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

Michael Kohlhase
FAU Erlangen-Nürnberg
`http://kwarc.info/kohlhase`

October 15, 2020

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.

Contents

[*]Version v1.6 (last revised 2020/10/14)

1 Introduction

Structures and views constitute ways of defining and relating theories in a theory graph that considerably extend the "object-oriented inheritance" constituted by the imports relation given by the sTeX `module` package.

Structures are like imports, only that they allow to define new theories via inheritance with renaming. Views relate pre-existing theories and model conceptual refinements, framing, and implementation relations, again via a mapping between the languages defined by the source and target theories; we call these mappings **theory morphisms**.

For details about theory morphisms we refer to [RK13], but hope to make the underlying concepts clear with examples.

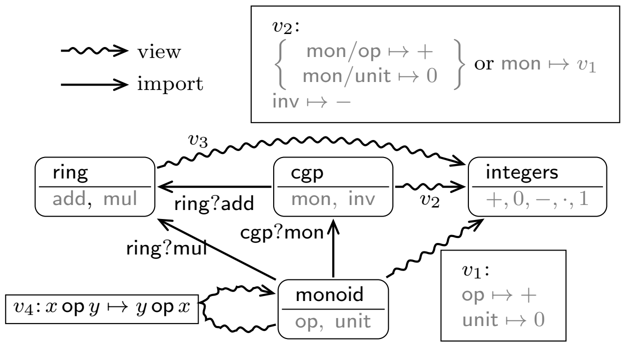

Figure 1: A Theory Graph with Structures and Views

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

`mh` The `mh` option turns on MathHub support; see [Koh18a].

2.2 Theory Morphisms

A theory morphism is a mapping between the languages of its source and target theory. This can be described mathematically using all the structures in the

[1] EdNote: explain the contribution of structures and views to theory graphs and synchronize with Figure 1.

sTeX distribution. However, in many situations, the language transformation of a morphism can be given in form of **assignments** that map symbols of the source theory to expressions of the target theory.

EdN:2 There are three kinds assignments:[2]

`\vassign` **symbol assignments** via `\vassign{`⟨*sym*⟩`}{`⟨*exp*⟩`}`, which maps a symbol ⟨*sym*⟩ from source theory an expression ⟨*exp*⟩ in the target theory.

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

`\tassign` **term assignments** via `\tassign{`⟨*sym*⟩`}{`⟨*tname*⟩`}`, another special case, where the value is the symbol with name ⟨*tname*⟩ in the target theory.

EdN:3 Figure 1 shows a concrete example[3]

The assignments above can be seen as abbreviations for a simple, formal definitions, which define a symbol of the source theory by an expression in the target theory.

2.3 Structures

`structure` Structures are specified by the `sstructure`[1] environment:

`\begin{sstructure}[`⟨*keys*⟩`]{`⟨*name*⟩`}{`⟨*sthy*⟩`}`⟨*morph*⟩`\end{sstructure}`

gives the structure the name ⟨*name*⟩, specifies the "source theory" via its identifier ⟨*sthy*⟩, and the morphism ⟨*morph*⟩. The `structure` environment takes the same keys as the `\importmodule` macro, which it generalizes. The morphism ⟨*morph*⟩ in the body of the `structure` environment specifies the morphism (see 2.2 above). In a structure, we take the target theory to be the current theory.

2.4 Views

A view is a mapping between modules, such that all model assumptions (axioms) of the source module are satisfied in the target module. For marking up views `view` the `structview` package supplies the `view` environment; see Figure 2 for the sTeX markup of view v_1 from Figure 1. The `view` environment takes one optional key/value argument followed by two mandatory ones: the names of the source and target modules. The `view` environment takes the following keys: `id` for a

EdN:4 name, `title` and `display` for visual presentation, `loadfrom`, `loadto`, and `ext`[4] for specifying the source files that supply the source and target modules, `creators`,

EdN:5 `contributors`, `srccite` for document metadata, and `type`[5].

[2] EdNote: MK: we need better macros here.
[3] EdNote: adapt when we fully understand this, and the implementation works.
[1] The old `importmodulevia` environment is now deprecated.
[4] EdNote: MK: we probably need toext and fromext here, but this never came up yet.
[5] EdNote: ????

```
\begin{module}[id=ring]
\symdef{rbase}{R}
\symdef{rtimes}[2]{\infix\cdot{#1}{#2}}
\symdef{rone}{1}
\begin{sstructure}{mul}{monoid}
  \tassign{magbase}{rbase}
  \fassign{a,b}{\magmaop{a}b}{\rtimes{a}b}
  \tassign{monunit}{rone}
\end{sstructure}
\symdef{rplus}[2]{\infix+{#1}{#2}}
\symdef{rminus}[1]{\infix-{#1}{#2}}
\begin{sstructure}{add}{cgroup}
  \fassign{a,b}{\magmaop{a}b}{\rplus{a}b}
  \tassign{monunit}{rzero}
  \tassign{cginvOp}{\rminus}
\end{sstructure}
...
\end{module}
```

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

```
\begin{view}{monoid}{integers}
  \vassign{magbase}{base}
  \fassign{a,b}{\magmaop{a}b}{\inttimes{a,b}}
  \tassign{monunit}{\intzero}
  \begin{assertion}
    The Integers with addition form a monoid in the obvious way.
  \end{assertion}
\end{view}
```

