6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

Describing Semantics of Data Types in XML

Szabolcs Hajdara[a], Balázs Ugron[b]

Department of Software Technology and Methodology,
Eötvös Loránd University
[a]e-mail: sleet@inf.elte.hu
[b]e-mail: balee@inf.elte.hu

Abstract

For the sake of code reusing, it may be useful to be able to transform program code between programming languages, for which a common format can be used. We chose XML because it is widely used. If we want to be able to change between programming languages, the most important thing is handling data types, and the description of some verifiable semantic properties. These constraints can be given in XML, too. We are considering the base issues in this paper, and we suggest solutions for some of these issues. If a solution was given for translating an abstract program to a concrete code then a special code generation technique could be implemented.

Categories and Subject Descriptors: D.1.5 [Programming Techniques]: Object Oriented Programming; D.2.1 [Software engineering]: Requirements specification; D.2.11 [Software engineering]: Software Architectures – *data, abstraction.*

Key Words and Phrases: code reusing, data types, semantics of data types, program transformation, XML.

1. Introduction

Notion of abstract data types ([1], [2], [3], [4]) is very important from the point of view of software designing. Different methodologies that support clear description of data types make designing easier, however a correct, formal way of specification may be favourable beside the clear description, because of different reasons. In the context of usability, it is considerable to choose some generally used, well known formalism for property representing. One of the most widely supported descriptive language is XML ([6]).

Nowadays, you are expected to be able to formulate different relations between data types. We mean different types of inheritance or association relationships.

If we are made familiar with one of the possible specification methodologies of data types, it would be advantageous to improve this theoretical approach in the direction of taking the opportunity to use these results in the subject of practical software designing. The representation of data types in XML may be a middle state of different stages of appearance of specification. This presents an opportunity to generate a program code skeleton from the model in a quite general way. Certain difficulties may come to the surface while you are trying to make an XML representation for properties and relations.

One of our main goals is to be able to deduce some properties of concrete data types from the properties that are defined in the specification of the associated abstract data types – generating the most possible code is of first importance. Describing concrete data types written in different programming languages seems to be a good starting-point. So in this paper we are considering the description of different information in XML about data types written in some programming language.

2. Describing Data Types in XML

The most issues related describing data types are caused by inheritance and the consequenced polimorphism and late binding. A relation "ancestor" is introduced for describing inheritance. This relation shows an other data type, the properties of which the child data type inherits. Besides, a meta class is introduced, which shows the programming language the type is written in.

Let us consider the following example:

```
class A {
  public:
    virtual void a();
    void b(bool c);
};

class B : public A {
  public:
    virtual void a();
};
```

One XML description of the above code is the following:

```
<class meta="yes" name="C++ Class"></class>

<class name="A">
  <ancestor>C++ Class</ancestor>
  <function visibility="public" type="void" virtual="yes"
    name="a">
  </function>
```

```
  <function visibility="public" type="bool" virtual="no"
    name="b">
    <parameter type="bool" name="c">
    </parameter>
  </function>
  <property visibility="protected" type="int" name="p">
  </property>
</class>

<class name="B">
  <ancestor modifier="public">A</ancestor>
  <function visibility="public" type="void" virtual="yes"
    name="a">
  </function>
</class>
```

Although our description language (see below) should be used for describing rules for synthesizing a concrete code from the XML code, we will show an example for describing a static semantic property because an example for the original aim would be too complicated. This mentioned description language can be applied similary for representating the polimorphism and the late bindig language independently.

3. Static semantic

Even if a XML description of an abstract data type is available, we may need to describe some static semantic properties ([5]). For instance, such property may be a constraint for a number or the type of the parameters of a given method. This kind of constraints may be needed if we are going to transform a program code described in XML to some target language. In this case we may wish to check, whether the XML file satisfies the static semantic constraints, that is, whether the program in the target language can be generated without certain type of errors.

