

# TinyTL

Dominique Guinard & Karin Altorfer

June 29, 2006

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
<b>2</b>	<b>User Manual</b>	<b>3</b>
<b>3</b>	<b>Architecture</b>	<b>3</b>
3.1	Overview . . . . .	3
3.2	Given Components . . . . .	4
3.3	TTLParser . . . . .	4
3.4	TTLInterpreter . . . . .	5
3.4.1	Basic Principles . . . . .	5
3.4.2	Example Translations . . . . .	5
<b>4</b>	<b>Validation Functions</b>	<b>6</b>
4.1	Program . . . . .	7
4.2	Pattern . . . . .	8
4.3	Template . . . . .	8
4.4	Item . . . . .	8
4.5	NodeRule . . . . .	9
4.6	NodeExpression . . . . .	9
4.7	ElmtList . . . . .	10
4.8	StepExpr . . . . .	10
4.9	Direction . . . . .	10
4.10	Application . . . . .	10
4.11	Selection . . . . .	11
4.12	Copy . . . . .	11
4.13	TextNode . . . . .	12
4.14	AttrNode . . . . .	12
4.15	ElmtNode . . . . .	12
<b>5</b>	<b>Examples</b>	<b>12</b>
5.1	Creating a CSV (Comma Separated Values) File . . . . .	13
5.2	Creating an HTML Web Page . . . . .	13
<b>6</b>	<b>Conclusion</b>	<b>13</b>
<b>A</b>	<b>Examples</b>	<b>14</b>
A.1	The XML <sup>-</sup> Input File ( <code>input.xml</code> ) used by both Examples . . . . .	14
A.2	The CSV Example . . . . .	14
A.2.1	The TinyTL Input Program <code>csv.ttl</code> . . . . .	14
A.2.2	The corresponding XML <sup>-</sup> Output File <code>csv.xml</code> . . . . .	15
A.3	The HTML Example . . . . .	15
A.3.1	The TinyTL Input Program <code>createWebPage.ttl</code> . . . . .	15
A.3.2	The corresponding HTML Output File <code>test.html</code> . . . . .	17
<b>B</b>	<b>Abstract Syntax XML<sup>-</sup> and TinyTL</b>	<b>18</b>
<b>C</b>	<b>Dynamic Semantics TinyTL</b>	<b>18</b>

# 1 Introduction

TinyTL takes two input files. The first is a simple XML document, whereas we are working with a reduced version of XML that we shall call *XML<sup>-</sup>*. The second is a TinyTL document. During the interpretation, the transformation rules defined in the TinyTL document will be applied onto the XML file. The output is a transformed XML document.

This report is structured as follows: section 2 explains how the program can be used and executed. Section 3 outlines and describes the architecture and its different components of the application. An important part of this documentation, section 4, explains and formalises the different validation functions. In section 5 two examples are briefly explained. In the appendix A those examples are given in detail by actually giving the XML input document, a TinyTL program and the corresponding XML output document.

For more specific information on the source code, please refer to the generated Java documentation that we extensively commented!<sup>1</sup>

## 2 User Manual

In order to run a TinyTL transformation on an XML file, there are three command line parameters to be specified:

1. The XML input document,
2. the TinyTL program, and
3. the name of the XML output document.

Now, in order to launch the transformation browse to `TinyTL\dist` and issue:

```
java -jar TinyTL.jar <XML_input> <TTL_program> <XML_output>
```

This will apply the TinyTL program (argument `<TTL_program>`) on the XML input file (argument `<XML_input>`) and generate an XML output to the specified file (argument `<XML_output>`).

## 3 Architecture

### 3.1 Overview

The input files are an *XML<sup>-</sup>* and a TinyTL document. From both documents a JDOM tree structure is being extracted before they are passed to their corresponding parser (JDOM2XMLParser respectively JDOM2TTLParser). The two parsed files, a parsed XML document and a TinyTL program, contain the abstract structure of their corresponding input files. This is the well-known procedure of data binding in object oriented programming. Finally, the interpreter takes these two parsed files as input and outputs a new XML document.

---

<sup>1</sup>All the methods were manually documented, including the private fields and methods.

