TTL: Transformational Template Language

ABSTRACT

TTL is a Transformational Template Language designed for developing templates that can transform inputs of different formats into desired outputs. A TTL template contains information about the format of its input data in addition to how its output should look like. TTL templates are also dynamic in nature. This property makes it easy to create and manipulate templates in an easy and flexible manner. Moreover, they can encapsulate computation; hence they are more like self-contained software components than just maps from input to output. TTL is also an extensible language which allows and encourages the creation of new user-defined constructs to meet the constant change in systems' business requirements. The TTL runtime engine runs either as a standalone application or as an embeddable system. TTL can prove very useful in many applications such as content-management and code generation.

Keywords

Template, Transformational templates, Componentized template.

1. INTRODUCTION

Transformational templates, in the context of this paper, denote those templates that transform their given inputs from one form to another. XSL templates are an example of such templates [1]. Typically these templates take input documents in some format along with some external parameters and transform them into desired outputs.

This paper proposes a new transformational language called TTL (Transformational Template Language).

1.1 Motivations

The work on TTL is motivated by the following factors:

- ***Dynamic vs. static***: Changing static templates requires modifying the source code and reloading the modified code for the change to take effect. Dynamic templates, however, have the capability to change at runtime without a need for reloading. Most template languages are static.
- ***Mass input vs. parameters***: Templates usually contain place holders that are filled with data as they get processed. Input data can be provided to templates either as parameters or as mass input. The former way is practical if the amount of input is relatively small (about 10-20 parameters perhaps). It grows unmanageable after that. The latter is more practical when a large amount of input is involved. Allowing for both is definitely an advantage.
- ***Free-formatted input vs. structured input***: Inputs received by templates can be limited to having to be in a certain format as in the case of XSL in which well formatted XML documents are expected as inputs. Templates that allow free-formatted inputs are more flexible and more applicable [1].
- ***Simple structure vs. no structure***: Many template languages such as Velocity impose little structure in their templates [7]. Others such as XSL are very strict in their structures. It is argued in this paper that templates with simple structures are more readable than those that have none. Too much structure, however, leads to inflexibility.
- ***Generic vs. specific***: Many template languages were designed with specific applications in mind. Generic ones are flexible enough though to stand out on their own.
- ***Componentization***: Typically, processing a template requires performing a considerable amount of computation outside the template. Such computation is needed either to provide and/or prepare the input data that will be passed to the template or because the logic could be too complicated to implement within the template. Another rather new approach would be to move computation in and to have templates encapsulate all the computations they need. Thus making them more self-reliant and less contingent on external entities.
- ***Evolvable vs. fixed***: Most template languages are fixed as far as the language constructs are concerned. This could be ok for application-specific languages. For a generic language such as the one proposed by this paper, evolvability must be considered.

1.2 Design Goals

TTL is designed to be:

- ***Easy to use***: It has simple self-describing structure that is easy to understand even for those with little programming experience.
- ***Flexible***: Being generic requires being flexible enough to accommodate most situations.

- ***Extensible***: Although it comes with its own built-in constructs, new constructs can be added easily as needed.
- ***Embeddable***: TTL is a compact template language in the sense that it has a small number of syntactic and semantic rules. This makes it ideal for embedding into other systems. The TTL engine must be able to run as a stand-alone application and as an embedded engine within a bigger system. For that TTL runtime engine has to expose the kind of interface that will make it easier for other systems to communicate with.
- ***Reusable***: Reusability is what most template languages including this one are all about. TTL allows for many forms of reusability such as template reusability and construct reusability.
- ***Dynamic***: TTL templates can change their behavior at runtime.
- ***Modular***: Every template is a module by itself. Templates are organized into libraries which can then be referenced and used from within other templates.
- ***Robust***: When an error occurs in a construct within a template, other parts of the templates can still produce their expected results without being affected by the error that just occurred.

1.3 Applications of TTL

There are many computing areas and fields where TTL can prove to be a very useful tool. The following are few of the many possible applications:

- Web applications where TTL templates can be very handy in making web development easier through externalizing the HTML to templates that are dynamically changeable.
- Content management systems where TTL templates can be used to create and manage the contents statically or dynamically.
- Code generation is another important application of TTL. The TTL engine could be embedded within systems like an IDE for instance and used to generate code for developers.
- System integration is another application for TTL where different software systems that are incompatible with one another need to work together. Embedded TTL can be plugged in between two of such systems and made to work as an adapter/translator to help these systems talk to one another effectively.

