TLA+ Mode

Supporting TLA+-based specifications in GNU Emacs
Edition 0.4, for TLA+ mode version 0.4
September 1994, for GNU Emacs 19

by Frank Wegmann

Copying

(The text below was taken from the TEXinfo 2.16 distribution).

The programs currently being distributed that relate to TLA-mode include lisp files for GNU Emacs, plus other separate programs (`makeinfo` and `texindex`). These programs are *free*; this means that everyone is free to use them and free to redistribute them on a free basis. The TLA-mode related programs are not in the public domain; they are copyrighted and there are restrictions on their distribution, but these restrictions are designed to permit everything that a good cooperating citizen would want to do. What is not allowed is to try to prevent others from further sharing any version of these programs that they might get from you.

Specifically, I want to make sure that you have the right to give away copies of the programs that relate to TLA-mode, that you receive source code or else can get it if you want it, that you can change these programs or use pieces of them in new free programs, and that you know you can do these things.

To make sure that everyone has such rights, I have to forbid you to deprive anyone else of these rights. For example, if you distribute copies of the TLA-mode related programs, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

Also, for my own protection, I must make certain that everyone finds out that there is no warranty for the programs that relate to TLA-mode. If these programs are modified by someone else and passed on, I want their recipients to know that what they have is not what I distributed, so that any problems introduced by others will not reflect on our reputation.

The precise conditions of the licenses for the programs currently being distributed that relate to TLA-mode are found in the General Public Licenses that accompany them.

Alpha Release Information

At the current stage nearly all planned features have been implemented. As before, those features being documented here, are implemented.

A substantial range of editing features now works. Insertion of templates for TLA+ headers and keywords basically work, although some fine tuning is still needed, especially updating the menus. Indentation as a key feature of every major mode works reasonably well. It supports bulleted lists of conjunctions and disjunctions even if they are inside other nested structures. There are still a few indentation bugs, but a very high percentage of code should now be indented according to the conventions of tTLA+.

The interface to other tools is now established although not everything is as stable as desired. Along with the definition of top modules for a given specification this should already be quite usable. Moreover, you can now go to the next error that was detected by the TBR (tTLA+Browser).

Full compatability to Lucid Emacs is still not accomplished, while compatability to Emacs 18 has finally been dropped. There is no reason not to use Emacs 19.

Introduction

This is a major mode for GNU Emacs (and derivatives) that serves as an environment for the specification process in TLA+, a specification language proposed by Leslie Lamport.

Apart from supporting the process of editing and modifying specifications by means of indentation and template insertion it also integrates related applications and tools via a common interface. It works not only with single files but with a whole range of files that syntactically constitute a TLA+ specification.

Together with a preliminary support for different variants of TLA+ this mode is designed to serve all users that are involved in writing specification code.

Here is a list of the major features:

- Insertion of templates such as module headers and keywords
- Short cuts for easy access to syntactic elements
- Sophisticated indentation for most tTLA+ expressions
- Support for multi-module specifications
- Running programs such as TeX, tpp or TBR
- Parsing tpp error messages
- Outlining for specifications
- Fontified specifications

1 Editing Features

This section covers the most commonly used features such as inserting templates for different types of modules, indenting spec code and commenting certain areas of text or code.

1.1 Inserting Keyword Templates

Using templates eases the process of inserting defined keywords and structures in a given language and reduces the amount of errors typically caused by typos. This also holds for specification code written in TLA+. All keywords can be easily accessed by using the mode's menu or a keyboard shortcut.

1.1.1 Module Headers

Module headers as they are defined in (Mester 93) not only cover the line that the keyword MODULE is in, but also include IMPORT, EXPORT and PARAMETERS clauses. The concept of header is used differently in that only accounts for the code line containing the keyword MODULE. It is, however, enriched with additional comments. In order to distinguish between these notions we just use *header* if we refer to the latter and *module header* in the other case.