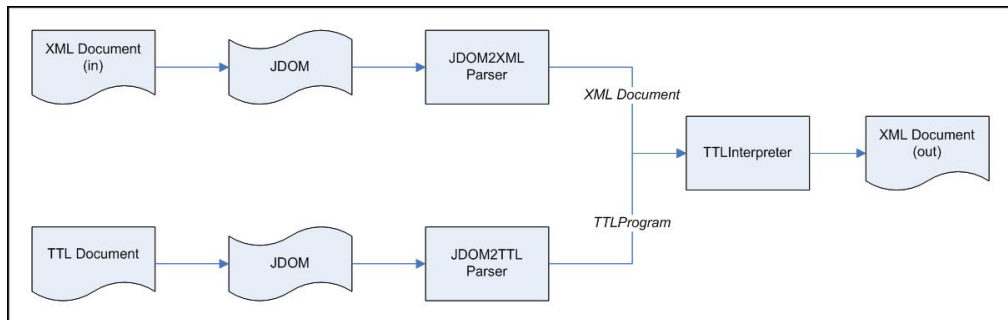


Figure 1: The flowchart of the application

### 3.2 Given Components

We were given the `JDOMTranslator` class transforming a given XML file into a JDOM tree structure. Further we received the classes representing the XML parser. In particular, the `JDOM2XMLParser` doing the data binding by parsing the JDOM tree structure and finally transforming it into a new (parsed) XML document.

Another given class is the `Writer` class, which writes the output received from the interpreter into an XML document.

### 3.3 TTLParser

The **TinyTL document** is written in the syntax of XML. It contains two main parts:

1. The main template starting the transformation by calling the main pattern.
2. The **patterns** containing the rules to be applied on the elements of the XML input document.

**The three main transformation rules are as follows:**

1. Creating a new element that didn't appear in the input XML file.
2. Copying an element from the input to the output file.
3. Dropping an element of the input file such that it won't appear in the output file anymore.

**TTLParser** The input to the TTLParser is a JDOM tree structure based on the given TinyTL input document. The TTLParser is performing the data binding by parsing the given JDOM tree structure with respect to the given abstract syntax of TinyTL.

For more details, please refer to the JavaDoc from the source code.

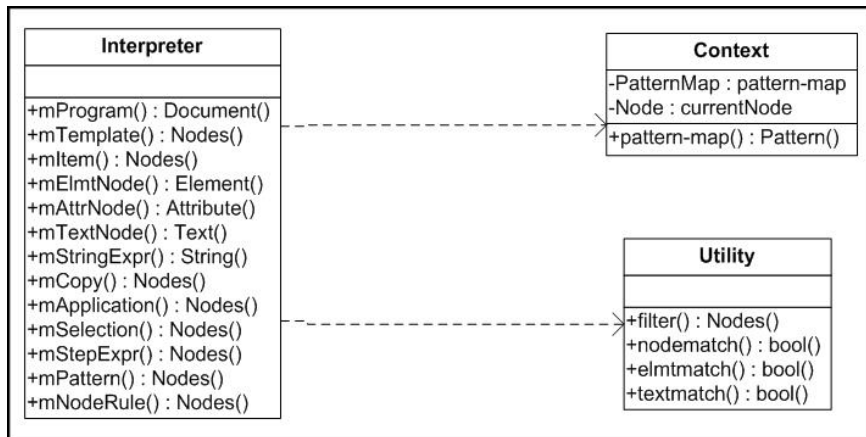


Figure 2: The interpreter’s class diagram

### 3.4 TTLInterpreter

#### 3.4.1 Basic Principles

**Transforming the dynamic semantics of TinyTL into Java code** Basically, we translated the given specifications into Java code. However, it was not always as easy and obvious as it seemed at first glance.

To separate the different types of methods, we defined three classes:

1. The class `TTLInterpreter` containing all the core methods of the interpreter,
2. the class `Context` containing the method `pattern-map` to build the context, and
3. the class `Utility` containing so called helper-methods such as `filter()`, `nodematch()` and others.

#### 3.4.2 Example Translations

##### Translation of MTemplate

```

[1] MTemplate : Template → (Context → Node*)
[2] MTemplate (t : Template) =△=
[3]   λ c : Context •
[4]     over t.items apply
[5]       λ i : Item • MItem(i)(c)
[6]     combine ++ empty <>
  
```

The first line tells us, that the function `MTemplate` takes a `TTLTemplate`, which is a list of `TTLItems`, and the context as input. In the end it outputs a list of `XMLNodes`. Line 4 states that on all items of the given template  $t$ , the function `MItem` (line 5) with the current item and the context is called. The function `MItem` returns a list of `XMLNodes`. The last line states that all these

returned lists of XML nodes will be appended to each other. This final list of XMLNodes is returned by the function `MTemplate`.

