Software Tools for XML to OWL Translation

Thomas R. Kramer, Benjamin H. Marks, Craig Schlenoff, Stephen Balakirsky, Zeid Kootbally, Anthony Pietromartire

National Institute of Standards and Technology Gaithersburg, MD 20899, USA

Abstract

This paper describes a set of closely related C++ software tools for manipulating XML schemas and XML instance files and translating them into OWL class files and OWL instance files: (1) an XML schema parser (2) an XML instance file parser generator, (3) the instance file parsers generated by generator 2, (4) an XML schema to OWL class generator, (5) a domain instance XML to OWL translator generator, and (6) the domain instance XML to OWL translators generated by generator 5. These tools have been applied to information models for kitting environments and kitting plans. The main focus is on the last three tools, which differ significantly from existing resources. The tools were built at the National Institute of Standards and Technology in support of the Knowledge Driven Planning and Modeling project conducted in connection with the Working Group on Ontologies for Robotics and Automation of the IEEE Robotics and Automation Society.

Keywords: automatic, C++, information model, generator, ontology, OWL, schema, software, tool, translator, XML, XSDL

[*]T. R. Kramer is a guest researcher at the National Institute of Standards and Technology, Gaithersburg, MD 20899 USA (e-mail: kramer@nist.gov).

[**]B. H. Marks is a student at Swarthmore College, Swarthmore, PA 19081 USA (e-mail: bmarks1@swarthmore.edu).

C. Schlenoff is with the National Institute of Standards and Technology, Gaithersburg, MD 20899 USA (e-mail: craig.schlenoff@nist.gov).

S. Balakirsky is with the Georgia Tech Research Institute, Atlanta, GA 20899 USA (e-mail: stephen.balakirsky@gtri.gatech.edu).

Z. Kootbally is a guest researcher at the National Institute of Standards and Technology, Gaithersburg, MD 20899 USA (e-mail: zeid.kootbally@nist.gov).

A. Pietromartire is a guest researcher at the National Institute of Standards and Technology, Gaithersburg, MD 20899 USA (e-mail: pietromartire@nist.gov).

1. Introduction

The IEEE Robotics and Automation Society's Ontologies for Robotics and Automation (ORA) Working Group is dedicated to developing a knowledge representation for robotics and automation. As part of this working group, the Industrial Robots sub-group is tasked with studying industrial applications of the knowledge representation. One of the first areas of interest for this subgroup is the area of kit building or kitting. This is a process that brings parts that will be used in assembly operations together in a kit and then moves the kit to the area where the parts are used in the final assembly. It is anticipated that utilization of the knowledge representation will allow for the development of higher performing kitting systems. The Knowledge Driven Planning and Modeling project at the National Institute of Standards and Technology worked in collaboration with the ORA group to develop information models related to kitting, including a model of the kitting environment and a model of a kitting plan.

Early in its existence, the ORA group made a commitment to use OWL (Web Ontology Language) for its models. As the authors used OWL, difficulties arose as described in section 3. The nature of the models being built lent itself to a more structured object model approach of the sort used in languages such as EXPRESS [14], C++ classes [21], and XML Schema Definition Language (XSDL) [3], [4], [5], [25]. It was decided to use XSDL as the language for initial modeling in the KDPM project and to produce OWL models from the XSDL models. One author had already had experience with XSDL and was building C++ software tools for manipulating XML schemas and instance files. To make the translation work easier and more reliable, additional C++ tools were built for that purpose.

Much research has been devoted to translating XML into OWL. A comparison between existing utilities can be found in [6], [7], [26]. Nevertheless, existing software has many limitations. In some cases, the software converts only XML Schema [24] or requires an existing OWL ontology [19]. The majority of tools incorporate information from either XML schema files or XML instance files, but not both [8], [11], [12]. This precludes the creation of accurate OWL instances from XML instances that conform to an XML schema. Additionally, existing utilities do not scale well with input size or complexity, either requiring human verification and restructuring of the converted file [12] or limiting the potential complexity of XML schema files by only analyzing a single schema at a time [11]. Finally, most tools are implemented using mappings encoded in XML stylesheets [23], [26], [27], which seem to scale in exponential time with the length of the converted document [8]. For all of these reasons, a different, scalable approach is needed.

The remainder of this paper focuses on the tools and how the translation tools were tailored to deal with the differences between OWL and XSDL. Section II describes the tools. Section III describes key differences between XSDL and OWL. Section IV gives details about the software in the tools, and Section V presents conclusions and future work.

Reserved words from XSDL and OWL or from sample files are set `in this font`.

2. The Tools

Figure 1 shows the tools, the file types the tools manipulate, and the connections among them. The tools all run from a command window; they have no graphical user interfaces. This makes them independent of any operating system.

The files (domain.xsd and domain.owl) on the left side are information model files. They show how instances of information should be structured. For example, a point might be modeled in an information model as x, y, and z coordinate values. The files on the right side (domain-instance.xml and domain-instance.owl) are instance files that contain specific data instances that conform to an information model. For example, a specific point in an instance file might be (1, 2, 3), corresponding to the x, y, z model. Many instance files may correspond to a given information model.

The subject matter area of an information model is called its domain. The tools on the left and in the middle of Figure 1 are domain independent. Each tool will work with any XML `schema` that meets that tool's restrictions on the usage of the XSDL. The restrictions vary among the tools. The tools on the right side of the figure are domain dependent. They take as input only XML instance files in the domain for which the tools were generated.

2.1. XML Schema Parser

As indicated by arrow A on Figure 1, the XML schema parser (henceforth xmlSchemaParser) reads and writes XML `schema` files. It is able to handle almost all of XSDL. When it runs, it reads an input file, stores it in terms of

