CoreTML framework 1.4

Tutorial

2010-2015

http://coretml.sourceforge.net

1. Introduction

CoreTML is an open-source framework allowing the developer to create parametrized templates which can be transformed to different output files based on template parameters chosen by the user. CoreTML uses a specially designed language, *TML* (*Template Markup Language*), which comes in two flavors: *TML-g* for parametrized templates themselves and *TML-c* that is used to describe graphical user interfaces allowing the user to set template parameters.

TML-g also makes extensive use of the Lua programming language (see http://www.lua.org).

TMLgen is a cross-platform console program for TML-g file processing. It is a core of the CoreTML system.

TMLconf is a Windows-only program displaying graphical user interfaces based on TML-c descriptions. It is currently not very flexible and serves mainly demonstration purposes.

CoreTML Editor is an editor supporting TML syntax highlighting and a few other TML-specific features. It is essentially a modified version of SciTE (see http://www.scintilla.org).

This tutorial covers only basic CoreTML features. Additional information is available in the other documents:

- *Template Markup Language Reference* is a detailed description of TML-g and TML-c languages.
- *CoreTML software manual* describes CoreTML tools in detail.

2. Hello, world!

TML-g works by adding control sequences to the otherwise transparent text file. A text file without control sequences can be a trivial case of a TML-g template. Let's create the following template (*example.tml*):

example.tml

```
Hello, world!
```

To process the template one can either invoke TMLgen from the command line, or use the CoreTML Editor.

Using TMLgen from the command line

To run TMLgen for the following example, execute:

```
tmlgen example.tml output.txt
```

This command tells TMLgen to process *example.tml* template and place output to the *output.txt*.

You can also omit the second filename, in that case output will be written to the standard output stream (i.e. the console window):

```
tmlgen example.tml
TMLgen 1.4 (c) 2010-2015 by Alex I. Kuznetsov
Hello, world!
```

Using the CoreTML Editor

In the CoreTML Editor, create a document with the above content, then save it with the *tml* extension. Now you can use *Run→Go* menu command, press 'F5' or click on a corresponding toolbar button. The TMLgen output will appear in the output window:

```
>tmlgen "example.tml"
TMLgen 1.4 (c) 2010-2015 by Alex I. Kuznetsov
Hello, world!>Exit code: 0
```

3. Adding control sequences

Now let's modify our *example.tml*:

example2.tml

```
Hello, world at \getinfo{time}!
```

After processing the above TML-g template, TMLgen should produce something like this:

```
Hello, world at Fri Oct 15 02:23:49 2010!
```

\getinfo is one of the TML-g tags. TML-g language defines about two dozen tags which are described in the “Template Markup Language Reference” in detail. Each tag starts with a backslash, has a *tag identifier* (“getinfo” in this case) and *tag content* surrounded by braces (“time” in this case, you can also try to replace “time” with “software”, “version” or “platform”).

As backslash and braces have special meaning in TML, one have to precede them with a backslash to insert them literally. As a special exception, it is possible to omit that backslash before an opening brace ('{'):

example3.tml

```
Special characters demonstration:
\\ - backslash
{ - opening brace
\{ - another opening brace
\} - closing brace
```

TMLgen will then produce the following file as output:

```
Special characters demonstration:
\ - backslash
{ - opening brace
{ - another opening brace
} - closing brace
```

In TML-g, you can use nested tags whenever you want. For example:

```
\if{\getinfo{version}>=1.1}
\then{TMLgen version is supported}
\else{Warning: TMLgen version is unsupported}
```

TML-g supports a \comment tag. Its content is ignored by a parser:

```
\comment{
     this line will be ignored
     \echo{this tag will also be ignored}
}
```

4. Parametrized templates

It was shown in the above examples that it is possible to create non-parametrized TML-g templates that are perfectly valid. However, in order to make any real sense the template should depend upon some parameters which are passed by the user.