Headers should follow conventions determining their information content, such as date and time of creation, information about the author, a one-line description and the template used by revision control systems such as RCS or SCCS. This is a typical example of a header that contains no type information but an already filled RCS ID slot:

```
MODULE LibraryActions
(* LibraryActions.t --- Library transactions                              *)
(* Created:     Tue Mar 29 11:10:48 1994                                  *)
(* Author:      Frank Wegmann <wegmann@talisker>                          *)
(* $Id: LibraryActions.t,v 1.2 1994/06/25 17:43:33 wegmann Exp wegmann $ *)
```

Before inserting a new header, the buffer is checked for an existing one. The user then has to confirm whether that header should be deleted. If not, no insertion is made and point remains unchanged. The complete header will be inserted and point is just behind the long dash to let the user provide short information about the module itself. If *tla-use-vc* is nil, don't insert a comment line for version control systems (default is t).

There are three shortcuts according to the possible type information. An untyped header contains no type information and is therefore considered a TEMPORAL module. However, it is recommended to indicate the module's type for better readability. Furthermore, the END keyword will be inserted at the very end of the buffer if it is not already present.

`C-c C-h C-u`
: Insert module header without type information.

`C-c C-h C-c`
: Insert module header indicating a CONSTANTS module.

`C-c C-h C-t`
: Insert module header indicating a TEMPORAL module.

tla-header *arg* Command

(`C-c C-h`) Insert template for header of current buffer.

arg determines the type of header, as specified by `tla-header-list`.

tla-delete-header Command

Delete complete module header. This is normally used from within `tla-header` and thus the user is not asked for confirmation.

tla-header-list User Option

List of module header types used by `tla-header`.

Each entry is a list with three elements. The first element is the key indicating the type of header. The second element is the entry appearing on the menu, and the third element is the string to insert just before the keyword MODULE.

tla-use-vc t Variable

If non-nil insert comment with RCS ID as last part of header.

1.1.2 Constant Modules

A certain range of keywords may appear in the body of a CONSTANTS module. The upper half of the submenu titled "Insert TLA+ Keyword" contains all allowed keywords. The command prefix `C-c C-k` is common to all keywords so that their respective templates can be inserted via shortcuts.

Before insertion takes place, the mode checks the current position of point. If point is inside a symbol definition or formula, the template gets inserted just behind that structure. A warning will be issued if the template is already present.

`C-c C-k C-i`
: Insert IMPORT template.

`C-c C-k C-x`
: Insert EXPORT template.

`C-c C-k C-l`
: Insert INCLUDE template. Optional additional keywords WITH and AS are not provided.

`C-c C-k C-p`
: Insert PARAMETERS template.

`C-c C-k C-c`
: Insert CONSTANTS template.

`C-c C-k C-b`
: Insert BOOLEANS template.

`C-c C-k C-j`
: Insert CONSTANT ASSUMPTIONS template.

`C-c C-k C-k`
: Insert CONSTANT THEOREMS template.

1.1.3 Temporal Modules

All keywords that are allowed in CONSTANTS modules can also be used in TEMPORAL modules. Furthermore, those keywords located in the lower half of the submenu can be used. The same restrictions that hold for the insertion of CONSTANTS MODULE keywords also hold for TEMPORAL ones. Additionally an error will be signalled when the user wants to insert one of the following keywords in a CONSTANTS module.

`C-c C-k C-r`
: Insert PREDICATES template.

`C-c C-k C-s`
: Insert STATE FUNCTIONS template.

`C-c C-k C-a`
: Insert ACTIONS template.

`C-c C-k C-f`
: Insert TRANSITION FUNCTIONS template.

`C-c C-k C-t`
: Insert TEMPORALS template.

`C-c C-k C-o`
: Insert ACTION ASSUMPTIONS template.

`C-c C-k C-n`
: Insert ACTION THEOREMS template.

`C-c C-k C-h`
: Insert TEMPORAL THEOREMS template.

tla-keyword-list User Option

List of tTLA+ keywords used by `tla-keyword`.

Each entry is a list with three elements. The first element is the key inserting the appropriate keyword. The second element is the entry appearing on the menu, and the third element is the string that will be inserted on a line by its own.

1.2 Commenting

It is often necessary to comment out temporarily a region of code. This can be done with the commands `C-c ;` and `C-c %`. `C-c ;` will comment out all lines in the current region, while `C-c %` will comment out the current symbol definition. To uncomment, simply type `C-u - C-c ;` to uncomment all lines in the region.

