

How To Write A Minimal L^AT_EX_{ML} Binding

Authors: Hang Yuan, Jinbo Zhang

Supervisor: Prof. Dr. Michael Kohlhase

February 20, 2015

Jacobs University Bremen

L^AT_EX has been widely used as a word processing tool among scholars, especially when one needs to use large quantities of mathematical representations. L^AT_EX is also a good choice for those who are meticulous about typographical quality of documents. However, L^AT_EX lacks a conversion tool to XML which DLMF(Digital Library of Mathematical Functions) uses for delivery. DLMF developed L^AT_EX_{ML}, trying to make a new typesetting system that allows users to be able to focus more on the content, but not the style, by providing extensive ways of customizations. In order to achieve this goal, building up the document class binding seems crucial, and yet L^AT_EX_{ML} seems fairly unfathomable for beginners. We want to make it easier for those who want to pick up using L^AT_EX_{ML} in the future, by going through how to construct a minimal L^AT_EX_{ML} binding step by step.

This document does not cover advanced topics related L^AT_EX_{ML}, and thus if you are interested the general theories, probably you can have the manual hand in hand with this document to have a better comprehension between theories and applications. In addition, We will refer you to the particular chapters in the manual, when needed.

1 Using L^AT_EX_{ML}

We are going to talk about various aspects of L^AT_EX_{ML}, and then we will move onto the workflow of creating your first L^AT_EX document class binding. In this tutorial, we use the command:

```
1 latexmlc <Filename> --format=XML --destination=<
    Filename> --log=<Filename>.log
```

for converting \TeX document into *.xml One quick note in regards to \LaTeX XML installation, when you think you have finished installing \LaTeX XML, run a simple conversion command. You should be able to see an XML interpretation of *mockDoc.tex* in a newly-generated XML file. It is totally fine to see tons of mysterious error messages at this point, because we have created anything yet. Under some circumstances when your \LaTeX XML doesn't seem to function, maybe you have overlooked the prerequisites such as `libxml2` or `libxslt` ¹.

For more information about how to use \LaTeX XML, please have a look at the \LaTeX XML Manual Chapter 2: Using \LaTeX XML.

2 How to Create A LaTeXXML Binding

The conversion from \TeX to XML is processed by \LaTeX XML. Basically \LaTeX XML maps the \TeX markups to the XML markups, more specifically macros, primitives and constructors. That's why you are able to customize the conversion between \TeX and XML, in three ways: modifying the bindings used by `latexml`, adding your own bindings that has not been implemented, and even creating your own \TeX style and \LaTeX binding which is exactly the goal of this tutorial.

2.1 Things We need

It probably would be a good idea to name every file after the same prefix which will make your life easier in the future. We need to have:

- *.tex** As your source file, so you can have something to convert from. You can write down whatever you want and based on this *.tex file, your other files will vary. Feel free to define your own macros into something unusual such that, even if you accidentally load the \TeX binding in \LaTeX XML, the conversion will fail, ensuring all conversions are done by \LaTeX XML binding.
- *.cls** For Xe \LaTeX , which essentially helps you to see what *.tex file looks like in a pdf format.
- *.cls.ltxml** \LaTeX XML binding, similar to the *.doc.cls you have for \LaTeX , but used for the conversion to other formats.
- *.rnc** The RelaxNG schema compact form, which defines the structure of your *.tex, crucial for executing tasks like placing the tags correctly and auto closing the tags when needed.

¹Please visit <http://dlmf.nist.gov/LaTeXML/get.html> for more information.

`trang.jar` (optional): \LaTeX XML cannot process the compact form scheme, and therefore you need `trang.jar` to convert your `*.rnc` into `*.rng`, unless you want to write your scheme in `*.rng` from the first, albeit this approach is not recommended for lack of efficiency and difficulty of maintenance.

After you have finished writing all the documents above, run `latexml`, and then you should be able to see the converted XML file for your `*.tex`. In the following chapters I will explain how to construct your `*.doc.ltxml` and `*.rnc`.

