Using XML and XSL to Express Formal Semantics

Karel Richta *
richta@fel.cvut.cz

Abstract: The paper deals with an algebraic specification prototyping via XSL-stylesheet over XML form of this specification. An algebraic specification is converted into XML-format. The signature is expressed as DTD. The semantic part of a specification is converted into XSL-code and used as a rewriting system. The prototyped expression can be formed as an XML-term, which is rewritten with the help of XSL into a canonical form – the meaning of this term.

Key Words: XML, XSL, algebraic specification, formal semantics

1 Introduction

At present it is widely accepted that formal specifications can play an important role in software design. The most important point is that the formal specifications can serve as an unambiguous and strict interface between customer, software analyst, designer, and programmer. Algebraic specifications with its clear syntax and semantics could be used for this purpose. The results of negotiations between the customer, who needs software for solving a certain problem, and a software producer can be declared as the algebraic specification of the given problem.

An algebraic specification consists of two parts: signature and a set of axioms. The signature servers as a definition of the syntax. The axioms specify the semantics. Any algebraic structure satisfying axioms is a so-called model of a specification. The meaning of a specification as the isomorphic class of all initial models – it means, the class of models, whose are exchangeable.

The specification can be prototyped – the problem is to find out automatically any member of the initial class. One possible way is to use a unique symbolic model constructed from Herbrand's universe of all well-formed terms by the smallest congruence relation generated by axioms of a specification. This symbolic model always exists, and if we are able to construct it, the result depicts so-called decision procedure for the equality problem in the defined class.

A decision procedure is usually modeled by term rewriting systems.

The problem is how to construct an appropriate term rewriting system for a given algebraic specification. Most systems are based on the well-known Knuth-Bendix completing procedure, which completes the appropriate term rewriting system for a given specification. If the procedure succeeded, we receive a term rewriting system that solves the equality problem for the specification and we can use it as a base for the construction of a prototype.

This paper discusses the construction of a prototype from a term rewriting system. The key problem in prototyping is to produce a prototype in a short time and at a sufficient level of efficiency for testing. A very simple method is to transform the term rewriting system to Prolog or another similar high level logical programming language [Bergstra 89]. In such a case the process of construction is simple, but the resulting product is directly dependent on the efficiency of a Prolog compiler.

* Dept. of Computer Science & Engineering, FEL ČVUT Praha, Karlovo nám. 13, 121 35 Praha 2
Another possibility is to translate the term rewriting system into a functional language [Prívara 88]. This process is not so straightforward as the above conversion to Prolog, but the resulting product should be more efficient due to an essentially simpler execution. The basic difference is that there is no need for general unification in prototyping (for parameter passing). The sufficient concept is matching of parameters with function definition's equations. Unfortunately, the efficiency is still dependent on the functional language compiler.

Another possible optimization consists of developing an abstract machine dedicated for this purpose. Generally it could be Warren's abstract machine [Warren 83] designed for Prolog and therefore it can also consequently be used for rewriting. But such a choice is not the most efficient one because the code of Warren's machine is designed for unification. However, the matching machine can be made much simpler, see [RN 91].

This paper deals with the idea how to use XSL transformations [W3C] for this purpose. The input algebraic specification is converted into XML-format and expressed as an XML-document. The signature is expressed a DTD (Document Type Definition [W3C]). The semantic part of a specification is converted into XSL-code and used as a rewriting system. The prototyped expression can be formed as an XML-term, which is rewritten with the help of XSL into a canonical form – the meaning of this term.

2 The Algebraic Specification in XML Format

In the following we suppose that an algebraic specification \( S \) is the couple formed by a signature \( \Sigma \) and a finite set of (possibly conditional) equations. The signature \( \Sigma \) of the specification introduces all symbols permissible to denote objects in the defined world. The signature consists of the definition of all sorts of data and declarations of all admissible operations with their arities.

An equation is a pair of well-formed terms \( L, R \) over signature \( \Sigma \), written as \( L = R \). Let \( t_1 \) and \( t_2 \) be well-formed terms of the same sort \( d \) over \( \Sigma \). The expression \( t_1 == t_2 \) is the atomic well-formed condition where the symbol \( == \) denotes the identity of the sort \( d \). Let \( P_1 \) and \( P_2 \) be two well-formed conditions over \( \Sigma \), then the following expressions are also well-formed conditions over \( \Sigma \):

\[
\text{not}(P_1), (P_1 \text{ and } P_2), (P_1 \text{ or } P_2).
\]

The conditional equations are equations extended by a well-formed condition \( P \) over \( \Sigma \) and written as \( L = R \text{ if } P \). All equations are supposed to be universally quantified over each variable occurring in it. We will use the OBJ-like notation [Bergstra89] for expressing specifications. E.g., the world of truth-values can be described by the following specification:

\[
\text{obj} \ \text{BOOL} \ \text{is}
\text{sorts} \ \text{Bool}
\text{opns} \ \text{true}, \text{false}: \text{Bool}
\quad \text{not}: \text{Bool} \rightarrow \text{Bool}
\text{eqns} \ \text{not(true)} = \text{false}
\quad \text{not(false)} = \text{true}
\text{endo}
\]

