Umple as a Template Language (Umple-TL)

Mahmoud Husseini Orabi, Ahmed Husseini Orabi and Timothy C. Lethbridge
School of Electrical Engineering and Computer Science, University Of Ottawa,
800 King Edward Avenue, Ottawa, Canada

Keywords: Umple, Umple-TL, Templates, Text Emission.

Abstract: We extend Umple, a model-oriented programming language, to incorporate text emission from templates as an integral feature. Umple as a template language (Umple-TL) is the term we use to describe the template sublanguage of Umple. Using Umple-TL, developers can benefit from synergies among UML modelling, templating and programming in several target languages (Java, C++), all in one textual language – Umple. Umple itself is written in Umple; using Umple-TL, we eliminated Umple's dependency on third-party libraries for text emission. We also support any other application developed in JET to be converted to use Umple-TL and attain benefits such as smaller and faster executables, target-language independence and IDE independence. The word 'template' in this paper refers to patterns for the generation of output, and not to generic types, another common use of the term.

1 INTRODUCTION

Umple is a textual modelling language with code generation capabilities for target languages such as Java, C++, PHP, and Ruby. plus generation of other artifacts such as diagrams (Orabi, Orabi, and Lethbridge, 2016). A goal of Umple is to keep the Umple source as 'master' and to insulate the developer from ever having to see generated code.

Users write models in Umple, and inject any additional needed target-language code directly into the textual model. An alternative and equivalent perspective is that Umple model constructs can be injected into target-language code to simplify it and reduce its bulk.

In this paper, we show how we extended Umple to act as a template language, so all text emission can be entirely represented in Umple. The word 'template' here refers to patterns of text to output, not to generic types, which is another use of the term 'template' in programming languages.

Languages such as PHP were designed with generation of textual output as their motivating use case. But languages such as Java do not come with built-in template mechanisms and rely on verbose method calls to generate text.

There are a wide variety of contexts where generating formatted textual content is an essential requirement: These include generation of data formats such as XML and html, generation of modelling and programming language code (in metaprogramming and code generation), generation of messages for inter-process communication, and generation of user interfaces.

A key objective of Umple is to make software development simpler by adding modelling and other constructs to base languages. Templates are just one of the many abstractions that have been added to Umple as a necessary step to achieve Umple's overall goals. Our intent is to obtain synergies by combining templates with modelling in an easy-to-use language.

A distinguishing feature of Umple-TL is that it supports template development for *multiple target languages*, allowing reuse of templates across such languages.

We will explain how Umple as a Template Language (Umple-TL) is used. The examples shown in this paper can be instantly run in the UmpleOnline online editor (try.umple.org), the Umple Eclipse plugin or the Umple command line compiler.

2 MAIN CONCEPTS

There are three key types of textual entity involved in Umple-TL:

Umple Source Text: This is the master code of the system under development. It includes templates,

Orabi, M., Orabi, A. and Lethbridge, T.
Umple as a Template Language (Umple-TL).
DOI: 10.5220/0007382000960104
In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 96-104
ISBN: 978-989-758-358-2

model constructs, and target-language (e.g. Java, C++) methods. Umple-TL is simply the elements of the Umple language that relate to generation of text from templates. The Umple compiler processes the source text to produce the following:

Compiled (Generated) Code: This is generated by the Umple compiler in a target language (e.g. C++. Java), and is intended to be ignored by the programmer (as would be byte code and other intermediate languages). This is executed to produce the following:

Runtime Output Text: This is the output when the compiled code is executed. The runtime output could be any type of text, including error messages, html, and so on. If the Umple source text describes a compiler, then the runtime output would be code itself. This is the case with Umple, which compiles *itself*; in fact a key motivation for Umple-TL to make it easier for Umple to be used to develop its own code generators (Orabi, 2017).

The two key elements in Umple-TL source text are *templates* themselves and *emitter methods*.