Imagine that we want to create a CSS table template for a website (note a backslash preceding the final '}'):

example4.tml

```
body {
	font-size:\eval{size}pt;
	color:\eval{color};
\}
```

\eval tag is used to evaluate Lua expressions. Assuming that we want to set *size* to 9 and *color* to “red”, we can invoke:

```
tmlgen --set "size=9;color='red'" example4.tml out.css
```

This will create *out.css* with the following content:

out.css

```
body {
	font-size:9pt;
	color:red;
}
```

Passing template parameters as TMLgen command line arguments is not very convenient and sometimes even can be impossible. In the next chapter you will learn about a more generic way to pass template parameters (using \create tag).

Although the above example is a correct TML-g template, it is recommended that every template file declares its input parameters with the help of \parameters tag. Let's modify our example:

example5.tml

```
\parameters{
	\req{name="size";type="number"}
	\opt{name="color";type="string";default="black"}
}body {
	font-size:\eval{size};
	color:\eval{color};
\}
```

Two parameters are declared in the above example: *size* with type *number* is a mandatory parameter (defined by \req tag), *color* with type *string* is an optional parameter (defined by \opt tag) which defaults to *black*.

Using \parameters tag declare template parameters can useful in a number of ways: mandatory parameters are automatically checked for presence, optional parameters can be assigned default values and, the last but not the least, this can be helpful to a person who is supposed to use the template.

Let's now process our modified example.

Using TMLgen from the command line

```
tmlgen --set size=9 example5.tml out.css
```

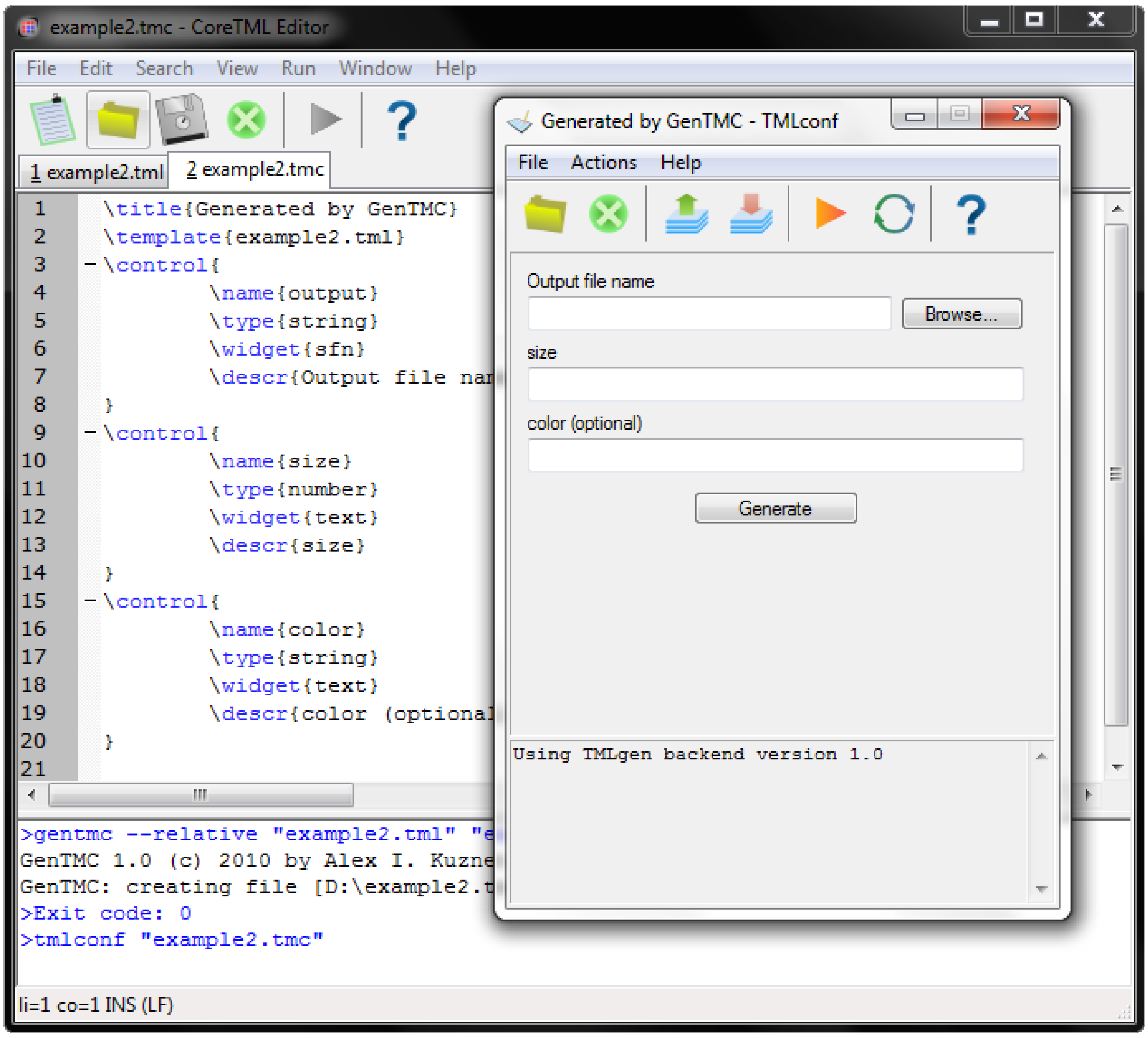
Note that we didn't specify the *color* parameter at this time since it's optional. The result should be:

out.css

```
body {
     font-size:9;
     color:black;
}
```

Using the CoreTML Editor

In the CoreTML Editor, save the above template with as a file with *tml* extension and invoke *Run→Generate configuration interface description* menu command. A file with *tmc* extension should be created. After issuing *Run→Go* for this *tmc* file the TMLconf window should appear:



```
\title{Generated by GenTMC}
\template{example2.tml}
\control{
        \name{output}
        \type{string}
        \widget{sfn}
        \descr{Output file nar
}
\control{
        \name{size}
        \type{number}
        \widget{text}
        \descr{size}
}
\control{
        \name{color}
        \type{string}
        \widget{text}
        \descr{color (optional
}
```

```
>gentmc --relative "example2.tml" "e
GenTMC 1.0 (c) 2010 by Alex I. Kuzne
GenTMC: creating file [D:\example2.t
>Exit code: 0
>tmlconf "example2.tmc"
```

```
Using TMLgen backend version 1.0
```

Now you can choose output file name and set template parameters as you wish. After pressing *Generate* the generated file will be also opened in the CoreTML Editor.

Note that the configuration interface description (*tmc* file) was generated based solely on \parameters tag content.

5. Template invocation

The most generic way to invoke a template is to refer to it from another template using the \create tag.

To invoke *example5.tml* from the previous chapter we can use something like this:

invoke5.tml
```
\create{
     template="example5.tml"
     output="out.css"
     size=9
     color="red"
}
```

Assuming that we name the above file as *invoke.tml*, we can either run

```
tmlgen invoke5.tml
```

from the command line or open the file in the CoreTML Editor and execute *Run→Go.*

Note that we don't pass any other parameters to TMLgen in this case. *invoke.tml* don't have input parameters and all parameters passed to *example2.tml* are defined in the \create tag. Such a simple TML-g template containing only \create tag and having no input parameters is called an *invocation script*.

Let's look at another example:

example6.tml
```
Hello from \getinfo{inputpath}!
x is \eval{x}.
y is \eval{y}.
\create{
     template="example7.tml"
     output="out2.txt"
     z=x+y
}
```

example7.tml
```
Hello from \getinfo{inputpath}!
z is \eval{z}.
```

invoke6.tml
```
\create{
     template="example6.tml"
     output="out.txt"
     x=7
     y=9
}
```

In this example *invoke6.tml* invokes *example6.tml* writing its content to *out.txt*; *example6.tml* in turn invokes *example7.tml* whose content is written to *out2.txt.* We should get something like this:

```
                                                          out.txt
Hello from D:\example6.tml!
x is 7.
y is 9.
```

```
                                                         out2.txt
Hello from D:\example7.tml!
z is 16.
```

If the *output* argument is omitted, the template output is written to the calling template's output stream. E.g. if we have removed “`output="out2.txt"`” line from the *example6.tml* template we would have got

```
                                                          out.txt
Hello from D:\example6.tml!
x is 7.
y is 9.
Hello from D:\example7.tml!
z is 16.
```

It is also possible to use the \create tag to invoke the so-called *snippets* instead of templates. The main difference between them is that snippets don't have to be deployed in separate files. Snippets will be covered later in this tutorial.