**Recursive functions using FIRST and TAIL** We didn't implement this kind of functions recursively. Instead, we used an iterator on a list and a while loop (`while(iterator.hasNext())`). Let's look at an example:

```
[1] MSelection : Selection → (Node* → Node*)
[2] MSelection (s : Selection) =△=
[3]     λ nodes : Node* •
[4]         if s.EMPTY then nodes
[5]         else MSelection(s.TAIL) (MStepExpr(s.FIRST)(nodes))
[6]         end
```

This function takes as input a `TTLSelection` and an arraylist of `TTLNodes`. Recall that a `TTLSelection` is a list of `TTLStepExpressions`. As we can see in the last line, the function is calling itself recursively, each time with the `TAIL` of the list of `TTLStepExpressions`, until the list is empty (line 4). When the list is empty, it returns a list of `XMLNodes`. At every recursion, the `TTLStepExpr` is being created by calling the function `MStepExpr`.

The recursion is actually an iteration through the list of `TTLStepExpressions`.

## 4 Validation Functions

The program `TinyTL` assumes that the `TTL` input files are valid. There are no validation functions implemented in our code, instead they will be discussed in this section.

The validation of a program can be done statically and/or dynamically. In our case, we will only perform the static validation. This means, we will check if the static semantics are respected by the `TTL` program. In other words, we will apply checking-rules that do not need to be aware of the current context (i.e. XML input document).

As an example consider the case of a Java program. The Java compiler verifies the static semantics of the Java program, whereas the Virtual Machine is in charge of checking the dynamic semantics. As a concrete example, let's consider the case of an array in Java. Statically it is possible to verify that the size of the array is non-negative. Dynamically, the Virtual Machine ensures that no instruction references an index that goes beyond the size of an instance of the array.

The following functions validate the static semantics of the `TinyTL` program. From a static point of view we can validate both the `TinyTL` program semantics and some parts of the XML output semantics. In order to distinguish the `TinyTL` validation rules from the XML validation rules, they will both be marked using the tags `<program>` respectively `<output>`.

## 4.1 Program

**Problems** For a program to be valid, the following conditions must hold:

1. It must consist out of a valid list of patterns and a valid main template. `<program>`
2. There can be only one item in the main template. `<program>`
3. The main template can't be of type `TextNode`, because the root in the XML output document must be of type `Element`. `<output>`
4. The first `StepExpression` executed by the program can't be of the direction "up" as we are at the root, and obviously it is not possible to go up beyond the root. `<program>`

### Solution idea

1. The first problem will be handled by the validation functions for patterns and templates. For the sake of simplicity the call to these functions won't be mentioned in the formalisation. This task could be delegated to a main function in charge of checking the global validity by calling the underlying subfunctions.
2. We need to check that the list of items of `p.main` is of length 1.
3. We assume that there exists an operator `is` returning whether an entity is of the given type.
4. A full solution of this problem would need a scan of all the nodes in order to find the first `StepExpression` being executed. This can be done, but is a very resource consuming task, since the first `StepExpression` can be enclosed in several elements. It is very hard to know in advance where it is located.

As an outline of the solution we provide a validation function that checks two cases:

- (a) If the main template is directly an `Application`, it checks that the first `StepExpression` won't indicate a direction "up".
- (b) If the main template is an `ElmtNode` and its template is of type `Application`, it checks that the first `StepExpression` in this `Application` isn't performing an "up" operation.

### Formalisation

```
VProgram : Program → B
VProgram [p : Program] = Δ =
  if (! p.main.items.LENGTH == 1) then false
  else if (p.main.items.FIRST is TextNode) then
    false
  else if (VBeyondRoot(p.main.items)) then false
  else true
end
```

```

VBeyondRoot : Items → B
VBeyondRoot [i: Items] = Δ =
  if (i.FIRST is Application) then
    if (i.FIRST.select.steps.FIRST.dir is Up) then true
    end
  else if (i.FIRST is ElmtNode) then
    if (i.FIRST.content.items.FIRST is Application)
      if (i.FIRST.content.items.FIRST.select.steps.FIRST.dir
        is Up)
        then false
      end
    end
  end
end