2. STRUCTURE OR THE TEMPLATE

TTL templates are designed to be dynamic - able to change on the fly at runtime without having to change their sources and be reloaded. In achieving this, TTL templates are compiled at load-time into tree representations.

2.1 Hello-World Template

To start with, Figure 1 (a) gives a complete TTL template of the famous hello-world example.

```
[c[ Hello-World template example ]]
[template HelloWorld[
   [declare[
      atomic str
   ]]

   [input[
      $str
   ]]

   [output[
         Hello $str!
   ]]
]]
```

(a) The template.

```
World
```

(b) The input.

```
Hello World!
```

(c) The output.

Figure 1. Hello-world TTL template

This template can take any input. It loads the input text into a variable called *str* and then uses it to produce its output. If the input is like Figure 1(b), it produces an output that is like that of Figure 1(c).

2.2 The Template Tree Structure

As mentioned before, TTL templates are represented as trees at runtime. In this representation, constructs represent the nodes of the tree. Leaves are nodes without children. They could be either free-formatted texts, or body-less constructs.

There are many advantages to having a tree representation of a TTL template:

- Trees are very common data structures among developers.
- Trees can be kept in memory at runtime for as long as they are needed to stay there and while they are there, they could be modified by adding, editing, and deleting nodes.
- The tree structure can be very easily converted to a persistent form. One such form would be a normal TTL template source code.
- Using tree structures, templates can be very easily created on the fly.
- This representation is also an appropriate data structure for caching templates for performance purposes.
- This representation provides an easy way for processing templates.

Note that the tree structure must retain the order of the constructs as it appears in the TTL template it represents. Moving a construct node on the tree around is equivalent to modifying the template's source code.

2.2.1 Tree Nodes

Every node has properties such as name and type. The name of a node is the same as the name of the construct unless it represents a free formatted text. Nodes can be of different types. These types are a reflection of the roles their corresponding constructs play in the template. Table 1 lists these types.

Table 1. Tree nodes types.

Type	Description
COMMENT	Represents a comment construct.
TEMPLATE	Represents the template construct.
DECLARE_SECTION	Represents the declare construct.
INPUT_SECTION	Represents the template's input construct.
OUTPUT_SECTION	Represents the template's output construct.
DECLARATION	Represents a variable declaration within the declaration section.
TEXT	Represents a free-formatted text within a template's input or output construct.
REFERENCE	Represents a handle used for accessing declared variables.
INPUT_CONSTRUCT	Represents any construct that can be contained within the template's input construct including user-defined input constructs.
OUTPUT_CONSTRUCT	Represents any construct that can be contained within the template's output construct including user-defined output constructs.
ROOT	This is a special type given only to one node in the template; the root node

2.3 Hello-world Template's Tree

Now that we have established the basis for our tree representation of a generic TTL template, it is time to demonstrate the whole tree representation process by showing the tree representing the hello-world template listed in Figure 1. Figure 2 illustrates just that. Clearly this tree structure can be very helpful in generating and modifying templates dynamically.

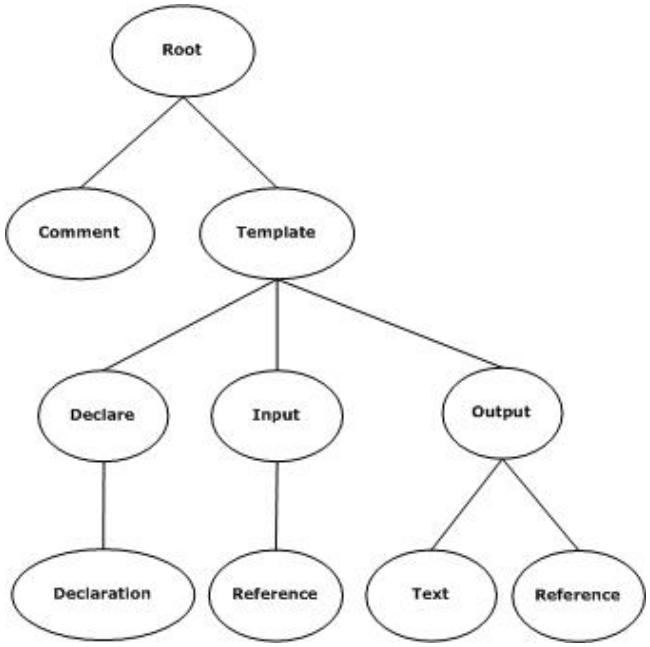

Figure 2. Hello-world template tree representation.

