TChecker - VSCode Extension

Guilhem Ardouin

Contents

1 Introduction **2**

2 Syntax highlighting **2**
- 2.1 Overview . . . 2
- 2.2 TextMate grammars . . . 3
- 2.3 Token categories . . . 4
  - 2.3.1 Meta's scopes . . . 4
  - 2.3.2 Declaration scope . . . 4
  - 2.3.3 Attributes scope . . . 4
  - 2.3.4 Other tokens . . . 5
- 2.4 Theming . . . 5

3 Programming support features **5**
- 3.1 Code Completion . . . 6
  - 3.1.1 Collecting variables . . . 6
  - 3.1.2 Checking for the right context to provide auto-completion . . . 6
- 3.2 Signature Help . . . 7
- 3.3 Hover . . . 7

4 TChecker tools **7**
- 4.1 Configuration settings . . . 7
- 4.2 `tck-syntax` . . . 8
  - 4.2.1 User utilisation . . . 8
  - 4.2.2 Implementation details . . . 8
- 4.3 `tck-reach` and `tck-reach` . . . 9
  - 4.3.1 User utilisation . . . 9
  - 4.3.2 Implementation details . . . 9
- 4.4 `tck-simulate` . . . 10
  - 4.4.1 User utilisation . . . 10
  - 4.4.2 Implementation details . . . 10

5 Conclusion **11**

LABORATOIRE
BORDELAIS
DE RECHERCHE
EN INFORMATIQUE
LaBRI

1 Introduction

TChecker is a verification tool for timed automata developed at the LaBRI by Frédéric Herbreteau and Gérald Point. TChecker provides a range of tools that enable models to be syntactically validated, simulated and formally verified.

Having a development environment for real-time systems would make it easier to use TChecker's various tools. Using TChecker implies writing or having a written model in a code editor. Thus, we choose to develop this environment in a Visual Studio Code extension

During my internship in the *Méthodes et Modèles Formels* (Formal Methods and Models) department at LaBRI, I contribute to the development of this Visual Studio Code extension. More particularly:

- syntax highlighting for TChecker's model;
- edition assistance: auto-completion, signature help, hover;
- implementing TChecker's tools in the VSC environment.

This document presents all the features available in the TChecker VSCode extension and how they were designed. Firstly, the section 2 provides an explanation about how syntax highlighting works. Then, section 3 focuses on programming support features, such as code completion or signature help. Section 4 is about the implementation of TChecker tools. At last, a conclusion of this work is provided in the section 5.

2 Syntax highlighting

Editing a TChecker model is comparable to editing any text document, as its syntax isn't recognized by any editor. Thus, providing syntax highlighting in the Visual Studio Code extension could make the code easier to read and edit.

The following sections explain how syntax highlighting is designed and implemented for the VSC extension.

2.1 Overview

There are two steps for providing syntax highlighting in a Visual Studio Code extension: first tokenization and then theming. Tokenization is allowed by the elaboration of a TextMate grammar file and is detailed in sections 2.2 and 2.3. Theming is discussed in section 2.4.

But first, we will see an overview of the needed files for providing syntax highlighting. As far as syntax highlighting is concerned, we will focus only on the files represented on figure 1.

The `package.json` file contains configuration settings for the entire extension. Concerning syntax highlighting, we are only interested in the *languages* field and the *grammars* field.
The former yields all information about the supported language, such as the file extension (`.tck` here). It also specifies the language configuration file.

```
tchecker-vscode
├── syntaxes/
│   └── tchecker.tmLanguages.json
├── language-configuration.json
└── package.json
```

Figure 1: Files concerned by the syntax highlighting feature

The latter gives us all the information about the language grammar (more details on section 2.2), including the grammar file path.

The `language-configuration.json` file contains all specificities of the language, such as the comment symbol or which characters represent brackets in the language.

The `syntaxes` folder contains the grammar file.

2.2 TextMate grammars

VS Code uses TextMate grammars as the syntax tokenization engine. In this extension, the TextMate grammar is written as a JSON file. The language grammar is used only to parse the document and break it into tokens. Each token will have a category assigned. This name allows VS Code to assign a color to the different tokens.