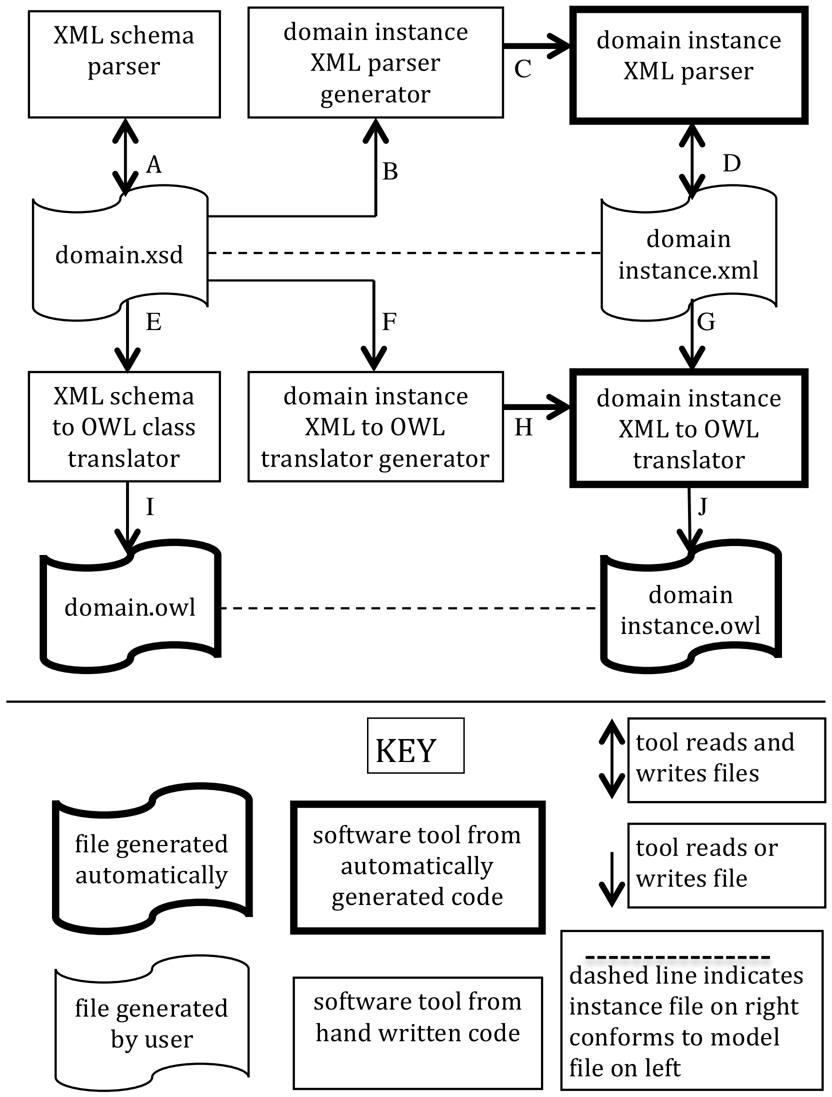
XML schema parser
domain instance XML parser generator
domain instance XML parser
A
B
C
D
domain.xsd
domain instance.xml
E
F
G
XML schema to OWL class translator
domain instance XML to OWL translator generator
domain instance XML to OWL translator
H
I
J
domain.owl
domain instance.owl
KEY
tool reads and writes files
tool reads or writes file
file generated automatically
software tool from automatically generated code
file generated by user
software tool from hand written code
dashed line indicates instance file on right conforms to model file on left

Figure 1: Software Tools, Model Files, and Instance Files

a C++ class model of XSDL `schema`s, and reprints it in a file with almost the same name as the input file; "echo" is appended to the file name. The output file is formatted to be easily human readable for the few humans who read XSDL files directly. While it runs, the xmlSchemaParser prints what it is reading in the command window in which it is running. If there is any syntax error, the xmlSchemaParser stops reading at the point where the first error occurred, prints an error message, and exits; no output file is generated.

In comparison with commercially available tools, and good free tools, this xmlSchemaParser has few advantages for general use. However, since it uses a YACC-Lex parser, it is very fast. It runs in $\mathcal{O}(N)$ time where N is the number of lines in the `schema` file. Also, it has one set of specialized options that were developed for another project. That is, the user has a choice of how `documentation` nodes are handled when the output file is generated. XSDL `documentation` nodes may be (1) deleted entirely, (2) formatted automatically for human readability, or (3) printed in a single string (for input to some other automatic formatting tool). In the second option, `documentation` nodes that have been specially formatted (as evidenced by extra indenting on one or more lines) are not reformatted. There is another option for retaining comments or removing them. That option has a simple implementation but requires that comments be located in the `schema` only where annotation nodes are allowed.

2.2. Domain Instance XML Parser Generator

The Domain Instance XML Parser Generator (henceforth xmlInstanceParserGenerator) reads an XML `schema` that models some domain and writes software for a parser that reads and writes XML instance files conforming to the `schema`. This is indicated by arrows B and C on Figure 1. The xmlInstanceParserGenerator is by far the most complex of the tools described in this paper. For a `schema` up to several thousand lines long, however, it runs in a fraction of a second on an ordinary desktop or laptop computer. If the number of `complexType`s in a `schema` file is N, the time taken by the xmlInstanceParserGenerator is $\mathcal{O}(N^2)$.

The files that are generated from the domain.xsd XML `schema` file (where domain may be any name allowed by XSDL and C++) are:

domain.lex – a Lex file for a lexical scanner used by the YACC parser.

domain.y — a YACC file for a parser for XML files in the domain.

domainClasses.hh — a C++ header file defining classes for the domain. Each class has two constructors, a destructor, and a printSelf function.

domainClasses.cc – a C++ code file implementing the classes.

domainParser.cc – a C++ code file with a main program.

If the XML `schema` file on which the xmlInstanceParserGenerator is operating `include`s one or more other XML `schema` files, a pair of domainClasses C++ files is generated for each additional `schema` file, but there is still only one Lex file, one YACC file, and one main program file.

After the xmlInstanceParserGenerator has finished running, further processing builds a Domain Instance XML Parser. The flex Lex processor [15], [18] is used to generate the C++ file domainLex.cc automatically from domain.lex. The Bison YACC processor [10], [15] is used to generate domainYACC.cc and domainYACC.hh automatically from domain.y. The four (or more) .cc files are then compiled and linked in the usual way, i.e. by using a Makefile in any operating system that uses standard Makefiles or by using Visual Studio for MS Windows [9]. As described in Section IV, an additional object file is also linked in.

In comparison with commercially available code generation tools, and good free tools, the xmlInstanceParserGenerator has few advantages for general use. The principal advantage to the authors is that we understand the code and can add any functionality we need. In addition, the many months invested in writing the code paid off in minimizing the time it took to build the XML Schema to OWL Class Translator, and the Domain Instance XML to OWL Translator Generator, each of which required only a week or so. Another, possibly minor, advantage is that all output code files are carefully formatted to be human readable if the reader is familiar with the language in which the file is written.

Another useful feature of the xmlInstanceParserGenerator is the ability to preserve changes made manually to the automatically generated domainClasses.hh header file if the input `schema` is modified and the header file is regenerated. If the arguments to the command that starts the xmlInstanceParserGenerator include *-h domainClasses.hh*, where *domainClasses.hh* is the old manually changed header file, any allowed changes in the old header file will be transcribed into the corresponding positions in the new header file that is generated. Two types of changes to header files are allowed. First, immediately after the list of #includes near the top of the file, a // style comment line may be inserted followed by more #includes. Second, immediately before the right curly brace that closes each class declaration, a // style comment line may be inserted followed by any lines that are syntactically correct in that position (for example, an attribute declaration or a constructor declaration). To accomplish the transcription of the latter type of changes, when the xmlInstanceParserGenerator starts, it reads the old header file and builds a map from class names to lists of character arrays containing the changes. When the new header is being printed, just before the printing of each class ends, the map is checked and the contents of the list of changes for that class are copied into the new header file. At the same time, "done" is put at front of the list to indicate that the changes for that class have been transcribed. After the new header file has been generated, the change map is checked to be sure all changes are marked done. If a change is not marked done, that implies that a class defined in the old header file is not present in the new one, and a warning message is printed.

