Intermediate Notation

1. Introduction

The architecture of the application about UML2Code Generation for Wireless sensor networks requires defining an intermediate notation or set of rules that allow to transmit the information from top to bottom layer and vice versa. To be defined such a notation it is necessary to be addressed several key factors:

- **Common Types -** collect possible type definitions, specify how type is defined and specific possible values. Defines the rules that ensures that the data types are written in an enough concrete level that reduces possibility of ambiguity.
- **Hierarchical order –** the structure of the notation was supposed to be hierarchical or tree type. That was necessary in general because nested elements are supposed to be accessed using a notation. List of properties must be transmitted through it and they must be define in a short form. Moreover, good intermediate notation must be self-explanatory, readable and with finite number of elements. To be able to represent complex hierarchical structures in a simple way.

2. XML as an intermediate notation

Given the fact that intermediate notation requires hierarchical order and structure and that project is going to converts UML to some other languages the use of Extensible Markup Language (XML) is appropriate as the underlying convention and transporting information, which after all was the initial idea of the language. [1] The main advantage of XML is the ability of user to create tags and the document structure. The other huge advantage is the ability to extract information using standardized XML parsers embedded in almost every modern object oriented language such as C# and Java.

3. XML schema

1. Motivation

According to W3Schools [2], XML schema is to define the legal building blocks of an XML document. The main goals of the schema is to reduce error possibility of block appearance, attribute appearance and unknown elements and values. In other words, our schema:

- ✓ XML schema defines elements, attributes and the order that they must appear in a particular document.
- ✓ XML schema defines which block is about to has child elements, they order and attributes.
- ✓ XML schema defines when is possible some of the attributes to be empty, data types and possible expected (default) values.
- ✓ XML schema defines which attributes and elements are compulsory and which are optional.

In order to anticipate changes the schema must be as abstract as possible but concrete enough to does allow ambiguity. In other words, our XML schema:

- ✓ XML schema allows additional parameters.
- ✓ XML schema allows easy modifications.
- ✓ XML schema is written in XML.
- ✓ XML schema supports data types, objects, components and other structures.

Many of the requirements concern different levels of validation and respectively verification of the data. One of the biggest advantages of XML schema is ability to describe allowable content, validate data correctness, put restrictions over it and describe precisely data conversion if it is required.

2. XML schema rules

To achieve good quality and to describe the schema syntax, the basic IM example will be presented.

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<interface>
    <attribute  name = "ITempReader" />
    <channel type = "bool" name = "tRequestChan" direction = "out"/>
      <channel type = "real" name = "tValueChan" direction = "in"/>
      <channel type = "any" name = "stdoutChan" direction = "out"/>
</interface>
<component>
    <attribute  name = "TempReader" />
      <presents name = "ITempReader"/>
      <field></field>
      <constructor>
      </constructor >
      <behaviour>
            <send identifier = "true"   on    = "tRequestChan"/>
            <receive  identifier = "temp" from = "tValueChan"/>
            <send identifier =  "any"  value = '"\nTemp:"'  on    = "stdoutChan"/>
            <send identifier =  "any"  value = ""  on    = "stdoutChan"/>
       </behaviour>
</component>
<connect >
      <from name = "TempReader"                    on = "tRequestChan"/>
      <to   name = "LightHumidTempSensor" on = "tempRequest"/>
      <from name = "TempReader"              on = "tValueChan" />
      <to   name = "LightHumidTempSensor" on = "tempOutput"/>
      <from name = "TempReader"                    on = "stdoutChan" />
      <to   name = "standardOut"                   on = ""/>
</connect>
</xsd:schema>
```

The following fragment:

```
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
```

Indicates that the elements and data types used in the schema partially comes from that particular namespace. Normally, the second part specifies that prefix **xsd** should be put with all elements and data types, but in that case this is optional.

3. Defining the common types

- ✓ xsd:integer
- ✓ xsd:bool
- ✓ xsd:real
- ✓ xsd:unsigned integer
- ✓ xsd:array
- ✓ xsd:string

4. **Defining the computational units:**

- ✓ **Interface element definition** – the syntax for defining an interface is:

```
<interface>
    <attribute  name = "ITempReader" />
    <channel type = "bool" name = "tRequestChan" direction = "out"/>
   <channel type = "real" name = "tValueChan" direction = "in"/>
</interface>
```

where attribute element is with compulsory parameter name, which is the name of the interface. The next child elements are channels.

- ✓ **Channel element definition:**

```
<channel type = "bool" name = "tRequestChan" direction = "out"/>
```

Where the syntax of the element is **type**, **name** and **direction.** Some of the attributes are normally optional but in that case all three are **compulsory**. The **direction** values are **fixed** – in and out. There is no any other option.

- ✓ **Component element definition:**

```
<component>
 <attribute  name = "TempReader" />
<presents name = "ITempReader"/>
<field></field>
<constructor>
</constructor >
<behaviour>
      <send identifier = "true"   on   = "tRequestChan"/>
      <receive  identifier = "temp" from = "tValueChan"/>
      <send identifier =  "any"  value = '"\nTemp:"'  on   = "stdoutChan"/>
 </behaviour>
</component>
```

Component is the main computation and logical unit in the intermediate notation. It is an extensive and complicated data three that contains a lot of child nodes. The attribute element is compulsory and describes the name of the component.

- ✓ **Presents element definition:**

```
<presents name = "ITempReader"/>
```

Presents element syntax contains the attribute **name** that defining the interfaces this component is going to implement. It is a **compulsory element and** the attribute **name** is also compulsory. One component element is **possible** to have more than one present elements. *The correct syntax requires* **presents** to be immediately after attribute name element of the component.

- ✓ **Field element definition:**

```
<field  type = "integer" name = "solar"/>
<field type = "" name = "avgTemp" value ="0.0"/>
```

Field element is following **immediately** after **presents** element. The element contains three attributes **type**, **name** and **value**. The **correct** order of the attributes is **type**, **name** and **value**. Two of the attributes **are optional** – type and value. **Name** is compulsory attribute for **field** element to be accepted as correct and valid.

✓ **Constructor element definition:**

```
<constructor>
</constructor >
```

The constructor element is mainly responsible for setting the data of the component. There is no attributes for the constructor element. In the body, there are possibility for **field, variable, if, else, send, receive, expression, etc**.

✓ **Behaviour element definition:**

```
<behaviour>
</behaviour>
```

Behaviour element is the main computational body of the **component**. Similarly to the **constructor** element it can contains a lot of child elements. There are no attributes that describe the **behavior** element.

✓ **Send element definition:**

```
<send identifier = "true"   on    = "tRequestChan"/>
<send identifier =  "any"  value = '"\nTemp:"'  on    = "stdoutChan"/>
```

Send element is described as three attributes. **Identifier**, **value** and **on**. Two of the attributes are **compulsory** – **identifier** and **on. Value** is optional, **except** in a case of **functional call any**. The correct order for **send element** is **identifier**, **value** and **on**.

✓ **Receive element definition:**

```
<receive  identifier = "temp" from = "tValueChan"/>
```

Receive element is described as two attributes. **Identifier** and **from**. Both of the attributes are **compulsory** and the correct order is **identifier** and **from.**

✓ **Instance element definition:**

```
<instance component="TempReader" name="tr" />
```

Instance element is described by two attributes. The attribute **component** from which the instance is going to be created and **name** that specifies the name of the instance. The correct order is **component** and **name**.

✓ **Connect element definition:**

```
<connect >
      <from name = "TempReader" on = "tRequestChan"/>
      <to   name = "lightHumidTempSensor" on = "tempRequest"/>
 </connect>
```

Connect element described two **compulsory** elements – **from** and **to.** There is no need of **connect** element if **each** of this two is missing. The correct order of the children is pair of first **from** and second **to.**

✓ **From element definition:**

```
<from name = "TempReader" on = "tRequestChan"/>
```

From element contains two attributes **name** and **on**. Both of them are compulsory. The correct order of the attributes is first **name** and second **on. From element** cannot exists without **to element.**

✓ **To element definition:**

```
<to   name = "lightHumidTempSensor" on = "tempRequest"/>
```

To element contains two attributes **name** and **on**. Both of them are compulsory. The correct order of the attributes is first **name** and second **on. To element** cannot exists without **From element preceding it.**

✓ **Print element definition:**

```
<print variable = "cycle"   type = "Int"/>
```

Print element contains two attributes. Both of them are compulsory and the correct order is **variable** and **type. Type** attribute is necessary to determine which print element is going to be used.

✓ **Struct element definition:**