3. USER-DEFINED CONSTRUCTS

Software systems are living objects. They change over time; the situation that brought them to being changes; the requirements behind them change; the roles and responsibilities they assume all change over time. Software systems that do not change do not last for long nor do those that do not have the capability to change. Changeability implies flexibility. Only flexible systems that meet the current needs of business requirements and have the capability to meet more needs as they come up over time can last for long.

To be able to meet new business requirements, TTL is designed to be extensible. Extensibility in the context of this paper implies both the ability to change the behavior of an existing construct and the ability to create a brand new construct. All constructs can be extended except for the *comment*, *template*, *declare*, *input*, and *output* constructs.

There are two kinds of user-defined constructs: input user-defined constructs and output user-defined constructs. Input user-defined constructs are meant to be used within the template's input section. On the other hand output user-defined construct are meant to reside within the template's output section. Input user-defined constructs are all of type INPUT_CONSTRUCT when represented as nodes on the template's tree. Output user-defined construct are of type OUTPUT_CONSTRUCT.

3.1 TTL Construct Model

TTL Construct Model is an OOP model that defines what a construct is, how it is defined, what elements of similarity can be found between constructs, and what elements of difference are there.

3.1.1 Model Overview

The main concept of this model is the construct. A construct is an object representing a TTL construct. The Construct class is the top parent of any TTL construct. It has all the common and

generic properties of a construct. A construct object can also be thought of as a node in the template tree as described before. The Construct type is extended to represent other sub-types based on the responsibilities these types assume in TTL, as shown in Figure 3. Types *Comment*, *DeclareSection*, *InputSection*, *OutputSection*, *Root*, *Declaration*, *Text*, *Reference* and *Template* are all final; i.e. they cannot be extended any more. Types *InputConstruct* and *OutputConstruct* are where extensibility comes to play. Both types are abstract meaning that they have to be sub-typed to be used.

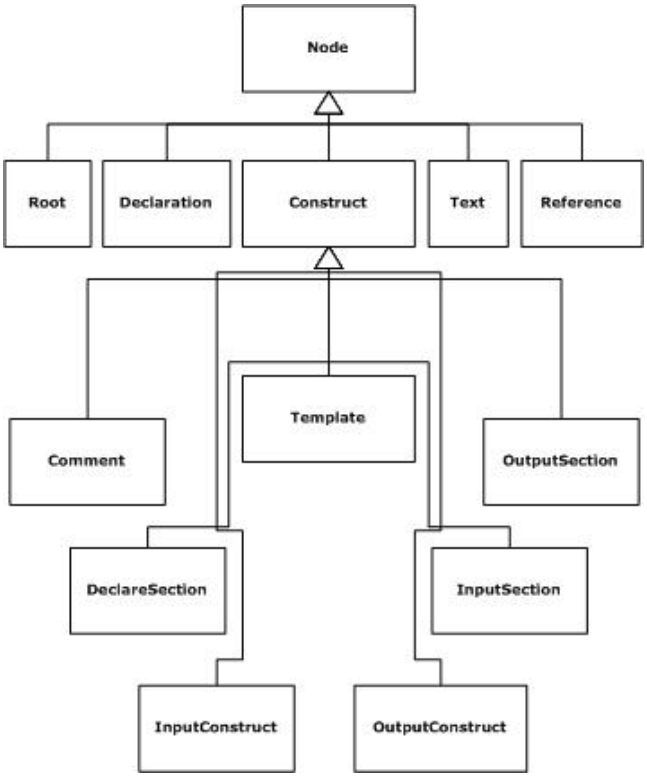

Figure 3. The construct model.

3.1.2 The Template Context

The template context is where the TTL runtime engine stores common variables. The Declare construct for example uses this context to store all of its variables. The template context has the following properties:

- It represents the state of the template at any given time.
- It represents the environment within which all constructs live and perform their roles.
- It is accessible to every construct in the template, and for that it is the glue that holds the template pieces together.
- It contains the template metadata that allows a construct to query about other constructs including its neighbors.
- The context object is associated with the template object. It gets created as part of the template creation, and when the template gets destroyed it gets destroyed too.
- The context can be serialized into a persistent form that can be later retrieved.