- **Templates:** These describe the runtime output text to be generated. A template is a specialized Umple attribute with a unique label and a body defined within a block delimited by <<! and !>>. Table 1 describes the blocks that can be nested inside the outer delimiters.
- **Emitter Methods:** These are invoked to create outputs based on one or more templates. The keyword 'emit' is used to indicate these (Snippet 1 – Line 3).

Since an Umple-TL template is an attribute, it must follow the naming rules of attributes – a name cannot start with a number, nor have the same name as any other attribute or template.

Similarly, an emitter method is a specialized Umple method, meaning that it must follow the naming conventions imposed on normal Umple methods. Arguments to the emitter method are referred to in expression and code blocks.

At least one emitter method and one template are required for a class to be recognized as a template class. Emitter methods are required since they enable passing of arguments to templates, and composing multiple templates.

2.1 Usage of the Various Blocks

Snippet 1 shows a simple template labelled as t1 (Line 2) that is used by emitter method, e1 (Line 3). The emitter method in this case has no arguments and can be called from any part of the system as a normal method, including in the expression blocks of other templates. For better usability, parentheses are optional when no parameters are defined similarly to languages such as Scala (Snippet 6 – Line 4).

1	**class** TemplateTest1{	***Umple***
2	*t1* <<! My Template !>>	
3	**emit** e1()(*t1*);	
4	}	

Snippet 1: A simple Umple-TL example.

Table 1: A summary of the syntax for the blocks used to write templates in Umple-TL.

Block type	Description
<<! {b} !>>	**Top Level:** Defines the start and end of a template body. The {b} represents arbitrary text to output, with any of the following nested within.
<<={e} >>	**Expression:** Computes strings to be inserted into the text. The {e} represents any expression in the target language that returns a string, such as a variable or method invocation.
<<# {c} #>>	**Code:** Code in the target language to define logical conditions (e.g. to make output of parts of the template optional) and loops. The content of code blocks is not appended to the runtime output text, but instead appears in the compiled code.
<</* */>>	**Comment:** Material not added to the runtime output, but which does appear in the compiled code. It must be in the syntax of the target language used. Comments could also be placed in code blocks, but using comment blocks can be simpler.
<<$ >>	**Exact space:** Specifies whitespace that will appear at the beginning of every line in the runtime output text.

Line 2 of Snippet 2 shows the use of a code block delineated by <<# and #>>. The code in the block can be in any target language that Umple supports, which includes Java and C++. The examples shown in this paper use Java. It is the developer's responsibility to

write valid code according to the target language of their selection.

```
1  class TemplateTest2{                                   Umple
2     t2a <<!<<#if(b)#>>This will be output if b is true !>>
3     t2b <<! … and this will always be output !>>
4     emit e2(Boolean b)(t2a, t2b);
5  }
```

Snippet 2: Umple-TL example illustrating a code block and multiple templates.

Lines 4 of Snippet 2 also shows an emitter method e2() that has an argument, b, referred to in template t2a (Line 2). The emitter method also demonstrates emission of two templates.

In Snippet 3, there are two variables string1 (Line 2) and string2 (Line 3). Umple creates a getter method for any public attribute (Lines 5 and 6) in a class; hence getString1() and getString2() will be generated, although code in a generated class can still access the variable directly without using these methods. The snippet has two expression blocks. The first uses the getString1(), and the second directly appends the value of string2.

```
1  class TemplateTest3{                                   Umple
2     String string1;
3     String string2;
4
5     t3<<!   String1=<<=getString1()>>;   String2=
6  <<=string2>>!>>
7     emit generate1()(t3);
8  }
```

Snippet 3: Using assign statements in Umple-TL.

A developer can use a combination of expression and code blocks. Snippet 4 shows an example of an expression block that uses a for loop to repeatedly print out parts of the content of the template block the number of times indicated by the argument *iterations*.

```
1  class TemplateTest4{                                   Umple
2     t4 <<!
3        <</* Use iterations to control output*/>>
4        <<# for(int index=0;
5             index<iterations; index++){#>>
6          Iteration <<=index>>;<<#
7        }#>>!>>
8     emit generate1(int iterations)(t4);
9  }
```

Snippet 4: An Umple-TL example using expression and code blocks.