By default, these commands will insert or remove a single '`(*`' or '`*)`'. To insert more than one, give an argument. For example, `C-u 5 C-c %` will add five '`%`' to each line, while `C-u - 5 C-c %` will remove up to 5 '`%`' from each line.

However, commenting commands will work on the whole region or symbol definition, even if there is already a commented line in it. Since nested comments are not allowed in TLA+, this should be avoided. This support for multiple levels of commenting will be removed in a later version.

tla-comment-region *count* Command

(`C-c ;`) Add or remove '`(*`' from the beginning and '`*)`' from the end of each line in the current region, as specified by *count*. Comments will only be added if there is at least one uncommented line in the current region.

tla-comment-formula *count* Command

(`C-c %`) Add or remove '`(*`' from the beginning and '`*)`' from the end of each line in the current spec formula, as specified by *count*. A formula begins at the line containing '`==`' and ends at the line containing '`;`'.

tla-comment-paragraph *count* Command

Add or remove '`(*`' from the beginning and '`*)`' from the end of each line in the current paragraph, as specified by *count*. Since most paragraphs only comprise one line, this is of minor interest.

1.3 Formatting and Indenting

At the current stage of implementation, indentation is not complete. One of the shortcomings is that you can only indent single lines.

The characters indented are spaces, since tabs are not feasible for most existing tools. Therefore the current buffer will normally be untabified each time you save it. This is controlled by the value of `tla-auto-untabify` which is normally non-nil. Instead of inserting tabs, `M-i` should be used for inserting space up to the next tab stop position.

Indentation is controlled by the environment containing point. The keywords whose templates can be inserted all start at the very beginning of a line and thus become reindented properly if they don't start a line. Symbol definitions and formulas start with a uniform left-hand side made up of an identifier with or without additional type information ("qualified") and the ASCII representation of "equals by definition" ('`==`'). These are generally indented by the value of `tla-indent` (default 2). The first continuing line is indented by the value of `tla-inner-indent` (default 4), if there are no structures on the first line that request a different indentation.

This is a typical example of indented code taken from the BankAction module[1].

[1] The code in this section is taken from the specification of a distributed banking system, which is written in the variant of tTLA+ and originally derived from a previous version of Lamport's Introduction to TLA+.

```
ReceiveRequest(A : ATM)  ==
    :E: r :in: ATMMessage
     ::
        /\ pending[A].req = "None"
        /\ AtoB.Rcv(r,A)
        /\ Unchanged( bToA )
        /\ pending' = [CHG pending ; A :mapsto: r]
;
```

TLA mode looks for a number of character sequences that determine the indentation of the next line. Here three different indentations have been used: the first body line of the action gets the standard *tla-inner-indent*, while the indentation of the second line is controlled by the position of the existential quantifier '`:E:`'. The characters indicating the end of the scope description ('`::`') demand the indentation of the "bulleted" list of conjunctions below. Finally, the semicolon concludes the code of the action and is therefore indented the same number of spaces as the beginning line.

One type of indentation deserves a special treatment: if conjunctions and disjunctions are to be bulleted as in the example above, operators of the same kind are stacked upon each other. This holds when *tla-bullet-lorand* is non-nil (default). But structures can be nested, as can be seen in the following lines from ATMAction:

```
N == :E: A :in: ATM
      ::
         \/ /\ :E: am :in: ATMMessage :: AtoB.Send(am, A)
            /\ Unchanged( bToA )
         \/ /\ :E: bm :in: BankMessage :: BtoA.Rcv(bm, A)
            /\ Unchanged( aToB )
;
```

The second and fourth conjunctions are properly indented since TLA mode knows that only operators of the same kind can be stacked upon each other. If *tla-bullet-lorand* were nil that line would have been indented according to the position of the double colon above. This behavior forces the user to pursue a common style of writing specification code. A last example shows that bulleting works even across other conjunctions or disjunctions that are enclosed in a separate structure:

```
Elevator == :EE: Light, Location, Direction
             ::
                /\ Ser(Light,Location,Direction).Init
                /\ [] ([
                        \/ :E: f :in: Floor
                            :: Ser(Light,Location,Direction).PressButton(f)
                               \/ Lif(Light,Location,Direction).MoveUp
                               \/ Lif(Light,Location,Direction).MoveDown
                       ] _ (Light,Location,Direction)
                      )
                /\ :A: f :in: Floor
                    :: Light[f] = "on" :leadsto: Light[f] = "off"
;
```

`TAB` `tla-indent-line` will indent the current line.