TextMate grammars rely on Oniguruma regular expressions[1] and therefore the whole tokenization process is based on these expressions.

The whole TextMate grammar for TChecker is written in the `tchecker.tmLanguage.json` file. The following paragraphs will describe how it had been organized.

Each token is associated with a scope that defines the context of the token. Thus, the grammar is structured with different scopes. A description of these scopes is given in section 2.3.
Patterns determines the different tokens. We can specify for each pattern a matching regular expression. Each pattern can be named, and therefore they can be colorized according to the naming conventions[2].

Redundant patterns are stored in the *repository* dictionary. Thus, they can be reused anywhere in other patterns.

For more understanding of how the TChecker grammar has been built, the scope inspector tool is helpful. This tool is available by executing the following command in VS Code command palette: `Developer: Inspect Editor Tokens and Scopes`. The scope inspector reveals all the scopes assigned to a token by hovering it.

[1]https://macromates.com/manual/en/regular_expressions
[2]https://macromates.com/manual/en/language_grammars#naming-conventions

2.3 Token categories

This section aims to describe the different categories used in the extension grammar. As previously stated, each token belong to scopes. The global scope, which includes the whole input file, is called by convention `source.tck`.

As a consequence of TChecker inline formulation, some tokens belong to different scopes. However, it is the most specific scope that determines the category of the token, and therefore its coloration.

For instance, the attribute category is contained in the declaration category. Thus, an attribute has a *declaration* parent scope and an *attribute* scope. Let the following location declaration: `location:P:l0{initial::do:y=0}`. The tokennization process yields:

$$\underbrace{\underbrace{\overbrace{\texttt{location}}^{keyword}\ \overbrace{\texttt{:}}^{separator}\ \overbrace{\texttt{P}}^{parameter}\ \overbrace{\texttt{:}}^{separator}\ \overbrace{\texttt{l0}}^{parameter}\ \underbrace{\{\overbrace{\texttt{initial:}}^{tag}\ \overbrace{\texttt{:}}^{separator}\ \overbrace{\texttt{do:}}^{tag}\ \texttt{y=0}\}}_{\text{meta.attributes}}}_{\text{meta.declaration}}}_{\text{source}}$$

In the following subsections, the different scopes and categories will be explained more thoroughly.

2.3.1 Meta's scopes

In order to refer to a larger part of the document, the TChecker grammar defined large scopes. These larger scopes are designated as "`meta`" scopes, and they contain either another "`meta`" scope or tokens.

If a line begins with a keyword (see section 2.3.2), the line is encompassed with a declaration scope. If in a declaration there are two curly brackets (`{}`) the content has an attribute scope.

Table 1 summarized the different scopes of the grammar.

Scope name	Scope pattern
source.tck	the whole document
meta.declaration.tck	all lines beginning with a declarative keyword
meta.attributes.tck	all characters between `{` and `}` in a declaration

Table 1: Grammar's scopes

2.3.2 Declaration scope

All declarations begin with a keyword. In this grammar, keywords are: `clock`, `edge`, `event`, `int`, `location`, `process`, `sync` and `system`.

Table 2 summarized the different token in a declaration scope.

2.3.3 Attributes scope

Attributes are part of a declaration scope. They are delimited by curly brackets (`{}`) and are composed of pairs `key:value`.

Table 3 summarized the different token in attributes scope.

Token name	Token pattern
keyword.other.declaration.tck	the line begin with a keyword
variable.parameter.tck	after a `:` separator
meta.attributes.tck	all characters bewteen `{` and `}` in a declaration
keyword.operator.at.tck	`@` characters

Table 2: Tokens in a declaration scope

Token name	Token pattern
punctuation.section.attributes.begin.bracket.curly.tck	`{`
punctuation.section.attributes.end.bracket.curly.tck	`}`
entity.name.tag.attribute.tck	following a `{` or a `:` separator

Table 3: Tokens in attributes scope

2.3.4 Other tokens

There are other tokens which have not been discussed yet. These tokens are not assigned to a particular scope and can therefore be found anywhere in a document.