Snippet 4 also shows a comment block in line 3.

By default, whitespace appears in the runtime output as it appears in the template. However, a given template may need to generate different indentation (whitespace at the start of lines) in different contexts. This can be necessary, for example, when code is being generated, and the structure of the code needs to be readable.

To control the indentation in runtime output, the developer can write an exact space block. This will fix the indentation of its contents no matter how that output is generated (e.g. from an expression, variable, or plain text).

In Snippet 5, there are two templates, internalTemplate and t5. In Line 2, t5 internally invokes generated emitter method internalGenerate() using exact space markers. There are four whitespaces after the <<$. These four spaces will appear in all lines of the runtime output.

```
1  class TemplateTest5{                                   Umple
2     t5 <<!<<$   internalGenerate()>>,!>>
3     internalTemplate<<!Some content!>>
4
5     emit generate()(t5);
6     private emit internalGenerate()(internalTemplate);
7  }
```

Snippet 5: An example of exact space handling.

Alternatively, developers can directly use the generated emission method generate() and pass as a second argument the amount of indentation required.

2.2 Emitter Methods

An emitter method behaves like a regular method (Snippet 6), but instead of writing a method body, the developer simply lists templates to be used for text emission.

A generated emitter method will output the content of all of the referenced templates. The templates will be concatenated in the order they are specified.

If there are two sets of parentheses following the name of an emitter method, the first is a list of arguments, and the second is a list of templates to emit. If there is just one set of parentheses, it is the list of templates and the method has no arguments.

Emitter methods assume public visibility by default. The emitter method in Line 4 of Snippet 6 is public with no parameters. The method above it (Line 3) differs only in that it is static.

```
1  class TemplateTest{                                   Umple
2      t6 <<! My Template !>>
3      public static emit generate1()(t6);
4      emit generate2(t6);
5  }
```

Snippet 6: Emitter method examples.

Snippet 7 shows an example of an emitter method that outputs three templates.

```
1  class TemplateTest{                                   Umple
2      t7a <<!Content1!>>
3      t7b <<!Content2!>>
4      t7d <<!Content3!>>
5
6      emit generate1()(t7a, t7b, t7c);
7  }
```

Snippet 7: Multiple references to templates in an emitter method.

Line 9 of Snippet 8, shows an emitter method internalGenerate() marked *private*. As such, it can only be called internally to the class; in this example it is invoked by another template t8 (Line 3).

```
1  class TemplateTest{                                   Umple
2    internalTemplate<<! Some content !>>
3    t8<<!<<=internalGenerate()>>!>>
4
5    emit generate()(t8);
6    private emit internalGenerate()(internalTemplate);
7  }
```

Snippet 8: An internal invocation of an emitter method.

The basic distinction between an emitter method and template, is that an emitter method is used to utilize a template or a group of templates, and to define other features such as formatting. In order to improve usability in the future, we are considering generating a default emitter method for each template, such that users can directly define templates without needing to define emitter methods if there is no special logic behind using these templates.