Any manually written code implementing manual changes in the header file, such as a new constructor, should be put into a separate .cc file, not into domainClasses.cc. There is no problem with having multiple .cc files to implement a single .hh file, but it is not possible to use a second header file to modify classes defined in a first header file. Hence, making changes to the original header file is necessary to change classes, and some method of preserving the changes is desirable. Changing an underlying information model and adding attributes and functions to classes to support building an application are both frequently done, so being able to preserve manual changes to header files is valuable.

The subset of XSDL that can be handled by the xmlInstanceParserGenerator is more limited than that for the xmlSchemaParser. In particular, it handles only `schema`s in which all type definitions are at the `schema` level, and it cannot deal with multiple namespaces. The xmlInstanceParserGenerator does not generate code to verify that an instance file satisfies key and keyref constraints in the `schema`.

2.3. Domain Instance XML Parsers

A Domain Instance XML Parser reads and writes XML instance files intended to conform to the domain.xsd information model. This is indicated by arrow D on Figure 1.

The main program in domainParser.cc provides a text-based user interface, calls the YACC parser and calls the routine that reprints the input XML instance file in the output XML instance file. The user gives the path to the input XML instance file in the command that starts the domainParser tool. As with the xmlSchemaParser, the name of the output file is almost the same as name of the input file; again, "echo" is appended to the file name. The Domain Instance XML Parsers require strict conformance of instance files to the syntax implied by the domain.xsd `schema`. Also like the xmlSchemaParser, while it runs, a Domain Instance XML Parser prints what it is reading in the command window in which it is running. If there is any syntax error, the parser stops reading at the point where the first error occurred, prints an error message, and exits; no output file is generated.

While a Domain Instance XML Parser does not check conformance of instance files to any key and keyref constraints that may be present in domain.xsd, it does check that all values of the XML built-in `ID` type in an instance file are unique and that every `IDREF` value is the value of an `ID`.

If the number of lines in an XML instance file is N, the time taken by a Domain Instance XML Parser is $\mathcal{O}(N)$.

2.4. XML Schema to OWL Class Translator

The XML Schema to OWL Class Translator (xmlSchemaOwlClassGenerator) reads XML `schema` files and writes OWL files declaring OWL `class`es. This is indicated by arrows E and I in Figure 1. The xmlSchemaOwlClassGenerator outputs one OWL `class` file for each input XML `schema` file. Each `class` file defines a syntactically complete OWL `ontology`. An XML `schema` file may be input either by being named in an argument to the xmlSchemaOwlClassGenerator or by being `include`d in the named file or in another `include`d file. Each OWL `class` file that is output contains an information model with the same meaning as the corresponding model defined by an XML `schema` file. The correspondence between the content of an XML `schema` file and that of the corresponding OWL `class` file is described in section 3. That section also describes the several restrictions on the subset of XSDL that may be used in a `schema` from which an OWL `class` file is to be generated.

If the number of `complexType`s in a `schema` file is N, the time taken by the xmlSchemaOwlClassGenerator is $\mathcal{O}(N^2)$.

2.5. Domain Instance XML to OWL Translator Generator

The Domain Instance XML to OWL Translator Generator (xml2owlGenerator) reads an XML `schema` and writes code for a Domain Instance XML to OWL

Translator. This is indicated by arrows F and H on Figure 1. The user provides a base name for the files to be generated on the command line that starts the xml2owlGenerator. If the base name is "domain", the code files the generator writes are:

> owlDomainClasses.hh – a C++ header file defining classes for the domain. Each class has two constructors, a destructor, and a printOwl function.
>
> owlDomainClasses.cc – a C++ code file implementing the classes.
>
> owlDomainPrinter.cc – a C++ program with a main routine.

The constructors and destructors that are generated are identical to those produced by the xmlInstanceParserGenerator.

The xml2owlGenerator does not generate Lex and YACC files. The ones generated by the xmlInstanceParserGenerator are used instead. However, when domainYACC.cc is compiled, owlDomainClasses.hh is included rather than domainClasses.hh. The four .cc files are compiled and linked in the usual manner. As described in Section IV, two additional object files are also linked.

If the number of `complexType`s in a `schema` file is N, the time taken by the xml2owlGenerator is $\mathcal{O}(N^2)$.

2.6. Domain Instance XML to OWL Translators

A Domain Instance XML to OWL Translator reads an XML instance file conforming to domain.xsd and writes an OWL instance file conforming to domainClasses.owl. This is indicated by arrows G and J on Figure 1. The two files have the same information content. The correspondence between the content of the XML instance file and that of the OWL `class` file is described in Section III.

The Domain Instance XML to OWL Translators are very fast. If the number of lines in an XML instance file is N, the time taken by a Domain Instance XML to OWL Translator is $\mathcal{O}(N)$. In an unexceptional test, a test file with 129,000 lines was translated in 0.45 seconds.

2.7. Limitations

The four handwritten tools shown in Figure 1 have different levels of capability in handling XML `schema` files. The xmlSchemaParser can handle almost any XML `schema` file. The xmlInstanceParserGenerator can handle only `schema`s in which all type definitions are at the top level and has other limitations that are not described in this paper. The translation tools (xmlSchemaOwlClassGenerator and xml2owlGenerator) have all the limitations of the xmlInstanceParserGenerator plus others that are described below.

3. XSDL and OWL

This section briefly describes XSDL models in subsection 3.1, XML instance files in subsection 3.2, OWL models in subsection 3.3, and OWL instance files in subsection 3.4. The descriptions of languages and file formats are sufficient only to support the explanation of translations. Full descriptions may be found for XSDL in [3], [4], [5], and [25], for XML in [1], and for OWL in [2], [13], and [17]. XSDL and OWL versions of the same small complete model are shown in subsections 3.1 and 3.3. XML and OWL versions of the same small instance file conforming to the model are shown in subsections 3.2 and 3.4.

Finally, subsection 3.5 provides additional discussion of problems with using OWL that are circumvented by using the translation tools.