```

## 4.2 Pattern

**Problems** A pattern is valid if it respects the following rules:

1. There shall not be two patterns bearing the same name. <program>
2. A pattern that is called must exist. <program>
3. It must contain a list of valid NodeRules. <program>

**Solution idea**

1. We could solve this problem by creating a list of all the pattern names and verifying that no name appears twice. In this case, in the function `pattern-map` of the `TTLInterpreter`, we could use the “++” operator in order to append the new (non-existing) pattern name to the list. However, we do not need to create such a verification since the dynamic semantics our interpreter is based on uses the dissymmetric union to create an initial list of patterns. That is: every new pattern bearing the same name as a pattern that is already in the list will override the existing one. It is worth noting that our final Java program implements the dissymmetric union by using a `Hashtable` which does not accept duplicate keys and thus, enforces this verification.
2. The second problem will be verified in the function `VApplication`.
3. The validity of the given `NodeRules` is checked by the function `VNodeRule`.

## 4.3 Template

**Problem** A template is valid if it consists of a list of valid items.

**Solution idea** The problem will be verified in the function `VItem`.

## 4.4 Item

**Problem** An item is valid, if it consists of a valid `ElementNode`, `TextNode`, `Copy` or `Application`. <program>

**Solution idea** We need to define a function checking the item's type and calling the appropriate validation function.

#### Formalisation

```
VItem : item → B
VItem [i : item] =  $\Delta$  =
  case i of
    ElmtNode ⇒ VElmtNode(i)
    TextNode ⇒ VTextNode(i)
    Copy ⇒ VCopy(i)
    Application ⇒ VApplication(i)
  end
```

### 4.5 NodeRule

**Problem** A NodeRule must satisfy the following conditions:

1. It must contain a valid NodeExpression, <program>
2. as well as a valid Template. <program>

**Solution idea** We call both underlying validation functions and check that both of them return true.

#### Formalisation

```
VNodeRule : NodeRule → B
VNodeRule [n : NodeRule] =  $\Delta$  =
  VNodeExpr(n) and VTemplate(n)
```

### 4.6 NodeExpression

**Problem** A NodeExpression must be either a valid ElmtList, or a valid AllElmts, or a valid AllTexts or a valid AllNodes. <program>

**Solution idea** We check the type using a case-statement.

#### Formalisation

```
VNodeExpr : NodeExpr → B
VNodeExpr [n : NodeExpr] =  $\Delta$  =
  case n of
    AllElmts ⇒ true
    AllTexts ⇒ true
    AllNodes ⇒ true
    ElmtList ⇒ VElmtList(n)
  end
```

## 4.7 ElmtList

The function `VElmtList` always returns true since any list of identifiers is semantically correct from a static point of view.

## 4.8 StepExpr

### Problem

1. One might think, it could be a problem if a `StepExpr` contains more than one of the same identifiers. `<program>`
2. Performing a “down” operation on a `TextNode` is not permitted. `<program>`

### Solution idea

1. This does not create any problems, neither statically nor dynamically because only the first matching expression will be considered.
2. The problem appearing when performing a “down” operation at the level of a `TextNode` is checked during the function `VSelection`.

## 4.9 Direction

**Problem** The direction should be either an “up” or a “down” and nothing else. `<program>`

### Formalisation

```
VDirection : Direction → B
VDirection [d : Direction] = Δ =
  case d of
    Up ⇒ true
    Down ⇒ true
  end
```

## 4.10 Application

**Problem** For an application to be valid it must respect the following rules:

1. It must be composed out of a valid Identifier and a valid Selection. `<program>`
2. A pattern that is called has to be defined in the document. `<program>`

### Solution idea

1. The verification of the Selection is done by a method `VSelection`.
2. We take the list of all the patterns defined in the document (`program.patts`). At every Application we call the function `VApplication` verifying if the pattern called exists in the list `program.patts`.