3 UML CONSTRUCTS AND GENERATION TEMPLATES

We have described how to use Umple-TL to implement basic features needed in template generation. In this section, we show how Umple-TL can use the UML modelling, separation-of-concerns and template features of Umple in a synergistic way. Umple UML modelling constructs include associations, state machine, and composite structure. Separation of concerns features include mixins, traits and aspects (Badreddin, Lethbridge, and Forward, 2014). This synergy is one of the key contributions of our work. In this section, we will focus on using

```
1   class Course {                                      Umple
2     String name;
3     String description;
4     0..1 -- * Student student;
5     cr <<!
6     !>>
7
8     courseInfo <<!
9     Course: <<=getName()>>
10    Description: <<=getDescription()>>
11    Number of registered students: <<=numberOfStudent()>>
12    <<# switch(getStatus()) {#>><<# case Opened:#>>
13    <<# switch(getStatusOpened()) {#>><<# case
14  WithoutLateRegistrationFees:#>>
15    Last day to register without late fees <<=d1>>
16    <<# case WithLateRegistrationFees :#>>
17    Last day to register with late fees <<=d2>>
18    <<# } #>>
19    <<# case Registered:#>>
20    Last day to withdraw from a course <<=d2>>
21    <<# } #>>
22    !>>
23
24    status{
25        Opened {
26            register -> Registered;
27            close -> Closed;
28            WithoutLateRegistrationFees {
29                register -> Registered;
30              deadLinePassed -> WithLateRegistrationFees;
31            }
32            WithLateRegistrationFees {
33                register -> Registered;
34                deadLinePassed -> Closed;
35            }
36        }
37
38        Registered {
39            requestToWithdrow -> Withdrawn;
40            LastDayToWithdraw {
41                requestToWithdrow -> Withdrawn;
42                deadLinePassed -> Closed;
43            }
44        }
45        Withdrawn { }
46        Closed {
47            open -> Opened;
48        }
49    }
50
51    emit printCourseInfo(String d1, String d2,
52         String, d3 )(courseInfo, cr);
53  }
54
55  class Student{
56     String name;
57     Integer id;
58  }
```

Snippet 9: An example of using UML constructs to develop templates.

templates with state machines.

Snippet 9 shows an Umple model that displays course information based on the number of the registered students. A student has a name and id, while a course has a name and description. According to Line 4, there is an association defined between a course and students. A course can have zero or more students and a student may or may not be in a course.

A course has four statuses, Opened (Line 25), Registered (Line 38), Withdrawn (Line 45), and Closed (Line 46). Based on the status, the displayed template information changes. If a course status is Opened, all information including fees will be displayed; otherwise, it is not shown. The description and number of students is displayed regardless of course status. If a course is in the Registered status, information such as last day to withdraw is shown.

Figure 1 shows the state machine defined in Snippet 9 (generated automatically by Umple). The logic of the state machine is given textually in Lines 24-49.

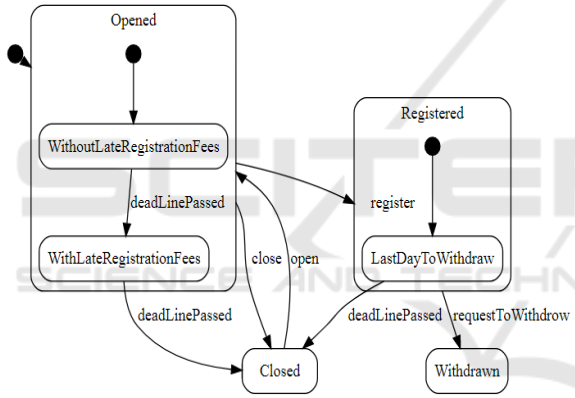

Figure 1: The state machine diagram of Snippet 9.

Snippet 10 shows an execution of the generated code of Snippet 9. For simplicity, we show the console output in green as lines of comments. In Snippet 10, we define two students and add both to a course. A course status is "Opened" by default. The first information printed is full course information, given that the course status is Opened. The output is shown in Lines 10-15.

In Line 17, we change the status of the course to be Closed. The output in that case, Lines 19-21, only shows the basic information about the course and its students.

