`structview.sty`: Structures and Views in sTeX*

Michael Kohlhase
Jacobs University, Bremen
http://kwarc.info/kohlhase

November 10, 2015

Abstract

The `structview` package is part of the sTeX collection, a version of TeX/LaTeX that allows to markup TeX/LaTeX documents semantically without leaving the document format, essentially turning TeX/LaTeX into a document format for mathematical knowledge management (MKM).

This package supplies infrastructure for OMDoc structures and views: complex semantic relations between modules/theories.

*Version v1.4 (last revised 2015/04/02)

Contents

1 Introduction

EdN:1 [1]

2 The User Interface

The main contributions of the `modules` package are the `module` environment, which allows for lexical scoping of semantic macros with inheritance and the `\symdef` macro for declaration of semantic macros that underly the `module` scoping.

2.1 Package Options

`showmods` The `modules` package takes two options: If we set `showmods`[2], then the views (see
EdN:2 `qualifiedimports` Section 2.3) are shown. If we set the `qualifiedimports` option, then qualified imports are enabled. Qualified imports give more flexibility in module inheritance, but consume more internal memory. As qualified imports are not fully implemented at the moment, they are turned off by default see Limitation **??**.

`noauxreq` The option `noauxreq` prohibits the registration of `\@requiremodules` commands in the `aux` file. They are necessary for preloading the module signatures so that entries in the table of contents can have semantic macros; but as they sometimes cause trouble the option allows to turn off preloading.

`showmeta` If the `showmeta` is set, then the metadata keys are shown (see [Koh15] for details and customization options).

2.2 Structures

`importmodulevia` The `\importmodule` macro has a variant `\importmodulevia` that allows the specification of a theory morphism to be applied. `\importmodulevia{`⟨*thyid*⟩`}{`⟨*assignments*⟩`}` specifies the "source theory" via its identifier ⟨*thyid*⟩ and the morphism by
EdN:3 ⟨*assignments*⟩. There are four kinds:[3]

`\vassign` **symbol assignments** via `\vassign{`⟨*sym*⟩`}{`⟨*exp*⟩`}`, which defines the symbol ⟨*sym*⟩ introduced in the current theory by an expression ⟨*exp*⟩ in the source theory.

`\fassign` **function assignments** via `\fassign{`⟨*bvars*⟩`}{`⟨*pat*⟩`}{`⟨*exp*⟩`}`, is a variant which defines a function symbol ⟨*sym*⟩ introduced in the current theory by mapping a pattern expression ⟨*pat*⟩ (⟨*sym*⟩ applied to ⟨*bvars*⟩) to an expression ⟨*exp*⟩ in the source theory on bound variables ⟨*bvars*⟩.

`\tassign` **term assignments** via `\tassign[`⟨*source-cd*⟩`]{`⟨*tname*⟩`}{`⟨*source-tname*⟩`}`, which assigns to the term with name ⟨*tname*⟩ in the current theory a term with name ⟨*source-tname*⟩ in the theory ⟨*source-cd*⟩ whose default value is the source theory.

[1] EdNote: What are structures and views?

[2] EdNote: This mechanism does not work yet, since we cannot disable it when importing modules and that leads to unwanted boxes. What we need to do instead is to tweak the sms utility to use an internal version that never shows anything during sms reading.

[3] EdNote: MK: this needs to be consolidated and researched better.

\ttassign **term text assignments** via \tassign{⟨*tname*⟩}{⟨*text*⟩}, which defines a term with name ⟨*tname*⟩ in the current theory via a definitional text.

```
\begin{module}[id=ring]
\begin{importmodulevia}{monoid}
  \vassign{rbase}\magbase
  \fassign{a,b}{\rtimes{A}B}{\magmaop{a}b}
  \vassign{rone}\monunit
\end{importmodulevia}
\symdef{rbase}{G}
\symdef[name=rtimes]{rtimesOp}{\cdot}
\symdef{rtimes}[2]{\infix\rtimesOp{#1}{#2}}
\symdef{rone}{1}
\begin{importmodulevia}{cgroup}
  \vassign{rplus}\magmaop
  \vassign{rzero}\monunit
  \vassign{rinvOp}\cginvOp
\end{importmodulevia}
\symdef[name=rplus]{rplusOp}{+}
\symdef{rplus}[2]{\infix\rplusOp{#1}{#2}}
\symdef[name=rminus]{rminusOp}{-}
\symdef{rminus}[1]{\infix\rminusOp{#1}{#2}}
...
\end{module}
```

Example 1: A Module for Rings with inheritance from monoids and commutative groups

2.3 Views

A view is a mapping between modules, such that all model assumptions (axioms) of the source module are satisfied in the target module. [4]

EdN:4

3 Limitations & Extensions

In this section we will discuss limitations and possible extensions of the `modules` package. Any contributions and extension ideas are welcome; please discuss ideas, requests, fixes, etc on the sTeX TRAC [**sTeX:online**].