```
<struct>
    <attribute name = "sensorRading" />
    <field  type = "integer" name = "solar"/>
</struct>
```

Struct element contains one attribute **name**. The attributes contains the name of the **struct** and it is compulsory. The correct syntax **requires** the name attribute to be the first child node of the **struct**. The next children could be **field elements.**

✓ **Procedure element definition:**

```
<procedure type="real">
        <attribute  name = "tempIntToCelsiusReal" />
        <attribute  name = "parameters">
                <field  type = "integer" name = "reading"/>
        </attribute>
        <body>
        </body>
        <return>
                <expression value ="-45.3 + 0.01 * reading"/>
        </return>
</procedure>
```

Procedure is represented by the tag **procedure**. There is an attribute **type** which could be empty or filled with integer, real, bool or void. In a case of **void** it could remain empty. The first **child tag** is the attributes contains the name of the procedure. It is compulsory. Next if the procedure has parameters or **arguments** must be attribute tag with the **name of the argument**. The next tag is **body tag**. It could contain **expression**, **field**, **return**, **if**, **for**, **print** and **variable** tags. The correct order of the procedure syntax is **procedure** with **children –** attribute name, attribute parameters (optional), body and return (optional).

✓ **Return element definition:**

```
<return>
        <expression value ="-45.3 + 0.01 * reading"/>
</return>
```

Return tag is optional for the procedure if the type is **void** and compulsory **in any other case.** One procedure could have **more than one** return tags. The last one outside the body is compulsory. The return tag **has** only one child **expression tag** that returns particular value.

✓ **Variable element definition:**

```
<variable name = "printOutput"   new = "true"/>
<variable type = "" name = "photoValue" new = "true" bindingTo =
"photoReading()"/>
```

The **variable tag** is describe with several attributes. The **type** attribute is optional. It contains the type of the variable, the **name** attribute is compulsory and contains the name of the

variable. Next tag **new** is optional in a case of static declaration, otherwise it is compulsory to notify that allocation is going to be dynamic. **BindingTo** is compulsory tag that points from how the variable is described in other case the type of the variable. The correct order of attributes is **type**, **name**, **new**, **bindingTo** and **value**.

- ✓ **For element definition**

```
<for counterName ="i" value ="0" iterateThrough="ints">
        <expression leftOperand ="ints" index ="i" rightOperand="ints.length -
1" operation="assign" />
      <expression leftOperand ="last" rightOperand="last + 1"
    operation="assign" />
     </for>
```

For element tag describes for loop. It has **three compulsory** attributes **counter name**, **value** and **iterate through**. The correct order is as they are listed. Within **for element tag** there could have **variable, field, if, else, return** and **print** tags.

5. Attributes representation specifics

Most of the types in the intermediate notation are complex. That means that they are described by attributes. The attributes can be presented internally within the element structure, for example:

```
<for counterName ="i" value ="0" iterateThrough="ints">
```

Or externally, outside the element structure, for example:

```
<struct>
    <attribute name = "sensorRading" />
```

When an XML element or attribute has a specific compulsory attribute such as name, it puts restrictions on the element's or attribute's content. Without it or without the correct order, the intermediate notation is not correct.

4. Examples and validation

There are several other examples that could be found on the address:
https://github.com/avasilsg/InsenseGenerator/tree/master/InsenseCodeGenerator/src/IntermediateNotation
The purpose of these examples is to verify, test and validate the correctness of the generate Insense source code as well as the sufficient and applicability of the proposed intermediate notation. Several web validator XML sites are used to prove the validity of the XML syntax. The main one is http://www.xmlvalidation.com/ .

5. Conclusion

Overall the schema and whole intermediate notation is understandable, clear and easy to transmit information. It is abstract and open enough to be modified and extended but concrete enough to reduce ambiguity. The validation follows the rules described in every tag such as hierarchical order, attributes and possible values. As a future improvement it could be suggested to get into deeper detail especially for the variables and more complicated expressions.

6. References:

1. W3Schools, [online] http://www.w3schools.com/xml/xml_whatis.asp visited 26.07.2014
2. W3Schools, [online], http://www.w3schools.com/schema/, visited 25.07.2014