We change the status back to Opened (Snippet 10-Line 23). Now the lines to be printed are as in Lines 25-30, which are identical to the Lines 10-14.

```
1  Student student1 = new Student("Name1", 1234);                    Java
2  Student student2 = new Student("Name2", 432);
3  String d1="JA 1"; String  d2="FE 1"; String d3="AP 1";
4
5  Course course = new Course("Course1", "This is a course");
6  course.addStudent(student1);
7  course.addStudent(student2);
8
9  System.out.println(course.printCourseInfo(d1,d2,d3));
10 //Course: Course1
11 //Description: This is a course
12 //Number of registered students: 2
13 //Last day to register without late registration fees JA 1
14 //Last day to register with late registration fees FE 1
15 //Last day to withdraw from a course AP 1
16
17 course.close();
18 System.out.println(course.printCourseInfo(d1,d2,d3));
19 //Course: Course1
20 //Description: This is a course
21 //Number of registered students: 2
22
23 course.open();
24 System.out.println(course.printCourseInfo(d1,d2,d3));
25 //Course: Course1
26 //Description: This is a course
27 //Number of registered students: 2
28 //Last day to register without late registration fees JA 1
29 //Last day to register with late registration fees FE 1
30 //Last day to withdraw from a course AP 1
31
32 course.register();
33 System.out.println(course.printCourseInfo(d1,d2,d3));
34 //Course: Course1
35 //Description: This is a course
36 //Number of registered students: 2
37 //Last day to withdraw from a course AP 1
```

Snippet 10: An invocation example of Snippet 9.

Finally, In Line 31, we change the status to Registered. This is similar to Closed state, but additionally shows the last day to withdraw (Lines 34-37).

3.1 Declarative Examples

In this section, we show two Umple-TL examples for HTML page generation. The first example (Snippet 11) is more tailored, and hence requires a few lines to execute (Snippet 12). The second example (Snippet 13) is more abstract, so it can support dynamic table generation, but will require more lines of code to define custom tables (Snippet 14).

In Snippet 13, HtmlNode has two attributes, tag and content. The tag attribute refers to a valid html tag such as html or body. The "content" attribute is optional and has an empty value by default; it refers to the text content of an html node. There is an association attribute; "children". Associations are one

```
1  class HtmlTemplate {                                   Umple
2   htmlTemplate <<!
3       <html>
4           <body>
5               <table>
6               <<# for  (String name: names) {#>>
7                   <tr>
8                       <td>
9                           <<=name>>
10                      </td>
11                  </tr>
12              <<#}#>>
13              </table>
14          </body>
15      </html>
16  !>>
17
18   emit printHTML(List<String> names)(htmlTemplate);
19  }
```

Snippet 11: HTML template generation (1).

```
1  System.out.println(new                                 Java
2  HtmlTemplate().printHTML(Arrays.asList(new
3  String[]{"Row1", "Row2", "Row3"}));));
```

Snippet 12: An invocation example of Snippet 11.

of the key features that Umple provides. Umple provides all different types and variations of associations. In (Snippet 13-Line 5), the type of association used is optional unbound self-reflexive; this means that it refers to an unlimited number of children of the same class, HtmlNode (or its subclasses). In addition, this list of children can be empty, which means that it is fine for an HtmlNode to have an empty list of children.

```
1  class HtmlNode{                                        Umple
2   String tag;
3   String content= "";
4
5   0..1--* HtmlNode children;
6
7    nested <<!<<#for(HtmlNode node: children) {
8           #>><<=node.generate()>><<#
9    }#>>!>>
10
11   print <<!<<<=tag>>>
12  <<=content>><<<=nestedPrint()>>
13  </<<=tag>>>!>>
14
15   emit generate()(print);
16  private emit nestedPrint()(nested);
17 }
```

Snippet 13: HTML template generation (2).

There are two emitter methods for two templates, "print" and "nested". The "print" template is the main template that is used to print out the content of an HtmlNode instance. The content of an HtmlNode instance includes the nested content of its children in nested ways. The "nested" template is used to take care of looping into the children list of an HtmlNode instance; a child node itself can have a list of children. This will continue recursively until a node does not have any children.

```
1  HtmlNode html = new HtmlNode("html");                  Java
2  HtmlNode body = new HtmlNode("body");
3  html.addChild(body);
4
5  HtmlNode table = new HtmlNode("table");
6  body.addChild(table);
7
8  for (String label : Arrays.asList(new String[] {
9      "Row1", "Row2", "Row3" })) {
10     HtmlNode row = new HtmlNode("tr");
11     table.addChild(row);
12
13     HtmlNode tableData = new HtmlNode("td");
14     tableData.setContent(label);
15     row.addChild(tableData);
16 }
17
18 System.out.println(html.generate());
```

Snippet 14: An invocation example of Snippet 13.