Table 4 summarized the other tokens.

Token name	Token pattern
comment.line.number-sign	begin with a `#` characters until the end of the line
punctuation.separator.colon.tck	`:` characters

Table 4: Other tokens

2.4 Theming

All previous subsections described how the tokenization process work. In this part, we will focus on the theming process.

The theming is handled by Visual Studio Code: the editor interprets tokens by their scope name. As a result, the syntax highlighting theme for TChecker input format is adapted for every VSC theme, whereas it's the theme that defines tokens colors.

3 Programming support features

The larger the model, the most difficult it is to keep track of all the information it contains. Certain tools, such as code completion or getting signature from declarations, can be implemented in order to facilitate the TChecker user to write his model.

The VSC extension provides different programming support tools. These tools will be presented in the following sections.

3.1 Code Completion

TChecker declarations are based on previously defined variables. Sometimes, when the model contains a large number of variables, it might be useful to have an automatic suggestion for all correct variables when writing a declaration.

Providing auto-completion for TChecker declarations requires focusing on each declaration individually. First, we need to distinguish between declarations that don't require auto-completion and those that do. For instance, `system` and `process` declarations don't require auto-completion, as they don't depend on variables already declared.

Only three kinds of declarations can benefit from auto-completion: `location`, `edge` and `sync`. We will also provide auto-completion for all keywords.

For all concerned declarations, the procedure for providing auto-completion is the same: collect all the potential variables and check whether it's the right context to suggest these variables.

We will focus on these two steps more thoroughly in the following subsections.

3.1.1 Collecting variables

For each declaration kind, we know which variables interest us. For instance, for declaring a `location`, we only need to collect all `process` variables.

At some point in the document, when it comes to declaring a `location`, we will collect all `process` variables defined above the user's current position (indeed, `process` defined below can't be used). To do this, we will match all lines above the current declaration with the string *process*. Then, all these lines will be parsed as follows.

In TChecker, the separator between variables is `:`. Thus, we will split each line using this separator. This operation yields an array of strings: `[process, P]` for a process named "P" for instance. Then, we know for each declaration what is the expected position for each variable. In our example, we know that the process name is the "second element". So, we will get all second elements of the collected array and we will have collected all the variables.

Please note that, in some cases, additional constraint may be necessary. For instance, some variables are associated with others, such as a location associated with a specific process. We therefore take this into account in the parsing for collecting variables.

3.1.2 Checking for the right context to provide auto-completion

Identifying what is the right context to suggest these variables will be the same principle as collecting them. First, we identify when the expected keyword is used. We know that, for triggering the auto-completion, we need the user to type `:`. So, we will only trigger the auto-completion when the user triggers the `:` at the right place on a declaration. Thus, we will specify for each parameter of a declaration the expected position.

In the case of synchronization, it's possible to write more than one declaration. We allow retriggering which basically do the same thing and don't really check the position, only the coherence for the correct amount of `:` and `@`.

3.2 Signature Help

Some users may forget about the order of arguments in a declaration. Instead of going back and forth between the code editor and the TChecker documentation, signature help could represent an alternative.

Signature help can be provided based on the VSC API. Thus, it's necessary to write all documentation associated to all declarations (signature, description...). All this information is available in the file `constants.ts` and are easily editable.

Now, for the implementation, we based the triggering of the signature help by typing keywords. Providing a certain information relies on the current declaration and the position of the user in it. We defined an `attributePos` for each keyword in order to tell VSC when attributes should come, and therefore stop triggering the signature help. The active parameter is known by looking at the number of the colon in the declaration.

3.3 Hover

All information provided by the signature help tool, explained in section 3.2, can be useful to the user. That's why it could be interesting to provide this information to the user only by hovering the model's text.

This feature is implemented by using the VSC API and mapping all collected information for the signature help on different keywords. Thus, the user will get this information by hovering a keyword.