Let us suppose that \( R \) is a term rewriting system completed via Knuth-Bendix procedure for an algebraic specification \( S \). \( R \) consists of a set of rewriting rules. The rewriting rule is a pair of well-formed terms \( L, R \) over signature \( \Sigma \), written as \( L \Rightarrow R \). The conditional rewriting rule is a rewriting rule extended by a well-formed condition \( P \) over \( \Sigma \) and written as \( L \Rightarrow R \text{ if } P \).
The rewriting rule can be viewed as an oriented equation. E.g., the world of truth-values can be described by the following rewriting system:

```
trs  BOOL  is
    sorts  Bool
    opns  true,false: Bool
           not: Bool → Bool
    rules  not(true) ⇒ false
           not(false) ⇒ true
endtrs
```

This term rewriting system is the canonical rewriting system for BOOL-terms. It means, that any constant well-formed BOOL-term can be rewritten (with the help of BOOL) into the canonical normal form of it that serves as a meaning of it. The resulting canonical term can be either true or false, e.g.:

\[
\text{not(not(true))} \Rightarrow \text{not(false)} \Rightarrow \text{true}
\]

This process of term rewriting can be simulated in XML-like fashion. It means, that the signature of term rewriting system is converted into DTD (Document Type Definition), e.g.:

```
<?xml version="1.0" encoding="iso-8859-2"?>
<!ELEMENT trs (sorts,opns,rules)+>
<!ATTLIST trs
  name    CDATA #IMPLIED>

<!ELEMENT ident EMPTY>
<!ATTLIST ident
  name    CDATA #REQUIRED
  type    (sort|const|var) #REQUIRED
  id     IDREF #REQUIRED>

<!ELEMENT sorts (sort)+>
<!ELEMENT sort (ident)+>

<!ELEMENT opns (op)+>
<!ELEMENT op (ident,args,result)>
<!ELEMENT args (arg)+>
<!ELEMENT arg (sort)>
<!ELEMENT result (sort)>

<!ELEMENT const (ident)>
<!ELEMENT var (ident,sort)>
<!ELEMENT term (const|var|apply)>
<!ELEMENT apply (head,tail)>
<!ELEMENT head (const)>
<!ELEMENT tail (term)>`

Accordingly to this DTD, the term rewriting system BOOL can be expressed as following XML-document:
The axioms have to be converted into XSL code. Any rewriting rule which left-hand side starts with the constant not is to be converted into XSL rule triggered by the same symbol. E.g., the rule:
not(true) ⇒ false
has to be converted into XSL fragment (part of trs.xsl):

```
<!DOCTYPE trs SYSTEM "specif.dtd">
<trs name="BOOL">
  <sorts> <sort> <ident name="Bool" type="sort"/> </sort> </sorts>
  <opns>
    <op> <ident name="true" type="const"/> <args> </args>
        <result> <sort> <ident name="Bool" type="sort"/> </sort> </result>
    </op>
    <op> <ident name="false" type="const"/> <args> </args>
        <result> <sort> <ident name="Bool" type="sort"/> </sort> </result> </op>
    <op> <ident name="not" type="const"/> <args> <arg> <sort> <ident name="Bool" type="sort"/> </sort> </arg> </args>
        <result> <sort> <ident name="Bool" type="sort"/> </sort> </result>
    </op>
  </opns>
  <rules>
    <rule>
      <term> <apply> <head>
          <const> <ident name="not" type="const"/> </const> </head>
          <tail> <term>
              <const> <ident name="true" type="const"/> </const>
          </term> </tail> </apply> </term>
      <term> <const> <ident name="false" type="const"/> </const>
    </rule>
    <rule>
      <term> <apply> <head>
          <const> <ident name="not" type="const"/> </const> </head>
          <tail> <term> <const> <ident name="false" type="const"/> </const>
      </term> </tail> </apply> </term>
      <term> <const> <ident name="true" type="const"/> </const>
    </rule>
  </rules>
</trs>
```
4  An Example of Rewriting

Let us suppose the term "not(not(true))" expressed in XML:

```
<!DOCTYPE term SYSTEM "specif.dtd">
<term>
  <apply>
    <head> <const> <ident name="not" type="const"/> </const>
    </head>
    <tail> <term> <apply>
      <head> <const> <ident name="not" type="const"/> </const>
      </head>
      <tail> <term> <const> <ident name="true" type="const"/> </const> </term>
      </tail>
    </term>
  </apply>
</term>
```

With the above XSL transformation this term is rewritten to:

```
<!DOCTYPE term SYSTEM "specif.dtd">
<term>
  <const> <ident name="true" type="const"/> </const>
</term>
```

which is the canonical form of input.
5 Conclusions

The paper briefly introduces the method of constructing an executable prototype from an algebraic specification. The proposed method serves mostly as an intermediate tool. After considerable experience with our method, we will ascertain its usefulness. The entire project seeks to develop an environment for prototyping algebraic specifications. The XSL stylesheet can be viewed as an abstract rewriting machine, that can assists as a tool for the step-by-step development of specifications.

There is wide area for optimization of XSL-code. At the present time we deal with the translation of OBJ specifications into XSL code by compiling all rules for some symbol simultaneously. The anticipated development will be another code construction for selecting rules. In a typical case no reconstruction of terms is needed if matching fails. A discussion to this approach can also be found in [Bergstra 89]. A promising approach is described in [DL 90] based on a concurrent machine for rewriting and narrowing.

This work has been partially supported by the research program No. MSM 212300014 “Research in the Area of Information Technologies and Communications” of the Czech Technical University in Prague (sponsored by the Ministry of Education, Youth and Sports of the Czech Republic).

References

[W3C] http://www.w3.org