3.1. Example

Let us consider the following example written in Java:

```
class JavaClass {
  void writeString(String str) {
    System.out.println(str);
  }
}
...
  JavaClass jc = new JavaClass();
  jc.writeString("Hello World!");
```

A simplified XML description of the above code is the following:

```
<class name="JavaClass">
  <function name="writeString">
    <parameter name="str">
      <type>String</type>
    </parameter>
    <body>
      <use-instance name="System.out">
        <use-function name="println">
          <use-parameter><parname>str</parname></use-parameter>
        </use-function>
      </use-instance>
    </body>
  </function>
</class>
...
  <instance class="JavaClass">jc</instance>
  <create-instance name="jc" type="JavaClass">
  </create-instance>
  <use-instance name="jc">
    <use-function name="writeString">
      <use-parameter>
        <data type="String">Hello World!</data>
      </use-parameter>
    </use-function>
  </use-instance>
```

This XML code may be "faulty" in the sense for example if the parameter number of a method differs at the calling and at the declaration.

We remark, that it is possible, that even the name of a method differs at the calling and at the declaration, but this and the similar syntactic errors can be found out quite simply, so we do not consider these kind of errors.

4. Describing static semantics in XML

We defined static semantic properties in XML (similar description can be found in [10]). The DTD of the language that we used is the following:

```
<!ELEMENT stat-sem ( sem-rule* )>
<!ELEMENT sem-rule ( op )>
<!ATTLIST sem-rule app NMTOKEN #REQUIRED>
<!ELEMENT op ( attribute, attribute )>
<!ATTLIST op name ( eq | leq | geq )>
<!ELEMENT attribute ( child-count | first-parent-by-name )>
```

```
<!ATTLIST attribute element NMTOKEN>
<!ELEMENT first-parent-by-name ( tag-name )>
<!ATTLIST first-parent-by-name element NMTOKEN #REQUIRED>
<!ELEMENT tag-name ( instance-type )>
<!ATTLIST tag-name attname NMTOKEN #REQUIRED>
<!ELEMENT tag-by-type ( fun-dec-by-class )>
<!ATTLIST tag-by-type element NMTOKEN #REQUIRED
                            attname NMTOKEN #REQUIRED>
<!ELEMENT fun-dec-by-attr ( attribute )>
<!ATTLIST fun-dec-by-attr attname NMTOKEN>
<!ELEMENT child-count EMPTY>
<!ATTLIST child-count element NMTOKEN>
```

The meaning of the elements of the above DTD are:

stat-sem: this is the root element of our language.

sem-rule: a static semantic rule associated with a program segment that is defined by a given tag, where the name of the tag is defined by the "app" attribute.

op: an operator (defined by the attribute name) with two arguments.

attribute: the argument of the operator op.

first-parent-by-name: finds the nearest "element" tag from the immediate descendants of the ancestors of the node given by the "sem-rule" tag.

tag-name: defines the tag that "tag-by-type" will find.

tag-by-type: defines the element tag of which attribute attname is equal with the value returned by "tag-name" and which element tag is the nearest to the actual node.

func-dec-by-attr: returns the node representing the class declaration of the instance defined by "tag-by-type".

child-count: returns the number of the tags that are directly below the given node.

4.1. The semantics of the previous example

The parameter number checking concerned the method call in the example in section 3.1 may be described in XML in the following way:

```
<stat-sem>
  <sem-rule app="use-function">
    <op name="eq">
      <attribute element="use-parameter">
```

```
        <child-count element="data"></child-count>
      </attribute>
      <attribute>
        <first-parent-tag-by-name element="create-instance">
          <tag-name attname="type">
            <tag-by-type element="instance" attname="class">
              <fun-dec-by-attr attname="name">
                <child-count element="paramname"></child-count>
              </fun-dec-by-attr>
            </tag-by-type>
          </tag-name>
        </first-parent-tag-by-name>
      </attribute>
    </op>
  </sem-rule>
</stat-sem>
```

The satisfaction of the static semantic constraints given in this way can be verified by the parser. The parser can be developed easily on the basis of the above description.