**Formalisation** This function is called at the very beginning of the document validation. It returns a new kind of object `InitPattsList` containing all the names of the patterns defined in the document.<sup>2</sup>

```

CreateInitPattsList : Program → InitPattsList
CreateInitPattsList [prog : Program] = Δ =
  over prog.patts apply
    λ patt : Pattern • patt.name
  combine ++ empty <>

VApplication : InitPattsList x Application → B
VApplication [ initPattsList : InitPattsList, app : Application]
  = Δ =
  if (app.name ∈ initPattsList) then true else false

```

## 4.11 Selection

**Problem** A Selection contains a list of StepExpressions. Each of them contains a direction (up or down) and a match-NodeExpression. If there is a match-NodeExpression matching “AllTexts” (= \$), the following StepExpression is not permitted to do a “down” operation, because we are already on the lowest level (on the leaves) of the XML hierarchy. <program>

**Solution idea** We need to check that, if we are on the level of TextNodes (leaves), we won’t perform a “down” operation in the following StepExpression.

### Formalisation

```

VSelection : Selection → B
VSelection[s : Selection] = Δ =
  VDown(s.steps)

VDown : StepExpr* → B
VDown [steps : StepExpr*] = Δ =
  if (steps.EMPTY) then true
  else if (steps.FIRST.matches == "$") then
    if (! steps.TAIL.EMPTY) then
      if(steps.TAIL.FIRST.dir == ‘down’) then
        false
      end
    end
  else
    VDown(steps.TAIL)
  end

```

## 4.12 Copy

**Problem** For a Copy to be valid, it must contain a valid template.

<sup>2</sup>This scheme is similar to the variables’ initialisation we used for GRAAL, except that we do not create a map, but a list.

**Solution idea** The validation will be done by the function `VTemplate`.

### 4.13 TextNode

**Problem** A `TextNode` can only contain a `StringExpression`. `StringExpressions` are always valid from a semantics' point of view.

### 4.14 AttrNode

The function `VAttrNode` always returns true since any `StringExpr` and any `Identifier` is semantically correct from a static point of view.

### 4.15 ElmtNode

**Problem** An `ElmtNode` needs to respect the following rules:

1. It must contain an identifier and a list of valid `AttrNodes`. `<program>`
2. It must not contain one or more attributes with the same name, otherwise the XML output document would be invalid. `<output>`

**Solution idea** We create a list of all the attribute names appearing in the element. Then we use another function to check whether every single attribute name appears twice or more in the list.

#### Formalisation

```
VElmtNode : ElmtNode → B
VElmtNode [e : ElmtNode] = Δ =
  (! IsAlreadyInList(AttrNamesList(e.attrs)))

AttrNamesList : AttrNode* → Identifier*
AttrNamesList [attrs : AttrNode*] = Δ =
  over attrs apply
    λ a : AttrNode • a.name
  combine ++ empty <>

IsAlreadyInList : Identifiers* → B
IsAlreadyInList [l : Identifiers*] = Δ =
  if (l.FIRST ∈ l.TAIL)
    then true
  else IsAlreadyInList(l.TAIL)
  end
```

## 5 Examples

There are two examples listed in the appendix A of this report.

## 5.1 Creating a CSV (Comma Separated Values) File

Given an address book as XML input file, the corresponding TinyTL program extracts all the information given in the address book and outputs a CSV (Comma Separated Values) file containing the extracted information.

## 5.2 Creating an HTML Web Page

This example uses as well the address book as input file. The TinyTL transformation outputs a HTML file displaying the information given by the address book. Please note that the output file should be given an .html extension. If it is given an .xml extension, an XML file is created containing the HTML code.

# 6 Conclusion

**Feedback on the project** Although it was not always easy to figure out the tasks and their solutions, this project helped us a lot to understand the subject of this lecture. It forced us to plunge into the course material and to hang in there until we understood the subject matter.

**Thank you** Many thanks for the helpful and kind support given by the professor and in particular by the assistant!

## A Examples

### A.1 The XML Input File (input.xml) used by both Examples

```
<addrbook>
<contact id="1">
<name>[Paul Dupont]</name>
<email>[paul@dupont.com]</email>
<phone>[123 456 789 ]</phone>
</contact>

<contact id="2">
<name>[Pierre Durand]</name>
<email>[durant@company.com]</email>
<phone>[452 351 789 ]</phone>
</contact>

<contact id="3">
<name>[Sylvie Dunand]</name>
<email>[sylvie.dunand@company.com]</email>
</contact>

<contact id="4">
<name>[Chantal Despont]</name>
<email>[chantal@despont.net]</email>
<phone>[999 111 333]</phone>
</contact>
</addrbook>
```