4 The Implementation

The `modules` package generates two files: the LaTeX package (all the code between ⟨*package⟩ and ⟨/package⟩) and the LaTeXML bindings (between ⟨*ltxml⟩ and

[4] EdNote: Document and make Examples

⟨/ltxml⟩). We keep the corresponding code fragments together, since the documentation applies to both of them and to prevent them from getting out of sync.

4.1 Package Options

We declare some switches which will modify the behavior according to the package options. Generally, an option `xxx` will just set the appropriate switches to true (otherwise they stay false). The options we are not using, we pass on to the `sref` package we require next.

```
⟨*package⟩
\DeclareOption*{\PassOptionsToPackage{\CurrentOption}{modules}}
\ProcessOptions
```

LaTeXML does not support module options yet, so we do not have to do anything here for the LaTeXML bindings. We only set up the Perl packages (and tell `emacs` about the appropriate mode for convenience

The next measure is to ensure that the `sref` and `xcomment` packages are loaded (in the right version). For LaTeXML, we also initialize the package inclusions.

```
\RequirePackage{modules}
```

4.2 Structures

`\importmodulevia` The `importmodulevia` environment just calls `\importmodule`, but to get around the group, we first define a local macro `\@@doit`, which does that and can be called with an `\aftergroup` to escape the environment grouping introduced by `importmodulevia`.

```
\newenvironment{importmodulevia}[2][]{%
  \gdef\@@doit{\importmodule[#1]{#2}}%
  \ifmod@show\par\noindent importing module #2 via \@@doit\fi%
}{%
  \aftergroup\@@doit\ifmod@show end import\fi%
}%
```

`\*assign`

```
\newrobustcmd\vassign[3][]{\ifmod@show\ensuremath{#2\mapsto #3}, \fi}%
\newrobustcmd\tassign[3][]{\ifmod@show #2\ensuremath{\mapsto} #3, \fi}%
\newrobustcmd\fassign[4][]{\ifmod@show \ensuremath{#3\mapsto #4}, \fi}%
\newrobustcmd\ttassign[3][]{\ifmod@show #2\ensuremath{\mapsto} ``#3'', \fi}%
⟨/package⟩
```

4.3 Views

We first prepare the ground by defining the keys for the `view` environment.

```
⟨*package⟩
\srefaddidkey{view}
\addmetakey*{view}{title}
\addmetakey{view}{display}
\addmetakey{view}{from}
```

```
\addmetakey{view}{to}
\addmetakey{view}{creators}
\addmetakey{view}{contributors}
\addmetakey{view}{srccite}
\addmetakey{view}{type}
\addmetakey[sms]{view}{ext}
⟨/package⟩
⟨*ltxml⟩
DefKeyVal('view','id','Semiverbatim');
DefKeyVal('view','from','Semiverbatim');
DefKeyVal('view','to','Semiverbatim');
DefKeyVal('view','title','Semiverbatim');
DefKeyVal('view','creators','Semiverbatim');
DefKeyVal('view','contributors','Semiverbatim');
DefKeyVal('view','display','Semiverbatim');
DefKeyVal('view','ext','Semiverbatim');
⟨/ltxml⟩
```

`\view@heading` Then we make a convenience macro for the view heading. This can be customized.

```
⟨*package⟩
\newcounter{view}[section]
\newrobustcmd\view@heading[4]{%
  \if@importing%
  \else%
    \stepcounter{view}%
    \edef\@display{#3}\edef\@title{#4}%
    \noindent%
      \ifx\@display\st@flow%
      \else%
        {\textbf{View} {\thesection.\theview} from \textsf{#1} to \textsf{#2}}%
        \sref@label@id{View \thesection.\theview}%
        \ifx\@title\@empty%
          \quad%
        \else%
          \quad(\@title)%
        \fi%
        \par\noindent%
      \fi%
      \ignorespaces%
  \fi%
}%ifmod@show
```