3.1. XSDL Schemas

XSDL is an object-oriented information modeling language. A model written in XSDL is called a `schema`. Data members may be represented in the model as `elements`. The contents of a `schema` normally include a root `element` and a number of type definitions. Objects are modeled as instances of `complexType`s that may have `elements`. XSDL also includes built-in data types such as `ID`, `integer`, and `string` and supports specializations of built-in data types in `simpleType`s. The following line.xsd `schema` file illustrates how a two dimensional `Line` might be modeled in XSDL using `PointType` and `VectorType`.

```
<?xml version="1.0" encoding="UTF-8"?>

<xs:schema
```

```
xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
attributeFormDefault="unqualified">

<xs:element name="Line"
  type="LineType">
  <xs:annotation>
    <xs:documentation>
      Root element
    </xs:documentation>
    <xs:documentation>owlPrefix=ax</xs:documentation>
  </xs:annotation>
</xs:element>

<xs:complexType name="BaseType"
  abstract="true">
  <xs:sequence>
    <xs:element name="Name"
      type="xs:ID"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="LineType">
  <xs:complexContent>
    <xs:extension base="BaseType">
      <xs:sequence>
        <xs:element name="Point"
          type="PointType"/>
        <xs:element name="Vector"
          type="VectorType"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<xs:complexType name="PointType">
  <xs:complexContent>
    <xs:extension base="BaseType">
```

```
        <xs:sequence>
          <xs:element name="X"
            type="xs:decimal"/>
          <xs:element name="Y"
            type="xs:decimal"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <xs:complexType name="VectorType">
    <xs:complexContent>
      <xs:extension base="BaseType">
        <xs:sequence>
          <xs:element name="X"
            type="xs:decimal"/>
          <xs:element name="Y"
            type="xs:decimal"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

</xs:schema>
```

The four type `names` in the file above are `BaseType`, `LineType`, `PointType`, and `VectorType`.

In general, the translation tools require that input `schema`s have a completely uniform style of using XSDL. For example, XSDL does not require that type definitions in a `schema` have `name`s and be at the top level of the `schema`, but in XML to OWL translation, we allow only `schema`s that meet those conditions. Requiring a uniform style does not limit what may be modeled in any way.

In order that `element name`s may be very similar to type `name`s, we have adopted the conventions that all type `name`s (and only type `name`s) must end in `Type`, and that wherever it is reasonable to do so, the `name` of an `element` will be the `name` of its type with the `Type` suffix removed.

Another requirement on `complexTypes` that we have imposed in order to support translation to OWL is that every `complexType` must have a `Name element` of `ID` type. The `ID` type is used to ensure that every `Name` for a named object in an instance file is unique throughout the file.

One `complexType` (child) may be derived from another (parent) by extending or restricting the parent. Restrictions of `complexType`s are awkward and verbose in XSDL and are not allowed in `schema`s used with the translation tools. Extensions usually add `element`s. The child has all the `element`s of its parent plus any that are added by the extension. XSDL does not provide any method for a child type to have two parent types. In modeling terms, that means multiple inheritance is not possible. In the `schema` file above, the `BaseType`, which provides the `Name element`, is the parent of the other three types.

The scope of `element` names in XSDL is local to the type in which the `element` appears. In the example above, for instance, both `Point` and `Vector` have `X` and `Y elements`.

Several restrictions on the use of XSDL in `schema`s that are to be used as input to the translation tools have already been mentioned. Others follow.

Attributes not allowed XSDL `attribute`s are not allowed. It is always straightforward to replace an XSDL `attribute` with an `element` having exactly the same semantic content. Thus, disallowing `attribute`s limits input syntax but not input semantics.

Namespace not allowed XSDL and OWL both provide for using prefixes to implement separate namespaces. However, they do this at different levels of granularity. XSDL allows multiple `schema` files in a single namespace (or no namespace) while OWL puts each `ontology` file in its own namespace. No `schema` file that is to be processed by the translation tools may have a namespace or use a prefix.

OWL prefix specification required In OWL, each namespace (i.e., file) must have a different prefix. One of these may be the empty prefix which is a bare colon (:). In the translation tools, the empty prefix is reserved for OWL instance files. The xmlSchemaOwlClassGenerator outputs one OWL `ontology` file for each input XML `schema` file. Some method of assigning a unique non-empty prefix to each output OWL file is required. The method that has been implemented is to require that there be a `documentation` node containing the prefix in each XML `schema` file. The text of the `documentation` node is of the form `owlPrefix=ax`, where the `ax` may be any combination of characters allowed for OWL prefixes. That `documentation` node should be placed in the root `element` of the XML `schema` if there is a root `element`, or anywhere else `documentation` nodes are allowed, if not. All such prefixes must be different. A colon will be added to the end of the prefix when it is used. In line.xsd, `owlPrefix=ax` is on the twelfth line.

Handling of Key Limited The handling of XSDL key is limited. This is because XSDL `key`s are `element`-based and apply only to specified instances of a type, while OWL `hasKey` statements are type-based and apply to all instances of a type.

Global Element Only for Root An `element` may be declared at the top (global) level of a `schema` only if it is the root `element`. In this case it should appear before any type definitions.

Specialized Use of ID and IDREF An XML instance file is a hierarchy that is structurally a tree. It is often the case in model building that we want the value of an `element` in one part of a tree to be in some part of the tree other than being directly below the `element`. In a model of a family tree, for example, the value of a cousin `element` will normally be that way. To deal with `element`s of this sort in XSDL, the usual method is to assign an identifier unique among all objects to each object that might be the value of some distant `element`. Then the value of the `element` would be the identifier. Any system processing the tree would be aware that when an identifier is the value of an `element`, the intent is really that the value of the `element` is the object identified by the identifier. The `Name element` of `ID` type discussed earlier, which is possessed by every instance of every `complexType`, serves as an identifier. References in an XML `schema` to `Name` identifiers must be of type `IDREF` (which is XML's built-in type for references to `ID`s). To enable translation, in the XML to OWL tools, it is also required that each `element` of type `IDREF` have a `annotation` node with an `appinfo` node inside it that gives the name of the type of thing the `IDREF` is referencing. Here is a file snippet with an example of that. The value of the `DesignName element` will be the name of an instance of `KitDesignType`, presumably to be found in a list of designs given elsewhere in the model.

```
<xs:complexType name=KitType>
  ...
  <xs:element name=DesignName
    type=xs:IDREF
    <xs:annotation>
      <xs:appinfo>KitDesignType</xs:appinfo>
    <xs:annotation>
  </xs:element>
  ...
</xs:complexType
```

Other items not handled The following XSDL constructs are not usefully handled by the translation tools: `choice`, `fixed`, `keyref`, `maxLength`, `maxOccurs` of a `sequence`, `minLength`, `minOccurs` of a `sequence`, `mixed`, `pattern`, `ref`, `list`, `substitutionGroup`, `unique`. For some of these, if the construct appears in a `schema`, the XML to OWL tool will print an error message and exit. For others, the tool will print a warning message and ignore the construct.