### A.2 The CSV Example

#### A.2.1 The TinyTL Input Program csv.ttl

```
<tinytl>

  <!-- main template -->
  <elt name="csv">
    <attr name="extractionDate" value="30.06.06"/>
    <apply pattern="selectAllText">
      <select>
        <step direction="down" match="*" />
      </select>
    </apply>
  </elt>

  <!-- pattern 1 : selectAllText -->

  <pattern name="selectAllText">
    <rule match="*">
```

```

        <apply pattern="selectAllText">
            <select>
                <step direction="down" match="#" />
            </select>
        </apply>
    </rule>
    <rule match="$">
        <copy/>
        <text content=";" eol="false" />
    </rule>
</pattern>

</tinytl>

```

### A.2.2 The corresponding XML<sup>-</sup> Output File csv.xml

```
<csv extractionDate="30.06.06">
```

```

    [Paul Dupont]
    [;]
    [paul@dupont.com]
    [;]
    [123 456 789 ]
    [;]
    [Pierre Durand]
    [;]
    [durant@company.com]
    [;]
    [452 351 789 ]
    [;]
    [Sylvie Dunand]
    [;]
    [sylvie.dunand@company.com]
    [;]
    [Chantal Despont]
    [;]
    [chantal@despont.net]
    [;]
    [999 111 333]
    [;]
</csv>

```

## A.3 The HTML Example

### A.3.1 The TinyTL Input Program createWebPage.ttl

```

<tinytl>
    <elt name="html">
        <elt name="body">
            <elt name="h1">

```

```

        <text content="Online Phone Book" eol="false"/>
    </elt>
    <apply pattern="getContact">
        <select>
            <step direction="down" match="contact"/>
        </select>
    </apply>
    <elt name="p">
        <text content="(C) Karin and Dominique, UNIFR 2006" eol="false"/>
    </elt>
</elt>
</elt>

<pattern name="getContact">
    <rule match="contact">
        <elt name="p">
            <apply pattern="getContactContent">
                <select>
                    <step direction="down" match="name|phone|email"/>
                </select>
            </apply>
        </elt>
    </rule>
</pattern>

<pattern name="getContactContent">
    <rule match="name">
        <elt name="b">
            <apply pattern="copyText">
                <select>
                    <step direction="down" match="$"/>
                </select>
            </apply>
        </elt>
    </rule>
    <rule match="email">
        <elt name="tt">
            <apply pattern="copyText">
                <select>
                    <step direction="down" match="$"/>
                </select>
            </apply>
        </elt>
    </rule>
    <rule match="phone">
        <elt name="u">
            <apply pattern="copyText">
                <select>
                    <step direction="down" match="$"/>
                </select>
            </apply>
        </elt>
    </rule>
</pattern>

```

```

                </apply>
            </elt>
        </rule>
    </pattern>

    <pattern name="copyText">
        <rule match="$">
            <copy/>
        </rule>
    </pattern>

</tinyt1>

```

### A.3.2 The corresponding HTML Output File test.html

Opened in a browser it will look something like this:

```

[Online Phone Book]

[Paul Dupont] [paul@dupont.com] [123 456 789 ]

[Pierre Durand] [durant@company.com] [452 351 789 ]

[Sylvie Dunand] [sylvie.dunand@company.com]

[Chantal Despont] [chantal@despont.net] [999 111 333]

[(C) Karin and Dominique, UNIFR 2006]

```

If the output file is given an .xml extension, the output will contain the following lines:

```

<html>
<body>

<p>
<b>
    [Paul Dupont]
</b>
<tt>
    [paul@dupont.com]
</tt>
<u>
    [123 456 789 ]
</u>
</p>

<p>
<b>

```

```
    [Pierre Durand]
  </b>
<tt>
  [durant@company.com]
</tt>
<u>
  [452 351 789 ]
</u>
</p>

<p>
<b>
  [Sylvie Dunand]
</b>
<tt>
  [sylvie.dunand@company.com]
</tt>
</p>

<p>
<b>
  [Chantal Despont]
</b>
<tt>
  [chantal@despont.net]
</tt>
<u>
  [999 111 333]
</u>
</p>

</body>
</html>
```

## **B Abstract Syntax XML<sup>-</sup> and TinyTL**

The specifications can be found in the enclosed document `abstractSyntax.pdf`.

## **C Dynamic Semantics TinyTL**

The specifications can be found in the enclosed document `dynamicSemantic.pdf`.