`view` The `view` environment relies on the `@view` environment (used also in the SᴛᴇX module signatures) for module bookkeeping and adds presentation (a heading and a box) if the `showmods` option is set.

```
\newenvironment{view}[3][]{%
  \metasetkeys{view}{#1}%
  \sref@target%
  \begin{@view}{#2}{#3}%
```

```
  \view@heading{#2}{#3}{\view@display}{\view@title}%
}{%
  \end{@view}%
  \ignorespaces%
}%
\ifmod@show\surroundwithmdframed{view}\fi%
⟨/package⟩
⟨*ltxml⟩
DefMacroI(T_CS('\begin{view}'),'OptionalKeyVals:view {}{}', sub {
  my ($gullet, $keyvals, $from_arg, $to_arg) = @_;
  my $from = ToString(Digest($from_arg));
  my $to = ToString(Digest($to_arg));
  my $from_file = ToString(GetKeyVal($keyvals,'from'));
  my $to_file = ToString(GetKeyVal($keyvals,'to'));
  my $ext = ToString(GetKeyVal($keyvals,'ext')) if $keyvals;
  $ext = 'sms' unless $ext;
  return (
    Tokenize("\\importmoduleI[load=$from_file]{$from}")->unlist,
    Tokenize("\\importmoduleI[load=$to_file]{$to}")->unlist,
    Invocation(T_CS('\begin{viewenv}'),$keyvals,$from_arg,$to_arg)->unlist
  );
});
DefMacroI('\end{view}',undef,'\end{viewenv}');
DefEnvironment('{viewenv} OptionalKeyVals:view {}{}',
   "<omdoc:theory-inclusion from='#2' to='#3'"
  .      " ?&defined(&GetKeyVal(#1,'id'))(xml:id='&GetKeyVal(#1,'id')')()>"
  .  "<omdoc:morphism>#body</omdoc:morphism>"
  ."</omdoc:theory-inclusion>");
⟨/ltxml⟩
```

`@view` The `@view` does the actual bookkeeping at the module level.

```
⟨*package⟩
\newenvironment{@view}[2]{%from, to
  \@importmodule[\view@from]{#1}{\view@ext}%
  \@importmodule[\view@to]{#2}{\view@ext}%
}{}%
```

`viewsketch` The `viewsketch` environment behaves like `view`, but only has text contents.

```
\newenvironment{viewsketch}[3][]{%
  \metasetkeys{view}{#1}%
  \sref@target%
  \begin{@view}{#2}{#3}%
  \view@heading{#2}{#3}{\view@display}{\view@title}%
}{%
  \end{@view}%
}%
\ifmod@show\surroundwithmdframed{viewsketch}\fi%
⟨/package⟩
⟨*ltxml⟩
# do the same for viewsketch, pity we cannot share some code.
```

```
DefMacroI(T_CS('\begin{viewsketch}'),'OptionalKeyVals:view {}{}', sub {
  my ($gullet, $keyvals, $from_arg, $to_arg) = @_;
  my $from = ToString(Digest($from_arg));
  my $to = ToString(Digest($to_arg));
  my $from_file = ToString(GetKeyVal($keyvals,'from'));
  my $to_file = ToString(GetKeyVal($keyvals,'to'));
  my $ext = ToString(GetKeyVal($keyvals,'ext')) if $keyvals;
  $ext = 'sms' unless $ext;
  return (
    Tokenize("\\importmoduleI[load=$from_file]{$from}")->unlist,
    Tokenize("\\importmoduleI[load=$to_file]{$to}")->unlist,
    Invocation(T_CS('\begin{viewsketchenv}'),$keyvals,$from_arg,$to_arg)->unlist
  );
});
DefMacroI('\end{viewsketch}',undef,'\end{viewsketchenv}');
DefEnvironment('{viewsketchenv} OptionalKeyVals:view {}{}',
   "<omdoc:theory-inclusion from='#2' to='#3'"
  .      " ?&defined(&GetKeyVal(#1,'id'))(xml:id='&GetKeyVal(#1,'id')')()>"
  .  "#body"
  ."</omdoc:theory-inclusion>");
```
⟨/ltxml⟩

EdN:5 `\obligation` The `\obligation` element does not do anything yet on the latexml side.[5]

```
⟨*package⟩
\newrobustcmd\obligation[3][]{%
  \if@importing%
  \else Axiom #2 is proven by \sref{#3}%
  \fi%
}%
⟨/package⟩
⟨*ltxml⟩
DefConstructor('\obligation [] {} {}',"<omdoc:obligation induced-by='#2' assertion='#3'/>");
⟨/ltxml⟩
```

Finally, we need to terminate the file with a success mark for perl.

```
⟨ltxml⟩1;
```

[5] EdNote: document above