Snippet 14 shows how the model written in Snippet 13 can be used to print out similar content to Snippet 11.

4 DEMONSTRATON OF PRACTICAL VALUE

To evaluate this work, we use two approaches. The first is demonstration of the value of our work in practice, discussed in this section. The second is the use of concrete metrics, discussed in the next section

The ultimate demonstration of the practical value of the work is that the Umple compiler is written in Umple, with all artifact generation (diagram DSLs, Java, C++, Ruby, PHP, XML interchange data, etc.) using Umple-TL.

Before Umple-TL was added, the compiler had used JET (Eclipse, 2003) to emit the text of generated artifacts (Umple, 2018). However JET was deprecated, and we faced limits on its capacity (maximum size of strings). For JET replacement, we considered a variety of solutions, including Eclipse tools such as Acceleo (Acceleo, 2017). However that would have tied Umple to both Eclipse and Acceleo, and would have added a lot of complexity for developers. We wanted a very simple solution, hence we developed Umple-TL to fit synergistically with other Umple features while at the same time making

template generation available for all Umple-developed applications.

After the introduction of Umple-TL, a tool was developed to automatically transform all templates written in JET into Umple-TL (Umple, 2018). This tool was applied to the Umple compiler and various other projects, with translation being accomplished in just a few minutes! The Umple compiler has hence become an extremely large test-case for Umple-TL, and also has proved that it works effectively.

Table 2: Evaluation. Marginally better values in italics; values that are more than 4% better in red bold.

Generators	JET			Umple		
	MasterSource LOC	Generated Java LOC	Generation Time (ms)	MasterSource LOC	Generated Java LOC	Generation Time (ms)
Java	*11594*	21283	5988	11877	**20287**	**4741**
PHP	*4760*	7544	3938	4909	*7444*	**3031**
C++				5409	10549	7908

The real-time C++ code generator in the Umple compiler never did use JET, and was directly written in Umple-TL, demonstrating that manual coding of Umple-TL is usable. It uses the Umple-TL features and capabilities more extensively than the generators automatically converted from JET since it offers a comprehensive set of features as compared to JET.

5 PERFORMANCE MEASURES

Performance of a templating tool can be measured based on three dimensions: a) Source templating code should be as small as possible, facilitating understanding; b) generated code should be as small as possible, reducing the size of runtime executables; and c) generation time, i.e. processing and compiling the templates code to produce a text generator, should be small.

In the following we compare the performance of the Umple compiler when it used Jet, as of release 1.23.1, on March 22, 2016 (Umple, 2016a), and its performance in release 1.24.0 the same day, after it was converted to use Umple-TL(Umple, 2016b). Note that Umple has been enhanced since the day of that conversion so the metrics as of the latest Umple release (Github Umple, 2018a) will be different.

The metrics are presented in Table 2 and further illustrated in Figure 2 and Figure 3. In Table 2 we compare the templating code written in JET (left), with the code written in Umple-TL (right).

Code size is measured in Lines of code (LOC) and translation time for a set of test cases is measured in milliseconds. The generators shown in our comparison are Java (Github Umple, 2018b), and PHP (Github Umple, 2018c) as transcoded from Jet. Data for C++ templates manually written in Umple-TL(Github Umple, 2018d) is shown for comparison in Table 2. The master source code refers to code written either in Umple-TL or JET, while the generated code refers to the code of the Umple compiler generated (by Jet of Umple-TL) in Java.

In Java and PHP, we can see that the master LOC is almost the same as the Umple-TL. However, Umple-TL is a bit larger, because it must be enclosed in a class (Snippet 1). The PhP generation template code LOC count is smaller (for both Jet and Java templates) than the Java generation template code because Umple supports fewer features when generating Php.