3.2. XML Instance Files Conforming to XSDL Schemas

Under the XML standards, an XML instance file conforming to an XSDL `schema` must be in a different format than the `schema` and must contain different sorts of statements. An XML statement naming the XSDL `schema` file to which an instance file corresponds is normally given near the beginning of the instance file. Many different instance files may correspond to the same `schema`.

The form of an instance file is a tree in which instances of the `elements` of each type are textually inside the instance of the type.

The following line1.xml XML instance file conforms to the line.xsd XML `schema`. Names of `elements` in the `schema` become XML tags in the instance file (e.g., `<Point>`). The line1.xml file models a line that passes through the origin and lies on the Y axis.

```
<?xml version="1.0" encoding="UTF-8"?>
<Line
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="../xmlSchemas/line.xsd">
  <Name>Line_1</Name>
  <Point>
    <Name>Point_1</Name>
    <X>0</X>
    <Y>0</Y>
  </Point>
  <Vector>
    <Name>Vector_1</Name>
    <X>0</X>
    <Y>1</Y>
  </Vector>
</Line>
```

In XSDL, there is a rule that a valid instance of a `complexType` must have valid instances of the required `element`s of the type in the order given in the `schema`, and `element`s are required unless explicitly made optional in the `schema`. Thus, for example, the `Line_1` instance of `LineType` shown above is valid since it has a valid `Name element` followed by a valid `Point element` followed by a valid `Vector element`. If it did not have those valid `element`s in that order, it would not be valid.

3.3. OWL Class Model

OWL is designed to support automated reasoning and is set theoretic. It is much more atomistic than XSDL. OWL has several different but equivalent syntaxes. The OWL functional-style syntax has been used in the translation tools.

Here is the OWL lineClasses.owl `ontology` file equivalent to the line.xsd `schema` file in subsection 3.1. The lineClasses.owl file was produced by the xmlSchemaOwlClassGenerator. The first section of lineClasses.owl is a header. The other four sections (starting with a `class` declaration) each correspond to one of the four `complexType` definitions in line.xsd. The first five lines of the header are boilerplate used in all OWL files. The sixth line declares that the prefix `ax` should be used with this `ontology`. That prefix is specified in a `documentation` node of the root node of line.xsd. The beginning of the `ontology` name `http://example/line/lineClasses.owl` is provided by the user as an argument to the xmlSchemaOwlClassGenerator. The generator adds ···`Classes.owl` to the end – where ··· is `line` in this case.

```
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
Prefix(xml:=<http://www.w3.org/XML/1998/namespace>)
Prefix(rdf:=<http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
Prefix(rdfs:=<http://www.w3.org/2000/01/rdf-schema#>)
Prefix(ax:=<http://example/line/lineClasses.owl#>)
Ontology(<http://example/line/lineClasses.owl>

Declaration(Class(ax:Base))

SubClassOf(ax:Line ax:Base)
SubClassOf(ax:Point ax:Base)
SubClassOf(ax:Vector ax:Base)
DisjointUnion(ax:Base
              ax:Line
              ax:Point
              ax:Vector)

Declaration(Class(ax:Line))

Declaration(ObjectProperty(ax:hasLine_Point))
ObjectPropertyDomain(ax:hasLine_Point ax:Line)
ObjectPropertyRange(ax:hasLine_Point ax:Point)
InverseFunctionalObjectProperty(ax:hasLine_Point)
FunctionalObjectProperty(ax:hasLine_Point)
EquivalentClasses(ax:Line ObjectIntersectionOf(
  ObjectSomeValuesFrom(ax:hasLine_Point ax:Point)
  ObjectAllValuesFrom (ax:hasLine_Point ax:Point)))

Declaration(ObjectProperty(ax:hadByPoint_Line))
InverseObjectProperties(ax:hasLine_Point
                        ax:hadByPoint_Line)
ObjectPropertyDomain(ax:hadByPoint_Line ax:Point)
ObjectPropertyRange(ax:hadByPoint_Line ax:Line)

Declaration(ObjectProperty(ax:hasLine_Vector))
ObjectPropertyDomain(ax:hasLine_Vector ax:Line)
ObjectPropertyRange(ax:hasLine_Vector ax:Vector)
```

```
InverseFunctionalObjectProperty(ax:hasLine_Vector)
FunctionalObjectProperty(ax:hasLine_Vector)
EquivalentClasses(ax:Line ObjectIntersectionOf(
  ObjectSomeValuesFrom(ax:hasLine_Vector ax:Vector)
  ObjectAllValuesFrom (ax:hasLine_Vector ax:Vector)))

Declaration(ObjectProperty(ax:hadByVector_Line))
InverseObjectProperties(ax:hasLine_Vector
                        ax:hadByVector_Line)
ObjectPropertyDomain(ax:hadByVector_Line ax:Vector)
ObjectPropertyRange(ax:hadByVector_Line ax:Line)

Declaration(Class(ax:Point))

Declaration(DataProperty(ax:hasPoint_X))
DataPropertyDomain(ax:hasPoint_X ax:Point)
DataPropertyRange(ax:hasPoint_X xsd:decimal)
FunctionalDataProperty(ax:hasPoint_X)
EquivalentClasses(ax:Point ObjectIntersectionOf(
  DataSomeValuesFrom(ax:hasPoint_X xsd:decimal)
  DataAllValuesFrom (ax:hasPoint_X xsd:decimal)))

Declaration(DataProperty(ax:hasPoint_Y))
DataPropertyDomain(ax:hasPoint_Y ax:Point)
DataPropertyRange(ax:hasPoint_Y xsd:decimal)
FunctionalDataProperty(ax:hasPoint_Y)
EquivalentClasses(ax:Point ObjectIntersectionOf(
  DataSomeValuesFrom(ax:hasPoint_Y xsd:decimal)
  DataAllValuesFrom (ax:hasPoint_Y xsd:decimal)))

Declaration(Class(ax:Vector))

Declaration(DataProperty(ax:hasVector_X))
DataPropertyDomain(ax:hasVector_X ax:Vector)
DataPropertyRange(ax:hasVector_X xsd:decimal)
FunctionalDataProperty(ax:hasVector_X)
EquivalentClasses(ax:Vector ObjectIntersectionOf(
  DataSomeValuesFrom(ax:hasVector_X xsd:decimal)
  DataAllValuesFrom (ax:hasVector_X xsd:decimal)))

Declaration(DataProperty(ax:hasVector_Y))
DataPropertyDomain(ax:hasVector_Y ax:Vector)
DataPropertyRange(ax:hasVector_Y xsd:decimal)
FunctionalDataProperty(ax:hasVector_Y)
EquivalentClasses(ax:Vector ObjectIntersectionOf(
  DataSomeValuesFrom(ax:hasVector_Y xsd:decimal)
  DataAllValuesFrom (ax:hasVector_Y xsd:decimal)))
)
```

For each XSDL type defined in the XML `schema` an equivalent OWL type is declared in the OWL `ontology` that is generated by the Class Translator. Also, for each `element` (other than `Name`) of each XSDL `complexType`, an OWL property is declared. If an XSDL type is a `simpleType`, the OWL equivalent is a `DatatypeDefinition`, and when it is used as the type of an `element`, the equivalent OWL property is a `DataProperty`. If the XSDL `element` type is a `complexType`, the OWL equivalent is a `class`, and when it is used as the type of an `element`, the equivalent OWL property is an `ObjectProperty`. The suffix `Type` is removed from the XSDL type name in order to make the OWL `class` name or `DatatypeDefinition` name. XSDL has built-in data types, such as `xs:decimal`. OWL uses many of the XSDL built-in data types directly. For these, translation is straightforward. For example, `xs:decimal` becomes `xsd:decimal`. The translation of line.xsd to lineClasses.owl provides examples of conversions of `complexType` and built-in type, but not `simpleType`.