One of the most important usages of the template context is as a way of communication between the template's constructs.

4. SYNTAX & SEMANTICS

4.1 Constructs

The basic building unit in TTL is the construct. A construct is a block of text within the template with a certain format. As shown in Figure 4, a construct is made of at least one clause. A clause has a name, an optional parenthesis-surrounded comma-separated argument list, and an optional body. Constructs can be nested inside one another.

```
"["
(clause-name ["("arg-list ")" ]
["["
   clause-body
"]"])+
"]"
```

Figure 4. TTL construct structure.

4.2 Comments

A comment in TTL is a construct that does not have an input nor an output. TTL comments are much like XML comments in the sense that they are retained as part of the template's tree. They can only be contained either outside the template construct or inside the constructs' bodies. Figure 5 shows the format of a TTL comment.

```
[c[ Comments here. ]]
```

Figure 5. TTL comment construct.

4.3 Template

A template is the unit of development and deployment in TTL. Although it is considered as a construct itself, it is a special construct that encloses all other constructs except for comment constructs which can be found outside. Only one template is allowed in a file. Templates may not be nested within other templates.

A template has three sections each of which is a construct; a declaration section, an input section, and an output section, as shown in Figure 6.

```
[template temp-name(arg1, arg2,…)[
   [declare[
      variable declarations here.
   ]]

   [input[
      input specification here.
   ]]
```

```
   [output[
      output specification here.
   ]]
]]
```

Figure 6. A TTL template structure.

4.4 Declarations

All variables used within a template must be declared either explicitly in the declaration section or implicitly by the runtime engine when they are used for the first time. Variables in TTL have template scope; they are accessible everywhere within the template.

Variables are used within either the input construct or the output construct. They are used either as place holders for the incoming data from the input or as a way of holding data temporarily until it is processed. They are organized into five high level types:

- ***Atoms***: An atomic value is a single value. A single value can be an integer, a double, or a string of characters.
- ***Aliases***: An alias is just a label name denoting an atom, a list, an object, or an attribute of an object.
- ***Objects***: An object is a collection of attributes that belong to the entity denoted by the object. It is similar to an OOP object except that it cannot be used to call upon methods. For that they are more like records or structures in Pascal or C, respectively. *$address(street)* for instance is used to get the street part of the address object.
- ***Lists***: A list is a parenthesis-surrounded comma-separated group of items. The items contained in the list could be atoms, objects, or other lists. Arrays are considered as lists too. TTL lists use one-based indexes.
- ***Macros***: A macro is a way of re-using a block of text over and over again. Macros are shortcuts to blocks that would have had to be repeated every time a macro is called. A macro has a name, optional argument list, and a body.

Explicitly declaring variables enhances template readability.

4.5 Input Construct

For TTL templates to be able to handle all kinds of inputs, they have to specify the kinds of inputs they expect. The input construct is where the input specification takes place. If a template is not intended to have any input then this section is not required. Input files that do not respect the specifications contained in this construct are ignored.

To answer the dilemma of how input specification can be generically specified, the following built-in constructs are provided:

- ***Free-formatted text***: A free-formatted text is a block of plain text that can be in any format. When processing a free-formatted text, the runtime engine tries to find for it as much a match within the input source as it can. References which are handles to declared variables can be scattered all over the text.
- ***Regex construct***: This construct allows for a Perl 5 style of regular expressions. A regular expression describes a pattern to be looked up and matched when reading the input.
- ***Loop construct***: This construct allows the developer to specify a repeating pattern expected in the input file. When processing the loop construct, the runtime engine tries to match as much input is it can. As this construct proceeds, it collects information that will be used later on in the output section.
- ***Ignore construct***: There are situations where some blocks in the input file are meant to be ignored. This construct specifies the patterns that should be ignored.

