Constructing the knowledge graph

Using a simple extraction method the following triples can be extracted:

```
<#acme-router>
  runs <#linux> .
<#linux>
  a <#operating-system> .
<#windows>
  a <#operating-system> .
<#ls>
  a <#command> ;
  runsOn <#linux> ;
  requiresUI <#terminal> ;
  outputs <#directory-listing> .
<#dir>
  a <#command> ;
  runsOn <#windows> ;
  requiresUI <#terminal> ;
  outputs <#directory-listing> .
```

### 3.2.3. Task document

The following document is validated against the constructed knowledge graph:

```
<task>
<title>List all files and folder.</title>
  <prolog>
    <prodinfo>
      <prodname>ACME Router</prodname>
    </prodinfo>
  </prolog>
  <taskBody>
    <steps>
      <step>
        <cmd>Execute <cmdname>dir</cmdname>.</cmd>
      </step>
    </steps>
    <result>An enumeration of all the files
    in the current directory.</result>
  </taskBody>
</task>
```

### 3.2.4. Validating the task

Based on the given knowledge model and the task document, the system is able to find two inaccuracies:

1. The 'ls' command should've been used instead of the 'dir' command.
2. A terminal is required in order to execute the command.

#### Finding the correct command name

In order to find the correct command name the following traversals need to be made in the knowledge graph:

Input:

1. Product 'ACME Router'.
2. Command 'dir'.

Traversals:

1. Infer the 'ACME Router' runs on 'Linux'.
2. Infer the 'dir' command 'is available on' 'Windows'.
3. Infer the 'dir' command 'outputs' a 'directory listing'.
4. Infer the 'ls' command also 'outputs' a 'directory listing' AND 'is available on' 'Linux'.

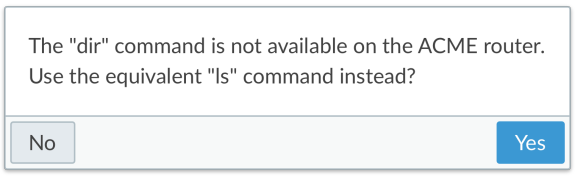
Based on the inputs and travels of the knowledge graph the system can return two facts to the user:

1. The command 'dir' is *not* available on 'Linux' and therefore *cannot* be correct.
2. The command 'ls' is a *substitute of* 'dir' and is available on 'Linux' and therefore *is* a suitable alternative.

Based on the inputs and traversal of the knowledge graph, the system can return the fact that the `dir` command should have been `ls` command. See Figure 2, "Replace command suggestions" for the suggestion. Approving the suggestion will change the XML into:

```
<cmd>Execute <cmdname>ls</cmdname>.</cmd>
```

**Figure 2. Replace command suggestions**




In order to find the missing prerequisite of the 'ls' command, the presence of a 'terminal', the following traversals need to be made in the knowledge graph:

Input:

1. Command 'ls'.

Traversals:

1. Infer the 'ls' requires a 'terminal'.

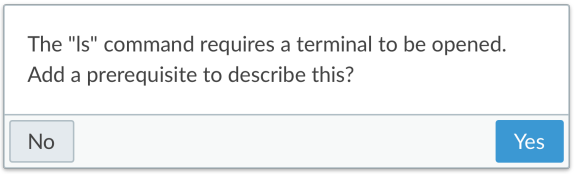
Check:

1. Check whether 'terminal' is referenced as a uicontrol.

Based on the inputs and traversal of the knowledge graph, the system can return the fact that a terminal is required. See Figure 3, "Replace command suggestions" for the suggestion. Approving the suggestion will insert the following XML snippet:

```
<prereq>Open a
          <uicontrol>terminal</uicontrol>.
</prereq>
```

**Figure 3. Replace command suggestions**



**Resulting task**

Now the resulting task is correct according to the available knowledge in the graph:

```
<task>
  <title>List all files and folder.</title>
  <prolog>
    <prodinfo>
      <prodname>ACME Router</prodname>
    </prodinfo>
  </prolog>
  <taskBody>
    <emphasis><prereq>Open a
      <uicontrol>terminal</uicontrol>.</prereq>
    </emphasis>
    <steps>
      <step>
        <cmd>Execute
          <emphasis>
            <cmdname>ls</cmdname>
          </emphasis>.</cmd>
      </step>
    </steps>
    <result>An enumeration of all the files
    in the current directory.</result>
  </taskBody>
</task>
```

Adding validation to the <result> of the task is left as an exercise to the reader of this paper.

# 4. Conclusions & Observations

As shown by the simple implementation presented in the previous section it is straightforward to derive a knowledge model based on reference content. Equally straightforward is the use of that knowledge model to validate task-based content for accurateness and completeness. This proves that the structured content feedback loop can be created.

However the implementation does not scale well: both the mapping to the knowledge model and the definition of the queries are done by hand. It is essentially an hard-coded rule engine.

The XML vocabulary used in the implementation example, does not support encoding all the relations out of the box. The DITA vocabulary does have a formal extension mechanism but, extending the vocabulary to support an evolving knowledge model does not scale either.

# 5. Future work

There is quite some work to be done in all the parts which make up the proposed system.

The first step is to remove the need for hard-coded rules by leveraging Information Extraction, a field of study which includes Part-of-speech (POS) tagging, phrase identification and word classification [1] and PATTY as described in Nakashole's 2012 PhD thesis [2] . Furthermore the mapping can be enhanced with the knowledge that can be mined from the XML schema as described in "Semi-Automatic Ontology Development" [3] and GRDDL [4].

The second step is to use Relational Machine Learning to query the knowledge model and provide useful suggestions and corrections to the subject-matter expert. The paper "A Review of Relational Machine Learning for Knowledge Graphs" [5] provides an excellent overview of that field of study.

Another step is to develop the software architecture based on OASIS Unstructured Information Management Architecture. This allows us to leverage and integrate existing components that are developed by third parties rather than developing everything ourselves.

The last, and perhaps the most important, step is to design and build an easy-to-use user interface. The suggestions must be easily understood and displayed in a relevant context to allow subject-matter experts to make quick and accurate decisions.

# Bibliography

[1] *Information Extraction and Named Entity Recognition*. Christopher Manning. Stanford University.
https://web.stanford.edu/class/cs124/lec/Information_Extraction_and_Named_Entity_Recognition.pdf

[2] *Automatic Extraction of Facts, Relations, and Entities for Web-Scale Knowledge Base Population*. Ndapandula T Nakashole.
http://nakashole.com/papers/2012-phd-thesis.pdf

[3] *Semi-Automatic Ontology Development*. Processes and Resources. Maria Teresa Pazienza and Armando Stellato.
doi:10.4018/978-1-46660-188-8

[4] *Gleaning Resource Descriptions from Dialects of Languages (GRDDL)*. Dan Connolly. 11 September 2007. World Wide Web Consortium (W3C).
http://www.w3.org/TR/grddl/

[5] *A Review of Relational Machine Learning for Knowledge Graphs*. Maximilian Nickel, Kevin Murphy, Volker Tresp, and Evgeniy Gabrilovich. 25 September 2015.
https://arxiv.org/pdf/1503.00759.pdf