The `Name element` required in every XSDL `complexType` has no counterpart in the OWL `class` equivalent to the `complexType`. In an OWL instance file, objects are usually named by a `NamedIndividual` declaration, so they do not have to be modeled in OWL `classes`. If there were a counterpart to the XSDL `Name` in the equivalent OWL `class`, each object of the `class` would have two names: the explicitly modeled one and the one assigned by the `NamedIndividual` declaration. The purpose of requiring the XSDL `Name` is so that XML instances of `complexTypes` will have names that can be used as the OWL instance names.

In the OWL `class` file above, almost all statements about a given property or `class` are clustered together. This is not a requirement of OWL; it's a feature of the xmlSchemaOwlClassGenerator. After the header, the order of statements in an OWL `ontology` file is irrelevant.

As seen in lineClasses.owl, the `DataProperty`s and `ObjectProperty`s are all declared globally in the `ontology`, not locally in a `class`. Hence, a method is required of making property names (such as `x` and `y`) that were local in XSDL be global in OWL. This has been done by constructing the property name by concatenating `has` with the XSDL type name (which is global), an underscore, and the XSDL `element` name. Thus, for example, we have the property names `hasPoint_X`, `hasPoint_Y`, `hasVector_X`, and `hasVector_Y`. Since the XSDL type names are unique within a `schema` file, and the `element` names are unique within a type, the OWL property names are unique within the `ontology` file.

In OWL, the domains and ranges of properties are specified using explicit `DataPropertyDomain`, `DataPropertyRange`, `ObjectPropertyDomain`, and `ObjectPropertyRange` statements.

If an XSDL `element` can occur at most once in a `complexType`, then a `FunctionalDataProperty` or `FunctionalObjectProperty` statement for the OWL property equivalent to the `element` is made.

If an XSDL `complexType` has one or more `element`s that are not optional, for each such `element`, an OWL `EquivalentClasses` statement is made saying that all members of the OWL `class` equivalent to the XSDL `complexType` and only members of that `class` have the OWL property equivalent to the `element`.

If an XSDL `element` of `complexType` can occur at most once, an OWL `InverseFunctionalObjectProperty` statement is made.

For each `objectProperty`, an inverse property is declared along with its domain and range. In the sample OWL instance file, `hadByPoint_Line` is the inverse of `hasLine_Point`. An explicit `InverseObjectProperties` statement is made to formalize the relationship of the two properties. Similarly, `hadByVector_Line` is the inverse of `hasLine_Vector`. In XSDL, under the line.xsd `schema`, an instance of a `VectorType` cannot be a `PointType` or a `LineType`. In OWL, absent a statement to the contrary, a `Vector` could be a `Point` or a `Line`. To prevent that from being possible, the last statement in the `Base class` section of the OWL `class` file says that the the `Line`, `Point`, and `Vector classes` form a `DisjointUnion` of the `Base class`. That means both (1) that no instance of `Line`, `Point`, or `Vector` can also be an instance of one of the others and (2) that any instance of the `Base class` must be an instance of one of its subtypes. The use in OWL of a `disjointUnion` (which implies both 1 and 2) rather than a `disjointClasses` (which would imply only 1) occurs because the `BaseType` was declared to be `abstract` in line.xsd.

The line.xsd and lineClasses.owl files do not use all XSDL and OWL constructs. The xmlSchemaOwlClassGenerator generates additional types of OWL statement corresponding to other XSDL constructs. The remainder of this section deals with those.

- An XSDL `include` statement is translated into an OWL `import` statement.
- As previously mentioned, XSDL `simpleType`s are translated to OWL `DatatypeDefinition`s.
- XSDL comments are not translated.
- An XSDL `documentation` node in a type definition or immediately after the file header is translated into an OWL `AnnotationAssertion`. The text of the `documentation` is modified to reflect the facts (1) that type names do not end in `Type` in OWL, (2) that the `Name element` is not used in OWL, and (3) that the term `element` is not used in OWL.
- An XSDL `documentation` node in an `element` definition is not translated.
- An `appinfo` in an `element` of type `IDREF` is translated by making the type of the range of the OWL `objectProperty` equivalent to the `element` be the type identified by the `appinfo`. For example, the OWL range statement for the OWL property corresponding to the `DesignName element` in the XSDL snippet shown earlier would be: `ObjectPropertyRange(kt:hasKit_DesignName kt:KitDesign)`.

3.4. OWL Instance Files

OWL has no built-in distinction between an instance file and a model file. Instance definitions and `class` definitions can be mixed in the same file. A `ClassAssertion` about an instance may even implicitly declare a new `class` (if the name of an existing `class` is misspelled, for example). The authors, however, have adopted the convention that statements about instances must be put into separate files from statements about `class`es that do not deal with instances. OWL files with statements about instances are being called instance files. We have also adopted the convention that an OWL instance file must have an OWL `Import` statement that names the `class` file to which the instance file corresponds. As in XML, many different instance files may correspond to the same model (i.e. `class`) file. The translation tools write instance translators that read one XML instance file and write one OWL instance file each time the translator is used.