4 TChecker tools

TChecker provides various tools for formal verification. Formerly, all tools had to be used in a terminal. Implementing these tools in the Visual Studio Code environment could make their utilization faster and more interactive for the user.

The following sections describe how these tools were incorporated in the Visual Studio Code extension and how to use them.

4.1 Configuration settings

Using TChecker's tools requires building TChecker's executables. Then, assuming TChecker has been successfuly built, it's successfully to specify the build path to Visual Studio Code. It can be done in VSC settings, searching for "TChecker" or going to extensions settings.

4.2 `tck-syntax`

One feature of TChecker is the syntax checking of a model. The `tck-syntax` tool performs a syntax analysis of a model given as an input. In the case of an incorrect syntax, the tool provides us all information concerning eventual errors and warnings.

Visual Studio Code offers a “Problems” panel where possible errors can be written. The idea for the extension was to implement a button in the VSC status bar that launches the syntax check and informs the user whether or not his model is correct. In the case of errors or warnings, the extension will provide to user all visual information concerning these errors.

4.2.1 User utilisation

Performing syntax checking can be performed in two ways:

- running the command `TChecker: Syntax check` in the Command Palette;
- triggering the status bar button “Check Syntax” (which executes the same command as above).

After executing this command, a pop-up message indicates whether the syntax of the current file is correct, or whether there are any errors or warnings. More precise insights, on potential errors or warnings, are available on the “Problems” panel and highlighted in the code.

`tck-syntax` has different options. It’s possible to change the associated command “TChecker: Syntax check” in VSC settings. Please note that default command is `tck-syntax -c`.

4.2.2 Implementation details

The command `TChecker: Syntax check` runs a child process with the associated command “TChecker: Syntax check” in VSC settings. Then, it parses the output of the child process. This output is a collection of information about the child process, such as its exit signal, *stdout*, *stderr*...

Knowing if the syntax is correct or not depends on the exist signal of the child process. The parsed output contains this information, the program distinguishes two cases:

- the exit signal is different from 0: the syntax is incorrect (error);
- the exit signal is 0: there is no error, but we have to check for warnings.

In the latter case, we can acknowledge the eventual presence of warnings by looking the *stderr* channel. If warnings or errors are detected, the program will parse the different messages. Elsewise, the program only displays a message which indicates that the model is syntactically correct.

Parsing errors messages consist of getting all important information, such as the position of an error or the cause. Then, the program creates VSC diagnostics and send them to the editor. VSC reads the diagnostics and outputs it in the “Problems” window and highlights the parsed zones in the code.

Let's take an example of an error message:

```
ERROR: 16.10 process P is not declared
```

This error indicates that an illegal declaration with an undeclared process has been done in the line 16, column 10. The parsing will identify the position (e.g. line 16, column 10) of the error. To do so, a regex has been used for matching the position.

Please note, other error or warning can yield more information about position, such as `WARNING: 17.15-17 unknown attribute test`. In this example, we have the information about the column begin and the column end of the warning.

A further comment: when this feature was implemented, TChecker's syntax checker didn't raise errors and warnings at the same time. In the event of an evolution where both can be fund in an output, the feature would be deprecated. An area for improvement would be to distinguish errors and warnings not by the program exit signal but by the associated message.

4.3 `tck-reach` and `tck-reach`

TChecker provides two verification tools that can be used to assess that a model satisfies its specification. The extension runs these tools through Visual Studio Code tasks[3].

4.3.1 User utilisation

Performing a verification of reachability or liveness requires the user to write his own tasks. We will provide in this section an example of a VSC task. To help the user use their tasks, we provided a shortcut button to the task manager on the status bar.

If it isn't already done, the user has to create a `tasks.json` in his `.vscode` file. Then, he can fill his file as presents in the Figure 2.

Please note that the expression `TCKBUILDPATH` must be replaced by the user's TChecker build path.

4.3.2 Implementation details

As explained in section 4.3.1, writing a VSC depends on the user. Thus, the implementation of this feature is not cumbersome. What's important for the user is to have a shortcut to access his written tasks. We have therefore provided this shortcut as a button in the VSC status bar.