LFD `reindent-then-newline-and-indent` indents the current line, and then inserts a new line (much like `RET`) and move the cursor to an appropriate position at the left margin.

tla-indent User Option
Number of spaces that make up the indentation of an unnested line.

tla-inner-indent User Option
Number of spaces to add to the indentation of *tla-indent*.

tla-auto-untabify t Variable
If non-nil, current buffer will be untabified upon saving.

tla-bullet-lorand t Variable
If non-nil, lists of conjunctions and disjunctions will be "bulleted" wrt to nested structures.

1.4 Outlining

All GNU Emacs versions prior to version 19.19 come with an outline mode with only limited functionality. A port of the current outline mode to Emacs 18 and Lucid Emacs is available from '`ftp.iesd.auc.dk:/pub/emacs-lisp/outln-18.el`'.

TLA+ mode supports the standard outline minor mode for outlining symbol definitions. (See section "Outline Mode" in *GNU Emacs Manual*.) Normally, `outline-minor-mode` will use the prefix key `C-c`, which is also used by TLA mode, so it might be better to choose another prefix key by inserting

```
(setq outline-minor-mode-prefix "\C-c\C-o") ; Or whatever...
```

in your '`~/.emacs`' file. There is a menu item "Toggle Outlining" turning the minor mode on or off.

Outlining a document means to impose a hierarchical order on it. Since there is no apparent hierarchical structure in specification code, the current version defines just two levels: the first one matching keywords and the second one matching symbol definitions. This suffices to get an overview of the current code and the regular expressions needed are not too complex to be noticeably slow.

For instance, when point is on a first-level heading (i.e. a keyword) you could hide the entire structure up to the next heading by hiding the subtree (`C-c C-o C-d`). If you then show branches by typing `C-c C-o C-k`, all headings of the next, i.e. second level are visible again but their contents is hidden:

```
ACTIONS

  N == :E: A :in: ATM...

  ReceiveRequest(A: ATM) == ...
```

```
DoRequest(A: ATM) ==...
            r == pending[A].req;
            a == pending[A].amt;...
Next == \/ CheckOut...
```

However, as can be seen in this example, there is a problem with nested, multiline symbol definitions introduced by LET. The second and following lines of a LET definition look absolutely perfect to the outline mode in the sense that they are matched as a second level heading (since the symbol definitions in LET are regular symbol definitions). There seems to be no apparent solution for this problem if one sticks to the regular outline mode.

1.5 Using Multiple Fonts

In Emacs 19 it is possible to use different fonts and colors in order to highlight certain regions of a document. While TLA+ mode doesn't support color it makes use of different fonts, also called faces. This is invoked automatically when the value of `tla-use-font-lock` is non-nil. The use of different faces is controlled by the font lock minor mode. If you want to toggle this mode, simply type `M-x font-lock-mode`.

In the current version the use of faces is restricted to comments and strings. Comments will be italicized and strings will be underlined. There is no need to emphasize keywords since they are already uppercase. There is no reason to fontify more than that (e.g. by fontifying symbol definitions) since fontifying large documents gets quite slow.

tla-use-font-lock t Variable

If non-nil then buffer will be automatically fontified when entering TLA mode.

2 Multifile Specifications

Normally, TLA+ specifications are split in several modules. There is, however, a top module whose name is by convention also the name of the whole specification. Therefore, a file 'LibraryActions.t' is not the top module of the Library specification, but 'Library.t' is. By using this top module as a "master file" for the rest of the modules, this relation can be used in several ways, such as issueing a command on the corresponding top module file when another module of the specification is being visited.