The C++ generator, which supports the same features as the Java generator, requires many fewer lines of template code, as it better employs Umple-TL features, resulting in 45% LOC reduction.

As is shown in Figure 3, The emission time of Umple-TL was faster than Jet in both Java and PHP generators.

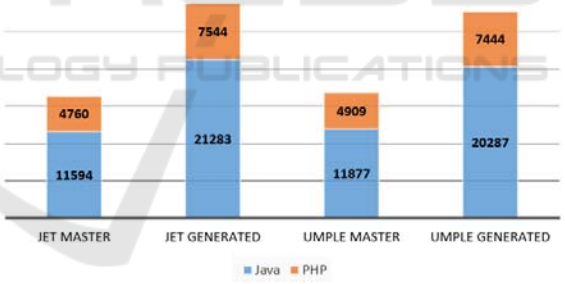

Figure 2: LOC comparison between Jet (left) and Umple (right). Master is the source in Jet or Umple.

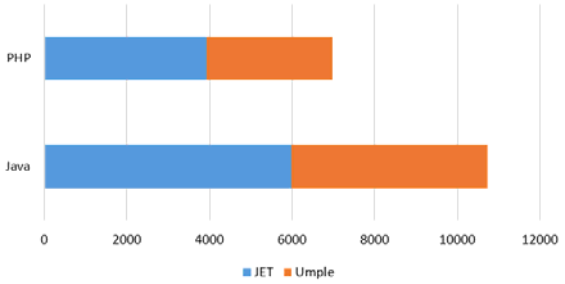

Figure 3: Generation time comparison.

6 RELATED WORK

There are many text emission tools such as Java Emitter Templates (JET), Apache Velocity, Acceleo, Epsilon Generation Language (EGL), Xpand, and Xtend.

JET was one of the most commonly-used Eclipse-based textual generation tools due to its ease of use and straightforwardness. In order to use JET, a developer will need to create a Java project containing the JET nature. JET has a JSP-like syntax, and it uses a skeleton template to customize text emission. JET does not provide a concise way to define rules that relate different JET files. It aims to reduces the complexity of text emission by using a single emitter method with no parameters. However, this approach requires additional configurations and eclipse dependencies, and restricts code development options.

Velocity Template Engine (VTL) is a part of the Apache Velocity project (Carnell, Harrop, and Mittal, 2006). Velocity provides an easy way to develop generation units based on a Model-View-Controller (MVC) pattern. Velocity requires dependencies on third-party libraries. Using Velocity requires a runtime library to generate the outputs of the VM files. This means that a Velocity configuration as well as the developed VM files must be a part of any product release.

Acceleo (or MTL) is an Eclipse modelling project that enables UML modelling and code generation. It follows the Object Management Group (OMG) specifications for the Model to Text Language (MTL) standard. Acceleo is installed on the top of Eclipse, and it requires additional libraries such as EMF and Ecore. The code generation requires familiarity with model-driven development tools and knowledge of ECore. An Acceleo model is written using the Eclipse Ecore XML Metadata Interchange (XMI) syntax. This model can have a hierarchical representation. It is used to define and associate the generation units, as well as the parameters and items required to generate the content of template files; i.e. Model Template (MT).

Xtend is an Eclipse-based project that was initially released as a part of the Xtext project (Xtend, 2017). After that, Xtend became a standalone Eclipse project. Xtend is intended to replace the Xpand workflow. Being influenced by many languages and projects such as Scala and Xpand, Xtend tries to improve the Java programming language by introducing additional capabilities and major features such as functional programming, text emission, operator overloading, and dynamic typing. Similarly to velocity, Xtend is restricted to Java applications only and requires a runtime library.

Epsilon Generation Language (EGL) or simply Epsilon is an Eclipse project that gives several options in terms of code generation including text emission. In a similar manner to Acceleo, a model is used to manage the content of a generation process. Epsilon has an advantage over Acceleo that there is no specific restriction on a certain model type. The Epsilon Model Connectivity (EMC) layer is used to enforce a model-driven paradigm by associating metamodels of several types such as EMF or XML There are several features and capabilities provided by the Epsilon script such as expression statements, polymorphism and annotations. However, these capabilities are dependent on a runtime library, meaning that it will have the same limitations that we referred to in Velocity.