5. Using the XML Description

This description language (see above) can be used for describing some special static properties of programs, of course (as in the example). But a special XML description can be given for every language that can help in generating a real program code for the abstract XML code. For example, we can describe, how the virtual or static methods must be compiled or we can describe some properties of the late bindig (similar to the example above).

In a special way this description can be used to make the XML description to a concrete program code, so finally we will be able to compile a program from a language to an another. For this purpose we have to be able to translate a program from its XML description to an arbitrary language and it is not negligible that if a program is given in some programming language then its XML descritpion have to be produced in order to compile this XML description to an other programming language.

We have shown an example, how to use our description language. This description can be used for describing some special properties of data types. Based on this description it is straightforward to make a compiler for translating our XML code to a programming language. But that is quite another thing to generate the XML description of some program written in a real programming language which subject we don't discuss in this paper.

If we can give an XML description for abstract data types that are defined by mathematical equations, than we can sythesize a program code for the math-

ematical specification. So we would be able to generate the program code from the abstract mathematical specification. Let us remark that producing the XML description of the mathematical equations is not trivial. In fact it is very difficult to write a compiler that can translate the XML code containing the representation of the mathematical equations to a program code. It would be an easier way to make a more complex XML description for the equations that contains not only the representation of the equations but some other information would be stored in the XML code. But in this case producing the XML code is more difficult. The translation of an abstract data type to a concrete one through the XML description is an open question.

6. Future work

We gave an XML description for some properties, but we have to produce the XML description of several additional properties. Finally, we have to write the compiler which can translate a program to an another program – written in an another language. That is we have to implement our algorithm.

Starting from the foregoing, we are going to generate the most detailed concrete code that is possible on the basis of the abstract data type – we will write a special code generator.

We have to write the abstract data types in XML. An automated method will be the best that can generate the XML description of the abstract data types that are given with mathematical equations. In fact we should have to compile the equations to XML form.

We have to give a well work method to give the XML description of a program. It is very difficult to do it (the inverse direction is more simplier).

References

[1] L. Kozma: *Absztrakt Osztott Adattípusok Egy Specifikációja*, Alkalmazott Matematikai Lapok 7, pp. 331-344, 1981.

[2] L. Kozma: *Proving the Correctness of Implementations of Shared Data Abstractions*, International Symposium on Programming 5th Colloquium Turin, Lecture Notes in Computer Science Vol. 137, pp. 227-241, 1982.

[3] L. Kozma, Z. Laborczi: *On Implementation Problems of Shared Abstract Data Types*, Conference on Operating Systems Visegrad, Hungary, Lecture Notes in Computer Science Vol. 152, pp. 146-152, 1983.

[4] F. Parisi-Presicce, A. Pierantonio: *An Algebraic Theory of Class Specification*, ACM Transaction on Software Engineering and Methodology, Vol. 3, No. 2, pp. 166-199, 1994.

[5] M. von der Beeck: *A structured operational semantics for UML-statecharts*, Software System Model, pp. 130-141, 2002.

[6] XML home: *http://www.w3c.org/XML*

[7] T. Bray, J. Paoli, and C. Sperberg-MacQueen: *Extendable markup language*, 1998.

[8] G. Psaila and S. Crespi-Reghizzi: *Adding Semantics to XML*, Second Workshop on Attribute Grammars and their Applications, WAGA'99 (Amsterdam, The Netherlands) (D. Parigot and M. Mernik, eds.), INRIA rocquencourt, pp. 113-132, 1999.

[9] Á. Beszédes, Á. Kiss, M. Tarkiainen, R. Ferenc, F. Magyar: *Tool for reverse engineering large object oriented software*, SPLST, 16-27, 2001.

[10] Ferenc Havasi: *XML Semantics Extension*, Acta Cybernetica 15, 2002,

Postal address

Szabolcs Hajdara, Balázs Ugron
Dep. of Software Technology and Methodology
Eötvös Loránd University
XI. Pázmány P. sét. 1/c.
H-1117 Budapest, Hungary