TLA+ mode has no inherent means to tell the name of the master file. By default, it will assume that the current file itself is a master file. If you insert the following in your '~/.emacs' file, you will be asked for the name of the top module which is the corresponding master file.

```
(setq-default tla-master nil)     ; query for top module
```

To avoid asking you again, this information will then be inserted at the very end of the module as a file variable. (See section "File Variables" in *GNU Emacs Manual*.)

```
(* Local Variables: *)
(* tla-master: "Library" *)
(* End: *)
```

Besides the convention that the name of the top module is also the name of the complete specification there is another convention requesting all modules belonging to the same specification to be in the same directory. Some existing tools depend on this restriction.

tla-master User Option

The master file, i.e. the top level module, associated with the current buffer. If the file being edited is actually imported or included from another file, you can tell TLA+ mode the name of the master file by setting this variable. If there are multiple levels of nesting, specify the top level file.

If this variable is `nil`, TLA+ mode will query you for the name.

If the variable is `t`, TLA+ mode will assume the file is a master file itself.

If the variable is `shared`, TLA+ mode will query for the name, but will not change the file.

It is suggested that you use the File Variables (see section "File Variables" in *The Emacs Editor*) to set this variable permanently for each file.

tla-save-specification Command

(`C-c C-d`) Save all buffers known to belong to the current specification.

tla-save-query t Variable

If non-nil, ask user for permission when saving a buffer.

3 Running Commands

Since there are a number of tools involved in the specification process, being able to run them is essential for the quality of the TLA+ working environment.

The commands that are available at your site depend on the configuration of `tla-command-list`. These are the standard commands:

Name	*Command*
`Browse`	Launch "tTLA+Browser" with the name of the current top module or buffer as argument.
`Check`	Run tpp, the preprocessor of the "tTLA+Browser", with the name of the current top module or buffer as argument.
`LaTeX`	Run LaTeX in nonstop mode together with the '`spec92.sty`' style file on the current buffer.

tla-command-list User Option

List of commands to execute on the current module or specification.

Each element is a list, whose first element is the name of the command as it will be presented to the user. The second element is the string handed to the shell after being expanded. The expansion accounts for different types of files. The third element is the function which actually start the process. Several such hooks has been defined which are explained in more detail in the source code.

3.1 Executing Commands

Analyzing the specification with the help of tpp, launching applications such as TBR, or documenting the current module by using Lamport's '`spec92.sty`' all require running an external command. Currently, you can only issue commands involving the whole specification with `tla-command-master`.

tla-command-master Command

(`C-c C-c`) Query the user for a command, and run it on the master file (top level module) associated with the current buffer. The name of the master file is controlled by the variable `tla-master`. The available commands are controlled by the variable `tla-command-list`.

tla-command-buffer Command

(`C-c C-b`) Query the user for a command, and run it on the current buffer. The available commands are controlled by the variable `tla-command-list`.

tla-command-default nil User Option

The default command to run in this buffer. Must be an entry in `tla-command-list`.

3.2 Controlling Output

The following commands allow the user to control the output of a command. They are also accessible via the “Command Output” menu.

tla-kill-job Command

(`C-c C-k`) Kill external command currently running. This may be either a command of tpp, TBR, TEX etc.

tla-recenter-output-buffer Command

(`C-c C-l`) Recenter the output buffer so that the bottom line is visible.

tla-home-buffer Command

(`C-c ^`) Go to the top module of the specification associated with the current buffer, or if already there, to the file where the current process was started.

When ‘debugging“ specifications with the help of tpp, you may browse through the errors. While it is possible to offer such support for other tools as well, this has been realized so far only for tpp.

tla-next-error Command

(`` C-c ` ``) Go to the next error reported by tpp (invoked by “Check”). The view will be split into two, with the cursor placed as close as possible to the error in the top view.

4 Different Variants of TLA+

Having several different variants of TLA+ means having to deal with several classes of tools and the varying language scope of each of them. The concept of a variant tries to cope with the difficulties resulting from this situation. A variant determines which specific features of the language is defined. For instance, eTLA+ does not allow typed identifiers in contrast to tTLA+. This yields a different behavior of the mode.

However, support for different variants is not completed yet. For now, the predominant variant is `"tTLA+"` and most routines still expect the code to behave like tTLA+ code, i.e. according to the rules in (Mester 93).

tla-variant-list User Option

List containing the different variants of TLA+ that are supported. Each element is a list whose first element is the name of the variant that will also be the entry in the menu bar.