For the future, it might be interesting to design a custom file `tchecker.json` in the `.vscode` folder. It would then be necessary to implement the tool for selecting new custom tasks. Using the "'QuickPick" functionality of the VSC API[4] could represent an option to do this.

[3]https://code.visualstudio.com/docs/editor/tasks

[4]https://code.visualstudio.com/api/references/vscode-api#QuickPick

```json
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "tck-reach with reach algorithm",
      "detail": "here some details",
      "type": "shell",
      "command": "TCKBUILDPATH/tck-reach -a reach ${file}",
      "presentation": {
        "reveal": "always",
        "panel": "new"
      }
    },
    {
      "label": "tck-liveness with couvscc algorithm",
      "type": "shell",
      "command": "TCKBUILDPATH/tck-liveness -a couvscc ${file}",
      "presentation": {
        "reveal": "always",
        "panel": "new"
       }
    }
  ]
}
```

Figure 2: An example of a `tasks.json` file

4.4 `tck-simulate`

TChecker provides a tool for interactive simulation of the model. It's possible to use the Visual Studio Code output box to display the simulation output, and an input box for the user to interact with the simulation.

4.4.1 User utilisation

To simulate the model, the user has a button on the status bar to launch the simulation for the currently edited file. The simulation then starts and waits for the user's output.

The user can interact with the simulation with an input box. It may happen that the input box disappears, that's why the user can show it back by clicking on the red button on the status bar.

Please note that a user can't launch more than one simulation at a time, and therefore has to end a simulation before beginning another.

4.4.2 Implementation details

The tool `tck-simulate` launches a process that requires user interaction. To provide this feature in VSC, an asynchronous child process is used. As mentioned previously, the user can interact with the sub-process using an input box. The sub-process receives the user's input and VSC, which is listening to the child process channel *stdin*, writes the output in an output box.

This procedure is illustrated in Figure 3.

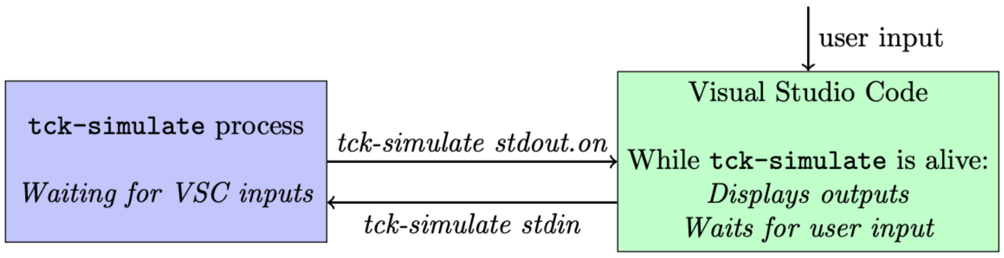

Figure 3: `tck-simulate` processes

More thoroughly, while the child process is alive, VSC requests to the user an input. In order to check if the child process is still alive, VSC refresh his request every half second. When the child process died, VSC runs a closing routine in order to clear the environment.

At the time this feature was conceived, the `tck-simulate` tool didn't provide a JSON output or a step by step simulation mode. This is now the case. For the future, it may possible to revise the current implementation of this feature by using the JSON output to provide a customized output window.

As the JSON format provides every needed information, it's possible to integrate the new output window with VSC's Views[5].

Moreover, it would be possible to parse this information and highlight important declarations in the code (such as location). To do this, it could be possible to rely on VSC API decorations[6].

5 Conclusion

The aim of this internship was to conceive a Visual Studio Code extension for TChecker. Throughout this document, the features available in this extension have been presented.

The extension provides syntax highlighting support and offers features to enhance the user experience, such as auto-completion, signature help and hover information.

Furthermore, the extension implements the different TChecker tools. Model's syntax checking, verifications and simulations are available in the Visual Studio Code environment.

Lastly, ideas for improving the future of the extension were explored.

[5]https://code.visualstudio.com/api/ux-guidelines/views
[6]https://code.visualstudio.com/api/references/vscode-api#TextEditorDecorationType