Example 2: A view from monoids to integers

`\inlineview` Just as for other statements (see [Koh18b]), we have an inline version of views that can be embedded into other statements: `\inlineview`. Intuitively, **inline views** are like inline assertions, however, they can bring along concepts from the source module. Example 3 shows a typical situation: the directory structure in a hierarchical file system forms a tree (the view), and we inherit concepts from that.

```
\begin{definition}
  \inlineview{tree}{The \trefi[hfs]{directory} structure in a
    \trefiii[hfs]{hierarchical}{file}{system} induces a \trefi[tree]{tree}
    or \trefi[cycle]{DAG}, so we inherit the concepts of (file system)
    \drefi[tree?root]{root}, \drefi[tree?parent]{parent} (directory),
    \drefi[tree?child]{child} from there.}
  ...
\end{definition}
```

Example 3: An inline view that brings along concepts.

`inlineView` Note that `\inlineview` does not specify the target module, that is the current module. We provide the `inlineView` environment as a block-level alternative.

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].

4 The Implementation

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⟩
\newif\if@structview@mh@\@structview@mh@false
\DeclareOption{mh}{\@structview@mh@true
\PassOptionsToPackage{\CurrentOption}{modules}}
\DeclareOption*{\PassOptionsToPackage{\CurrentOption}{modules}}
\ProcessOptions
```

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}
\if@structview@mh@\RequirePackage{structview-mh}\fi
```

4.2 Theory Morphisms by Assignments

EdN:6 *assign [6]

```
\newrobustcmd\vassign[3][]{\ifmod@show\ensuremath{#2\mapsto #3}, \fi\ignorespacesandpars}%
\newrobustcmd\fassign[4][]{\ifmod@show \ensuremath{#3(#2)\mapsto #4}, \fi\ignorespacesandpars}%
\newrobustcmd\tassign[3][]{\ifmod@show \ensuremath{#2\mapsto} #3, \fi\ignorespacesandpars}%
```

4.3 Structures

sstructure The structure 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 structure.

```
\newenvironment{sstructure}[3][]{%
  \gdef\@@doit{\importmodule[#1]{#3}}%
  \ifmod@show\par\noindent importing module #3 via \@@doit\fi%
}{%
  \aftergroup\@@doit\ifmod@show end import\fi%
}%
```

importmodulevia This is now deprecated, we give an error, but punt to structure.

```
\newenvironment{importmodulevia}[2][]%
{\PackageError{structview}%
  {The {importmodulevia} environment is deprecated}{use the {sstructure} instead!}%
  \begin{sstructure}[#1]{missing}{#2}}
{\end{sstructure}}
```

4.4 Views

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

```
\srefaddidkey{view}
\addmetakey*{view}{title}
\addmetakey{view}{display}
\addmetakey{view}{loadfrom}
\addmetakey{view}{loadto}
\addmetakey{view}{creators}
\addmetakey{view}{contributors}
\addmetakey{view}{srccite}
\addmetakey{view}{type}
```

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

```
\ifdef{\thesection}{\newcounter{view}[section]}{\newcounter{view}}
\newrobustcmd\view@heading[4]{%
  \if@importing%
  \else%
    \stepcounter{view}%
    \edef\@display{#3}\edef\@title{#4}%
```

[6]EdNote: probably get rid of the optional argument

```
    \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%
      \ignorespacesandpars%
  \fi%
}%ifmod@show
```

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

```
\newenvironment{view}[3][]{% keys, from, to
  \metasetkeys{view}{#1}%
  \sref@target%
  \begin{@view}{#2}{#3}%
  \view@heading{#2}{#3}{\view@display}{\view@title}%
}{%
  \end{@view}%
  \ignorespacesandpars%
}%
\ifmod@show\surroundwithmdframed{view}\fi%
```

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

```
\newenvironment{@view}[2]{%from, to
  \@importmodule[\view@loadfrom]{#1}{export}%
  \@importmodule[\view@loadto]{#2}{export}%
}{}%
```

`viewsketch` The `viewsketch` environment is deprecated, we give an error

```
\newenvironment{viewsketch}[3][]%
{\PackageError{structview}%
  {The {viewsketch} environment is deprecated}{use the {view} environment instead!}%
  \begin{view}[#1]{#2}{#3}}
{\end{view}}
```

`inlineView` We essentially do the same as for the `view` environment, but we are already in a module, hence we can use `\mod@id` for the target module and do not have to load it. All presentational keys are ignored.

```
\newenvironment{inlineView}[2][]{% keys, source
  \metasetkeys{view}{#1}\sref@target%
  \@importmodule[\view@loadfrom]{#2}{export}%
  \ignorespacesandpars}
{\ignorespacesandpars}
```

inlineview

```
\newcommand\inlineview[3][]{\begin{inlineView}[#1]{#2}{\mod@id}#3\end{inlineView}}
```

EdN:7

\obligation The `\obligation` element does not do anything yet on the latexml side.[7]

```
\newrobustcmd\obligation[3][]{%
  \if@importing%
  \else Axiom #2 is proven by \sref{#3}%
  \fi%
}%
⟨/package⟩
```

[7]EdNote: document above