4.6 Output Construct

The output construct is where the template heart is. It specifies exactly how the template outcome should look like. Within this construct, the output of the template is synthesized. For doing so, the following built-in constructs are provided:

- ***Free-formatted text***: This construct is similar to that of the input construct.
- ***If-elsif-else construct***: This construct is used to select a branch of processing based on whether a given condition is satisfied. If no condition holds and the else part of this construct exists, the body of the else part will be processed.
- ***While construct***: The While construct instructs the runtime engine to continue processing of the block enclosed within its body until the condition given at the beginning does not hold any more. In that case, the runtime engine moves to the next construct.
- ***Foreach construct***: This is another form of TTL repetition constructs which allows for a great deal of flexibility. It walks through the elements of the given lists one item at a time while processing its body. This construct is ideal for iterating through single or multiple lists.
- ***Continue construct***: This construct is used within the body of either the While construct or the Foreach construct or any other constructs enclosed in their bodies. It causes control to leave the remaining of the current iteration and jump to the next iteration if any.
- ***Break construct***: The Break construct is used within the body of any construct to exit that construct or any enclosing construct. Since control could be within a very long nested chain of constructs, it is necessary to specify to what level control should jump to. For that, the break construct takes one numeric argument that tells it how many enclosing constructs to get out of.
- ***Eval construct***: The Eval construct is where expressions can be evaluated. It is also used to assign the evaluated values to declared variables.
- ***Call construct***: TTL provides this construct to make it possible for a template to call other templates. When a

template is called from another template, the input to the called template can be either passed as parameters, or constructed in the body of the Call construct or both.

- ***Include construct***: In many situations there is a need to include the contents of a file or a list of files into the output of a template without involving any processing. This construct does that.
- ***String construct***: A great deal of what most template languages do is about string manipulation. For that TTL provides developers with this construct. This construct has many frequently used string manipulation operations such as concatenate, substring, index, and upper/lower case.
- ***Math construct***: This construct is similar to the string construct mentioned above. It comes as a result of recognizing the fact that more complex operations than the basic arithmetic operations that are used in TTL expressions are needed. This construct allows for arithmetic and mathematic operations such as sin, cos, tan, abs, and random.

5. RELATED WORK

Templates have been found useful since the early days of computing. Many software systems are using them. In principle, there is no limit to what templates can be used for. In reality, however, their usage is dependent on the environment they are part of.

One computing area in which templates have been frequently used is text processing. In such area, templates are used for reporting, presenting, and authoring text data using languages like AWK, Perl, and Python [2][3][4].

Content management systems are another computing area in which templates are used extensively. Typically, such systems have many users that contribute to the contents delivered to their clients. Templates are used to maintain commonality among all contents.

Most of template languages were created to be used with specific applications. They all contain place holders and many of them accept only input parameters. Having input parameters as the only input source make such templates unsuitable for manipulating high-volume inputs since it is impractical to pass a large amount of data as input parameters.

XSL templates however are different from other templates in the sense that they allow for transforming their inputs to outputs of some formats. Yet they only accept well-formatted XML documents as inputs [1][6][9]. XSL templates are transformational templates and for that they have also been widely used in web applications. Cocoon for instance uses XSL templates to process input XML documents and produce HTML documents on the fly [8]. Other template languages such as PTL, and Velocity Templates are used in conjunction with other languages such as Python, PHP, Java [5][7].

Finally, it is important to mention that the work on TTL is only a continuation of the work that has been done on many template languages such as XSL, Velocity, and much more.

6. A NOTE ON IMPLEMENTATION

There is not an ideal language in which user-defined constructs can be implemented. Any language with input and output capability such as C, C++, and Java can be used. More than one language can also be used. When this happens, the requirement that:

> *Any user-defined construct created using one implementation language must be guaranteed to continue working under another TTL engine implemented in another language.*

is absolutely a must to achieve independence from the underlying implementation language.

A reference implementation in Java is underway at the time of writing this paper.

7. ACKNOWLEDGMENTS

I would like to express my sincere appreciation to Jonathan Cook of New Mexico State University for his invaluable comments and suggestions without which this paper would not have been the way it is.

8. REFERENCES

[1] Extensible Stylesheet Language (XSL) **`http://www.w3.org/TR/xsl/`**

[2] Introduction to akw **`http://allman.rhon.itam.mx/dcomp/awk.html`**

[3] Schwartz, Randal L. and Phoenix, Tom. Learning Perl O'Reilly; 3rd edition (July 15, 2001)

[4] Python Documentation **`http://docs.python.org/download.html`**

[5] PHP Manual **`http://www.php.net/manual/en/`**

[6] Deitel, Harvey M., Deitel, Paul J., Nieto, T. R., Lin, Ted, Sashu, Praveen. XML: XML How to Program., Prentice Hall,2000

[7] Velocity Templates **`http://jakarta.apache.org/velocity/index.html`**

[8] The Apache Cocoon Project **`http://cocoon.apache.org`**

[9] McLaughlin, Brett. Java and XML, O'Reilly, 2001