6. Embedding Lua scripts

TML-g language is tightly integrated with Lua scripting language. We have already seen basic examples of scripting since \eval tag in fact evaluates a Lua expression. Moreover, expressions used to initialize template parameters in the \create tag are also managed as Lua expressions.

More versatile scripting can be accomplished by using \script tag that invokes a chunk of Lua code. Note some rules about it:

- all scripts in the template (be it via \script tag, \eval tag or otherwise) are run in one context, therefore Lua chunks can use variables set by previous Lua chunks;
- when \create tag is used to invoke a new template or snippet, that template or snippet gets a new Lua context and can't reference caller's variables except those that were explicitly passed in the time of creation;
- \script tag content still needs to adhere to the TML syntax, i.e. backslashes and closing braces must be preceded by a backslash.

Let's look at the following example:

example8.tml

```
\script{
    print("Invoking Lua script...")
    write("<!-- RGB values calculated by Lua-->")
    rhex=string.format("%.2X",R)
    ghex=string.format("%.2X",G)
    bhex=string.format("%.2X",B)
}
<html>
<head>
</head>
<body>
    <font  color="#\eval{rhex..ghex..bhex}">This  text
will have user-defined color</font>
</body>
</html>
```

print is a standard Lua function used to write something to the standard output (e.g. console). The *"Invoking Lua script..."* string will be displayed on the screen but will not get to the output file. On the contrary, *write* is a non-standard function defined as TML-g Lua extension which is used to write to the output stream of the current template.

This template expects three number parameters (R,G and B), converts them to hexadecimal representation using Lua's *string.format* function using format string similar to that used by C's *printf*. Then those three strings are concatenated using Lua's ".." operator.

invoke8.tml

```
\create{
    template="example8.tml"
    output="out.html"
    R=5
    G=200
```

```
     B=100
}
```

After processing *invoke8.tml* that in turn invokes *example8.tml* with certain parameters we get the following result:

out.html

```
<!-- RGB values calculated by Lua-->
<html>
<head>
</head>
<body>
     <font  color="#05C864">This  text  will  have  user-
defined color</font>
</body>
</html>
```

Besides Lua extensions (such as *write*) defined implemented in TMLgen core, there are also some functions defined in the TMLgen Lua library called *scriptlib.tml*. One of such functions is a *hexstring* function that could have been used in *example8.tml* instead of *string.format*:

example8.tml

```
\include{scriptlib.tml}
\script{
     print("Invoking Lua script...")
     write("<!-- RGB values calculated by Lua-->")
     rhex=hexstring(R,2)
     ghex=hexstring(G,2)
     bhex=hexstring(B,2)
}
<html>
<head>
</head>
<body>
     <font  color="#\eval{rhex..ghex..bhex}">This  text
will have user-defined color</font>
</body>
</html>
```

Note the \include tag.

Lua language itself will not be covered in this tutorial. Refer to the official Lua documentation (http://www.lua.org/docs.html). Full list of TMLgen Lua extensions can be found in the *Template Markup Language Reference*. Functions for the TMLgen Lua library (*scriptlib.tml*) are defined in *scriptlib.lua*.

7. Branches, loops and formatting

TML-g supports branching similar to that of conventional programming languages. It is implemented as a set of tags: \if, \then, \elsif and \else. Let's see a modified CSS example from chapter 4:

example9.tml

```
body {
     font-size:\eval{size}pt;
     color:\eval{color};
     \if{decorate}
     \then{
          font-style: italic;
          text-decoration: underline;
     }
\}
```

invoke9.tml

```
\create{
     template="example9.tml"
     output="out.css"
     size=14
     color="red"
     decorate=true
}
```

After processing *invoke9.tml* we get:

out.css

```
body {
     font-size:14pt;
     color:red;


          font-style: italic;
          text-decoration: underline;

}
```

Note the redundant indentation and empty lines in the output file. It can be near impossible to maintain proper indentation in complex templates while keeping reasonable output file formatting. Luckily, TML-g offers a special type of syntax called *strict formatting* for such cases.