7 CONCLUSIONS

Umple, as both a modeling and programming language, provides a unified approach to develop executable models. One goal of Umple to enable usable textual modeling. Another is to ensure that both Umple sources and any generated systems avoid dependency on third party libraries or IDEs. The introduction of Umple as a Template Language (Umple-TL) has helped achieve this goal.

Similarly to other features of Umple, Umple-TL is target-language-agnostic. We showed during our discussion in this paper that we were able to write Umple models with the assumption that a target language can be C++, PHP or Java. Language-independence is one of the core advantages that Umple-TL can provide as opposed to some of the commonly used template generation tools such as JET (Eclipse, 2003), Xtend (Xtend, 2017), and Apache Velocity (Carnell et al., 2006), which all restrict the development to Java.

Umple provides a high level of abstraction; this requires ensuring that development will be on a single artifact; an Umple model or a collection of Umple models. Thus, using Umple-TL, all template development will be in Umple without asking users to switch among different development contexts, compilers, file types or IDEs.

We showed that compared to JET, Umple-TL can help reduce the emission time and generated code size, while not impacting source size. Also, with Umple-TL, developers can take advantages of UML, and object orientation features that Umple provides.

ACKNOWLEDGEMENTS

This research was supported by OGS, NSERC, and ORF.

REFERENCES

Acceleo. (2017). Acceleo eclipse page. Retrieved October 1, 2017, from http://www.eclipse.org/acceleo/

Badreddin, O., Lethbridge, T. C., and Forward, A. (2014). A Test-Driven Approach for Developing Software Languages. In *MODELSWARD 2014, International Conference on Model-Driven Engineering and Software Development* (pp. 225–234). SCITEPRESS - Science and and Technology Publications. https://doi.org/10.5220/0004699502250234

Carnell, J., Harrop, R., and Mittal, K. (2006). Velocity Template Engine. In *Pro Apache Struts with Ajax* (pp. 317–357). https://doi.org/10.1007/978-1-4302-0252-3

Eclipse. (2003). JET Tutorial (Introduction to JET). Retrieved October 1, 2017, from https://eclipse.org/articles/Article-JET/jet_tutorial1.html

Github Umple. (2018a). Github Umple. Retrieved from https://github.com/umple/umple/releases/latest

Github Umple. (2018b). Github Umple. Retrieved from https://github.com/umple/umple/tree/master/UmpleToJava/UmpleTLTemplates

Github Umple. (2018c). Github Umple. Retrieved from https://github.com/umple/umple/tree/master/UmpleToPhp/UmpleTLTemplates

Github Umple. (2018d). Github Umple. Retrieved from https://github.com/umple/umple/tree/master/UmpleToRTCpp/UmpleTLTemplates

Orabi, M. H. (2017). *Facilitating the Representation of Composite Structure, Active objects, Code Generation, and Software Component Descriptions for AUTOSAR in the Umple Model-Oriented Programming Language (PhD Thesis)*. University of Ottawa. https://doi.org/10.20381/ruor-20732

Orabi, M. H., Orabi, A. H., and Lethbridge, T. (2016). Umple as a Component-based Language for the Development of Real-time and Embedded Applications. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development* (pp. 282–291). SCITEPRESS - Science and and Technology Publications. https://doi.org/10.5220/0005741502820291

Umple. (2016a). Github Umple. Retrieved from https://github.com/umple/umple/releases/tag/v.1.23.1

Umple. (2016b). Github Umple. Retrieved from https://github.com/umple/umple/releases/tag/v.1.24.0

Umple. (2018). JETToUmpleTL. Retrieved from https://github.com/umple/JETToUmpleTL

Xtend. (2017). Xtend. Retrieved January 1, 2015, from http://eclipse.org/xtend/