Here is line1.owl, the OWL equivalent of the line1.xml. The line1.owl file was produced by owlLinePrinter, an instance translator produced automatically from line.xsd by the xml2owlGenerator. The line1.xml file was used as input to the owlLinePrinter.

```
Prefix(xsd:=<http://www.w3.org/2001/XMLSchema#>)
Prefix(owl:=<http://www.w3.org/2002/07/owl#>)
Prefix(xml:=<http://www.w3.org/XML/1998/namespace>)
Prefix(rdf:=<http://www.w3.org/1999/02/22-rdf-syntax-ns#>)
Prefix(rdfs:=<http://www.w3.org/2000/01/rdf-schema#>)
Prefix(:=<http://example/line/line1.owl#>)
Prefix(ax:=<http://example/line/lineClasses.owl#>)
Ontology(<http://example/line/line1.owl>
Import(<file:lineClasses.owl>)

//****************************************
// 1 start ax:Line Line_1
Declaration(NamedIndividual(:Line_1))
ClassAssertion(ax:Line :Line_1)
ObjectPropertyAssertion(ax:hasLine_Point
                        :Line_1 :Point_1)

//****************************************
// 2 start ax:Point Point_1
Declaration(NamedIndividual(:Point_1))
ClassAssertion(ax:Point :Point_1)
DataPropertyAssertion(ax:hasPoint_X :Point_1
                      "0.000000"^^xsd:decimal)
DataPropertyAssertion(ax:hasPoint_Y :Point_1
                      "0.000000"^^xsd:decimal)
// 2 end ax:Point
//****************************************
ObjectPropertyAssertion(ax:hasLine_Vector
                        :Line_1 :Vector_1)

//****************************************
// 2 start ax:Vector Vector_1
Declaration(NamedIndividual(:Vector_1))
ClassAssertion(ax:Vector :Vector_1)
DataPropertyAssertion(ax:hasVector_X :Vector_1
                      "0.000000"^^xsd:decimal)
DataPropertyAssertion(ax:hasVector_Y :Vector_1
                      "1.000000"^^xsd:decimal)
// 2 end ax:Vector
//****************************************
// 1 end ax:Line
//****************************************
)
```

The first five lines of line1.owl are the same boilerplate used for lineClasses.owl.

As indicated by the eighth line of the file, the instances constitute an `ontology`.

As seen in the file, `Line_1` is not defined using a hierarchy. Rather, the definition is given by a set of `Declarations` of `NamedIndividuals`, `ClassAssertions`, `ObjectPropertyAssertions`, and `DataPropertyAssertions`, all of which occur at the top level of the file. The owlLinePrinter, however, has used comments to divide subsets of the statements hierarchically into groups and used integers to indicate the hierarchical level of each group. This hierarchy matches the hierarchy of the `Line` root `element` of line1.xml.

Notice that the `ax` prefix is used in front of all the `class` and property names occurring in the OWL `class` file, but no prefix is used for the items introduced in the instance file. OWL provides that if one `ontology imports` one or more other `ontologys` (as on the ninth line of line1.owl), a prefix must be assigned to all but one of the `ontologys` (as on the sixth and seventh lines). The sixth line explicitly assigns the empty prefix to the items introduced in the instance file. We decided that instance files should use the empty prefix. Hence, every `class` file must have a prefix that is not the empty prefix.

The values of the `Name elements` in the XML instance files are used as the names of the objects in the OWL instance file.

If more than one `NamedIndividual` is created of any given instantiable `class`, under OWL's open world assumption, unless a statement is made to the contrary, the individuals may be the same individual with two different names. Since, as in XML instance files (where it is implicit), the intent is that all `NamedIndividual`s be distinct, if there are two or more individuals in an instantiable `class`, at the end of the OWL instance file a `DifferentIndividuals` statement that lists all the individuals is made for the `class`. The sample file above has only one individual in each instantiable `class`, so it contains no such statement. Since all the instantiable `class`es or their ancestors are explicitly made disjoint in either a `disjointClasses` statement or a `disjointUnion` statement, `NamedIndividual`s of different `class`es cannot be the same individual.

3.5. OWL problems obviated by using the translation tools

A number of features of OWL [2], [17] and Protégé [13], the main tool available for building OWL `ontology`s, make it impractical to build OWL models and instance files directly. The primary reason for this is that user errors in spelling the names of `NamedIndividual`s, properties, and `class`es are not recognized as errors.

The feature of OWL that creates the problem is OWL's open world assumption. Simply put, the assumption is that anything might be true that is not explicitly ruled out (1) by OWL statements directly, or (2) by reasoning from statements that have been made. The Open World assumption is appropriate in some contexts, however the kitting domain may be readily handled under a closed world assumption. Using an open world assumption introduces difficulty without providing any advantages.

Also, if the name of a `class`, property, or individual is used without being explicitly declared as such in the file (as happens when a name is misspelled), that `class`, property, or individual is implicitly declared. It would be helpful if Protégé had a switch that would cause Protégé to give a warning if a name were used that was not explicitly declared elsewhere in the file, but there is no such switch. Protégé does provide some help with spelling by having an auto complete window to use when expressions are being constructed. A misspelled term will appear as one of the choices while the user types, if the first few letters are correct.

Another problem is that, while constructing an OWL file, it is easy to omit OWL statements one intends to make. Omitting any one statement or any set of statements after the header in either lineClasses.owl or line1.owl will not be an OWL error and will not cause Protégé to flag any error or give any warning. The same would be true of many other OWL files.

Finally, Protégé does not check completely whether an OWL file conforms to the OWL spec. For example, if an OWL instance file `import`s an OWL `class` file and the prefix declared for both files is the empty prefix, no error will be signaled, even though the OWL spec says explicitly that this is not allowed.

Some of these issues can be detected, and research aimed at developing better OWL consistency checkers is ongoing [16], [22]. One utility, Pellet, offers some support for advanced reasoning and debugging [20]. In our tests, the Pellet command line linter was able to detect spelling errors within OWL, but Pellet was unable to detect a missing statement. Further, Pellet seems to support OWL XML syntax, but was unable to parse functional style syntax. Limitations still remain.

The use of an undefined type in an XML `schema` file is an error, and readily available XML tools will detect and flag it. Similarly, a missing `element` in an instance file will be detected and flagged. If an `IDREF` is made to an `ID` that has not been used, that will be detected and flagged. If a portion of an XML `schema` file is omitted, in many cases, that will be detected when the file is read, and in most cases an error will be signaled if an instance file is read that conforms to the complete intended `schema` file. Thus, almost all spelling errors that will pass in OWL will fail in XML, and most errors of omission that will pass in OWL will fail in XML.

The translation tools do not make spelling errors or errors of omission. Hence, by using them on tested XML schemas and instance files, correct OWL files may be produced. In addition, it is easier to work with XML files since (1) they are structured while OWL files are not, and (2) XML files are about half as long as the equivalent OWL files.

4. Software Details

As mentioned previously, the source code for the four hand-written XML to OWL tools (all of which take an XML `schema` file as input) is primarily in C++. All of them use xmlSchemaClasses.cc (classes for representing XSDL structures), xmlSchema.y (the YACC parser for `schema` files), and xmlSchema.lex (the lexer used by the YACC parser). In order to deal with XSDL pattern constraints, the xmlSchemaParser and the xmlInstanceParser-Generator also use a second YACC-Lex parser built from pattern.y and pattern.lex. Each of the four tools has a C++ file dedicated to its particular job in addition to the other files. The largest of those is xmlInstanceParserGenerator.cc at over 11,000 lines.