The following elements will each indicate presence or absence of a particular language features: The second element denotes typing. If non-nil, typing is mandatory and thus identifiers followed by bracketed expressions will signal an error. If nil, typing is optional and untyped identifiers are allowed. The third element denotes the use of the END keyword. If non-nil, insertion of a header automatically takes care of a final END keyword. The fourth element denotes mixfix operators. If non-nil, mixfix operators are allowed. The fifth element denotes postfix operators. If non-nil, postfix operators are allowed.

tla-current-variant Variable

Indicates current variant of TLA+. Must be an element of `tla-variant-list`.

tla-variant-default "tTLA+" User Option

Indicates default variant of TLA+. Must be an element of `tla-variant-list`.

5 Error Messages

The indentation of spec code depends to a large extent on a correct usage of TLA+ constructs. If some conditions are violated or not respected, there are generally two ways to proceed: make a good guess about the proper indentation relying on other contextual information or inform the user that there is an error-prone condition violating syntactic rules. The TLA mode pursues the strategy of signalling error messages to the user that clearly indicate a wrong structure. A TLA error message generally consists of the string '`TLA Error:`' and a short message explaining the failure of the current operation.

message *explanation*

`Missing LHS of formula or definition`
: The current line includes a semicolon supposed to conclude a structure which is obligatory begun with an identifier followed by '`==`'. That identifier—making up the left-hand side of the current structure—is missing.

`No corresponding LET found`
: The current line contains an '`IN`' statement that requires a preceding '`LET`' in the same formula or symbol definition. That '`LET`' statement is missing.

`Not within symbol definition`
: A command has been issued that involves a whole symbol definition such as commenting it, but point is not inside a symbol definition.

`Corresponding opener of grouping missing`
: The current line contains a '`}`', '`)`' or '`]`' that has no preceding opening counterpart.

`Missing THEN branch`
: The current line contains the '`ELSE`' branch but there is no preceding '`THEN`' branch.

`INCLUDE keyword missing`
: The current line contains the '`WITH`' keyword but there is no preceding '`INCLUDE`' keyword.

`Template not allowed in CONSTANT module`
: An attempt has been made to insert a keyword reserved for '`TEMPORAL`' modules into a '`CONSTANT`' module.

Additionally, this warning can be issued:

message *explanation*

`Template` *Template* `already used`
: A keyword has been inserted that was already present in the current buffer.

6 Installation and Customization

Installation is accomplished with the help of Autoconf. Unpack the original compressed distribution file '`tla-mode-0.4.tar.gz`' with GNU zip. If you also have GNU tar (version 1.11.1 or later), you could do the following:

```
gnutar xvzpf tla-mode-0.4.tar.gz
```

Otherwise, you could use:

```
gunzip -c tla-mode-0.4.tar.gz | tar xvf -
```

1. Type '`./configure`'. This will produce the Makefile including system-dependent variables and functions. For more details refer to the '`INSTALL`' file in the distribution directory.
2. If there is more than one flavor of Emacs on your system, you should check the Makefile and set the value of the variable `EMACS` accordingly. Type `make`. This produces the utilities needed for the documentation.
3. You probably need to change `tla-command-list` to make sure that the various tools at your site can be invoked from within Emacs. This is also the right place to account for the support of different variants. The standard installation shows how to do that.
4. Lisp code will normally go into '`/usr/local/lib/emacs/site-lisp`'. Change the Makefile if necessary. Type `make install` to byte-compile the Lisp files. Lisp and info files will then be installed in their respective directories. Update the '`dir`' in the info directory (normally '`/usr/local/info`') with an entry like this:
   ```
   * TLA+:      (tla-mode).
   Major mode for editing TLA-based specifications
   ```
5. If you use an unusual directory for the compiled Lisp code, users should enhance their *load-path* by modifying '`~/.emacs`':
   ```
   (setq load-path (append load-path (list "/dir/with/lisp-code")))
   ```
   In order to let Emacs automatically switch to tla-mode when a file with appropriate extension (`.t` or `.t+`) is visited, users should put the following code into '`~/.emacs`':
   ```
   (require 'tla-mode)

   (setq auto-mode-alist
         (append '(("\\.t$" . tla-mode))
    auto-mode-alist))
   ```

This completes the installation. Please report encountered bugs.

Key Index

C

L

M

T

Concept Index

V

W

Table of Contents