2.2 Minimal \LaTeX XML

Since \LaTeX binding is a perl module, we need to initialize a binding file by add the followings in the beginning of `*.doc.ltxml`:

```
1 package LaTeXXML::Package::Pool;
2 use strict;
3 use LaTeXXML::Package;
4 use warnings;
```

At the end of `*.doc.ltxml` don't forget to include

```
1 1;
```

to make sure perl work properly.

It will be good to read the \LaTeX XML Manual Chapter 4: Customization, before your proceed and come back to see how the theories are implemented.

Assuming you have read chapter 4 thoroughly, and get some feelings about how things work. Now you want to teach \LaTeX XML the new commands you created in your `*.tex` file. Let's look at an example below:

```
1 DefConstructor('\newline', "<mock:break/>");
```

The reason why I use the `break` as an example is because you might encounter problems dealing with `break` in \LaTeX XML. The two backslashes macro is preserved in pool package, that's why if you still use the regular `newline break` macro, your \LaTeX XML will have a malformed error. Renaming your `newline` macro in your `*.tex` will solve the problem for you.

After you link your `*.tex` file and `*.cls.ltxml` file by changing your document class in your `*.tex` into your \LaTeX XML binding name, in our case, "doc". \LaTeX XML will load your binding file, when it tries to do the conversion.

You might be wondering how \LaTeX XML reads your binding. To put it in a simple way, during the conversion process, whenever \LaTeX XML encounters a macro or

control sequence, it will look for its replacement in your binding and then put the replacement in `*.xml`. This is where things get a little tricky. How about the closing tag? Just like `section` macro, you declare where the `section` starts and where the next `section` starts, nevertheless, you never write now close section, so `LATEXML` will never close the section tags? Yes and no. Indeed `LATEXML` will have no clue of where to close the declared tags if we don't tell it when to do so. Using a schema can solve this problem.

2.3 RelaxNG Schema

Schema is a crucial document that decides how `*.xml` is constructed. When you are creating your own schema, it is a good idea to have your `*.tex` document open side by side to make sure your scheme works well with your `*.tex` file.

One good approach to test this is to create your expected `*.xml` by hand, according to your `*.tex`. You can easily accomplish this by using *emacs nxml mode* in which you have the freedom to write your expected `*.xml`, while validating your `*.xml` at the same time. If validation fails, you can see the error message instantly, such that you can debug your `*.xml` or schema accordingly.

Tutorial: Emacs: Nxml Mode

In our `mockDoc.rnc`, you can easily see under a document, there can be either `p` or `section`, and under a `section` there can be a title followed by `p` or a title followed by a `subsection`. This is because in the first section in `mockDoc.tex`, there is no `subsection` but text directly. But in the other `sections`, there are `subsections`. In your schema you need to consider all kinds of possible hierarchy of your elements.

Before you write your expected xml and RelaxNG schema, having a look at the links below can be beneficial:

I. RelaxNG Syntax Tutorial;

II. XML tutorial.

Some more improvements: If you have followed what we said, very likely you still have many errors when you use `LATEXML` to compile your files. Don't be frustrated by this, when we tried to make our first binding, it didn't exit at all. The success is within a reach. We only need to deal with two more things, namespace and putting spaces in your text.

We have a default namespace in the schema and we need to declare the schema in the binding and associate the prefix with the namespace. That's an easy step. Then we come to the obscure command of putting spaces between two words.

It is related to the architecture of L^AT_EX XML, which is far beyond the scope of this tutorial. So you can just do what is in the doc.cls.ltxml.

```
1 DefEnvironment('{document}', "<mock:document>#body</mock:document>", beforeDigest => sub { AssignValue(inPreamble => 0); });
```

Now you should have a minimal setup of what is required for a L^AT_EX XML binding.

Congratulations for being able to follow this tutorial to the end. After processing the makefile, you should be able to see the generated *.xml in your current directory which hopefully should look something similar to your expected *.xml!