The source code for three of the four tools defines a generator class containing all the functions needed for the tool as well as a set of variables for data about the XML `schema` being processed. The xmlSchemaParser does not need a generator class since it does not process `include`d files and is not generating anything new. The YACC parser in the xmlSchemaParser builds a model of the input schema. The rest of the xmlSchemaParser just needs to print out the model. In the xmlSchemaOwlClassGenerator and the xmlInstanceParserGenerator, if `include` commands are used in the `schema`, so that more than one `schema` file is to be processed, a separate instance of the generator class is created for each `include`d file. In those tools there is one or more additional output files for each additional input file. The xml2owlGenerator outputs the same number of files regardless of the number of included schema files, so it requires only one instance of its generator class.

The source code for the automatically generated domain instance XML parsers and domain instance XML to OWL translators was partially described in subsections 2.2 and 2.5 of Section 2. To help with writing XML instance data, these tools also link in an object file compiled from the hand-written domain-independent xmlSchemaInstance.cc file. The OWL instance file writer needs help from that file because primitive OWL data is XML data. The domain instance XML to OWL translators also link in the object file compiled from the hand-written domain-independent owlInstancePrinter.cc file, which contains a set of functions that know how to print specific types of OWL constructs.

5. Conclusions and Future Work

This paper has described a suite of domain-independent software tools that enable the completely automatic generation of OWL model files and instance files from XSDL model files and XML instance files. We are using these tools in connection with our work in robotic kitting. The tools should be useful in other projects using OWL if the domain of the project is controllable and XSDL is adequately expressive to build a model. Of course, if XSDL models and XML instance files are usable, there may be no reason to use OWL.

The software tools presented differ from existing utilities. By incorporating both the XML `schema` and instance files, we are able to produce OWL instances conforming strictly to the corresponding XML `schema`. Input `schema` files can be complex and may `include` other `schema` files. The produced OWL instances do not require human refactoring or manipulation. Finally, the Domain Instance XML to OWL Translators, which are the only tools needing to be run more than once for a given schema, scale in linear time with the number of lines in an instance file.

We suspect that large projects using OWL files created directly will require automated methods of checking the integrity of the files beyond what is provided by Protégé. Some integrity checking work has been done, however current software still is limited in error checking abilities.

Future work on the XML to OWL tools might be directed towards (1) expanding the range of XSDL syntax that the three generators can handle, and (2) making the generators run in $\mathcal{O}(N \log N)$ time rather than $\mathcal{O}(N^2)$ time. The first target for expanding the range of syntax is handling attributes as well as elements. The speed might be improved as indicated by using more efficient search mechanisms with lists of pointers to classes.

References

[1] Extensible Markup Language (XML) 1.0 (Fifth Edition) W3C Recommendation (26 November 2008).

[2] OWL 2 Web Ontology Language Primer W3C Recommendation (27 October 2009).

[3] XML Schema Part 0: Primer Second Edition W3C Recommendation (28 October 2004).

[4] XML Schema Part 1: Structures Second Edition W3C Recommendation (28 October 2004).

[5] XML Schema Part 2: Datatypes Second Edition W3C Recommendation (28 October 2004).

[6] Khalid M Albarrak and Edgar H Sibley. A Survey of Methods that Transform Data Models into Ontology Models. In *Information Reuse and Integration (IRI), 2011 IEEE International Conference on*, pages 58–65. IEEE, 2011.

[7] Ivan Bedini, Georges Gardarin, and Benjamin Nguyen. Deriving Ontologies from XML Schema. *arXiv preprint arXiv:1001.4901*, 2010.

[8] Hannes Bohring and Sören Auer. Mapping XML to OWL Ontologies. *Leipziger Informatik-Tage*, 72:147–156, 2005.

[9] Microsoft Corporation. *Microsoft Visual C++ 2010 Express*. 2010.

[10] Charles Donnelly and Richard Stallman. Bison, The YACC-compatible Parser Generator. `http://dinosaur.compilertools.net/bison/`, 2006.

[11] Roberto García. A Semantic Web Approach to Digital Rights Management. *Doctorate in Computer Science and Digital Communication. Department of Technologies. Universitat Pompeu Fabra, Barcelona*, 2006.

[12] Raji Ghawi. *Ontology-based Cooperation of Information Systems*. PhD thesis, University of Borgogne, March 2010.

[13] Matthew Horridge. A Practical Guide To Building OWL Ontologies Using Protégé 4 and CO-ODE Tools. *University of Manchester*, 2011.

[14] ISO. 10303-11: 2004: Industrial automation systems and integration — Product data representation and exchange — Part 11 : Description method: The EXPRESS language reference manual. 2003.

[15] John R Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly, 1992.

[16] Boris Motik, Ian Horrocks, and Ulrike Sattler. Adding Integrity Constraints to OWL. In *OWLED*, volume 258, 2007.

[17] Boris Motik, Peter F Patel-Schneider, and Bijan Parsia. OWL 2 Web Ontology Language: Structural Specification and Functional-Style Syntax W3C Recommendation (27 October 2009).

[18] V. Paxson, W.L. Estes, and J. Millaway. Flex, Version 2.5.31 — A Fast Scanner Generator. `http://www.gnu.org/software/flex/`, 2003.

[19] Toni Rodrigues, Pedro Rosa, and Jorge Cardoso. Mapping XML to Existing OWL Ontologies. In *International Conference WWW/Internet*, pages 72–77. Citeseer, 2006.

[20] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A Practical OWL-DL Reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.

[21] Bjarne Stroustrup. *C++ Programming Language.* Addison-Wesley, special edition, 2000.

[22] Jiao Tao, Evren Sirin, Jie Bao, and Deborah L McGuinness. Integrity Constraints in OWL. In *AAAI*, 2010.

[23] Pham Thi Thu Thuy, Young-Koo Lee, and SungYoung Lee. Dtd2owl: Automatic Transforming XML Documents into OWL Ontology. In *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, pages 125–131. ACM, 2009.

[24] Chrisa Tsinaraki and Stavros Christodoulakis. Xs2owl: A Formal Model and a System for enabling XML Schema Applications to interoperate with OWL-DL Domain Knowledge and Semantic Web Tools. In *Digital Libraries: Research and Development*, pages 124–136. Springer, 2007.

[25] Priscilla Walmsley. *Definitive XML Schema.* Prentice Hall, 2001.

[26] Nora Yahia, Sahar A. Mokhtar, and Abdel Wahab Ahmed. Automatic Generation of OWL Ontology from XML Data Source. *International Journal of Computer Science Issues*, 9(2):77–83, 2012.

[27] Mustafa Yüksel. *A Semantic Interoperability Framework for Reinforcing Post Market Safety Studies.* PhD thesis, Middle East Technical University, 2013.