Templates using strict formatting syntax begin from the \format{strict=true} line. When this mode is employed, TMLgen ignores newline characters in the template. Spaces and tabulation characters in the beginning of each line are also ignored. Newline characters, spaces and tabulation characters can be inserted explicitly using the \x tag.

Let's rewrite *example9.tml* using strict formatting:

example9.tml

```
\format{strict=true}
body {\x{n}
     \x{t}font-size:\eval{size}pt;\x{n}
     \x{t}color:\eval{color};\x{n}
     \if{decorate}
     \then{
          \x{t}font-style: italic;\x{n}
          \x{t}text-decoration: underline;\x{n}
     }
\}
```

This modified template generates the following output:

out.css

```
body {
     font-size:14pt;
     color:red;
     font-style: italic;
     text-decoration: underline;
}
```

Most programming languages provide at least two types of loops, commonly called *for* and *while*. Unlike that languages, TML-g has only one generic type of loop, an infinite loop. Initialization, loop variable increment and exiting from loop must be done separately.

example10.tml

```
\format{strict=true}
Print all numbers from 1 to 10 inclusively\x{n}

\script{n=1}
\loop{
     \eval{n}\x{n}
     \breakif{n==10}
     \script{n=n+1}
}
```

The above template results in the following output:

```
Print all numbers from 1 to 10 inclusively
1
2
3
4
5
6
7
8
9
10
```

Expressions inside \if, \elsif and \breakif tags have to be valid Lua expressions. They are evaluated as follows: any non-empty string, non-zero number or boolean *true* counts as true, anything other counts as false. Note that this is different from the standard Lua way to evaluate boolean expressions (Lua considers expression to be false only if it equals to boolean *false* or if it is *nil*, any other values are considered to be true).

8. Snippets and libraries

Besides the template invocation mechanism described in the Chapter 5, TML-g provides two other means of code reuse:

- a special reusable piece of code called *snippet* can be declared and later instantiated.
- one file can be included from another using the \include tag;

Snippets are somewhat similar to templates. They must be instantiated using the \create tag, moreover, they are processed in a separate context (as full-fledged templates do). Unlike templates, snippets don't need to occupy the whole file.

example11.tml

```
\format{strict=true}
\snippet{
     \name{mysnippet}
     \body{
          \parameters{
               \req{name="ch";type="string"}
               \req{name="n";type="number"}
          }
          \script{i=1}
          \loop{
               \breakif{i>n}
               \eval{ch}
               \script{i=i+1}
          }
     }
}
The 'a' character will be repeated 10 times:\x{n}
\create{
     snippet="mysnippet"
     ch="a"
     n=10
}
```

Here, we define a snippet with name *mysnippet* and parameters *ch* and *n* that inserts *ch* character to the output stream *n* times. Note that now we use *snippet* argument of the \create tag (instead of *template*).

```
The 'a' character will be repeated 10 times:
aaaaaaaaaa
```

It can be convenient to place some reusable snippets as well as Lua functions to the dedicated file that can be included by the other templates. You have already seen an example of \include tag use in the Chapter 6.

When one file is included from another, TMLgen interprets it as if included file's contents were literally inserted in place of the \include tag (few minor caveats still apply; for example, strict formatting mode is considered separately).

Let's place our snippet to an external library file along with some Lua function definitions:

library12.tml

```
\format{strict=true}
\snippet{
     \name{mysnippet}
     \body{
          \parameters{
               \req{name="ch";type="string"}
               \req{name="n";type="number"}
          }
          \script{i=1}
          \loop{
               \breakif{i>n}
               \eval{ch}
               \script{i=i+1}
          }
     }
}
\script{
     function f1()
          write("Hello from F1!\\n")
     end

     function f2()
          write("Hello from F2!\\n")
     end
}
```

Including *library12.tml* allows the developer to use snippets, Lua variables and functions defined there:

example12.tml

```
\format{strict=true}
\include{library12.tml}
The 'a' character will be repeated 10 times:\x{n}
\create{
     snippet="mysnippet"
     ch="a"
     n=10
}\x{n}
\script{
     f1()
     f2()
}
```

After processing example12.tml we get the following output:

```
The 'a' character will be repeated 10 times:
aaaaaaaaaa
Hello from F1!
Hello from F2!
```