Towards Logic Based Representation of XML Models

Călin Jebelean*, Ciprian-Bogdan Chirila*, Vladimir Creţu* and Marieta Făşie*
*University Politehnica of Timişoara, Romania
Email: {calin,chirila,vcretu}@cs.upt.ro, marietta_or@yahoo.com

***Abstract*—Both code analysis and code transformation are processes that rely on software models instead of actual software systems. In the context of software modeling, we have done so far some efforts to attach logic representation to programs written in any language by using an automatic and grammar-driven approach. However, XML proved to be a difficult candidate for such an approach, because we discovered the XML format is already close to the logic format that we desired, and running our generic grammar-driven approach on XML files would add unnecessary complications. Therefore, we imagined a different technique for transforming XML files into logic models, a technique that preserves useful information already present in XML files. As a benefit of the approach we will show how UML models (also described in XML) can be transformed into logic models and analyzed or transformed further at a logical level.**

I. Introduction

Representing programs as logic factbases (Prolog-like) has become over the years a strong approach to support processes like program analysis and program transformation ([Ciu99], [TM03]). Analyses and transformations of programs can be described in a much more expressive manner than that of an imperative language.

The idea of logic representation for programs has led to the development of ProGen (PROlog GENerator), a fully automatic tool capable of constructing logic representations for programs written in any programming language ([CJM08], [JCM08]). ProGen is equipped with a grammar repository for several programming languages and the process of building a logic representation for a program is actually driven by the grammar of that program. Thus, using the approach for different programming languages does not involve writing a new tool for each new language, but rather configuring the same tool with another input grammar.

In this paper we address the problem of building logic models for XML files. When we configured ProGen with the XML grammar and used it to parse XML files and build logic models for them we realized the result is in each case far more complicated than the initial XML file. Normally, an XML file uses a hierchical structure to present information: there are XML tags (parents) that encapsulate other XML tags (children). ProGen models also have a hierarchical structure, driven by the grammar of the language. Thus, parent entities (non-terminals of the grammar) are linked to child entities (non-terminals or terminals) as specified by the grammar. Each grammar rule explicitly defines links between several child entities (members of the right side of the rule) and a single parent entity (the left side of the rule). The two hierarchies have very few in common since ProGen describes hierarchies at the syntactic level while XML depicts them at the textual level. The need arised for ProGen to perform different on XML files, such that the textual hierarchy is considered instead of the syntactic hierarchy, thus alleviating the usage of the generated logic model. This will be shown next.

XML is the de-facto standard for sharing information between different applications. It is not a surprise that UML tools use XML as the preferred format to export their artifacts. Thus, if ProGen is used to translate such UML artifacts (described in XML) to the realm of logic, the analysis and transformation steps that we mentioned at the beginning of this section could very well be used on them. Thus, analysis and transformation of UML models is achieved. This is the main reason why we even considered XML for this discussion.

Analysis of UML models is not a very common research topic and there are reasons for that. Analyzing models has the great disadvantage of not being able to grasp all the information about the system being analyzed, because models are only abstractions of the system. Thus, model-based analyses can't expect to be more successful than code-based analyses ([Mar04], [SSL01], [Ciu99], [TM03], [Jeb04], [JLB02]). Still, model-based analyses have one major advantage. They can be applied early in the development phase and eventual problems could be solved before the coding phase even begins, thus saving time and money. This is the reason why we believe there is some potential in performing analyses directly on UML models, even if the analyses are limited in power or depth.

The next sections are structured as follows: section II presents an overview of the logic based representations performed by ProGen, section III shows how certain modeling tools like ArgoUML or Enterprise Architect describe UML models using XML and also how ProGen should be adapted to better benefit from the information already present in the XML format, section IV describes how the generated ProGen model can be used to perform analysis on UML models (even if XML was the modeled language), and section V concludes.

II. Overview of Logic Based Representations

In this section we will briefly present how ProGen generates its logic models for programs written in any language. We

will study an example for that. Figure 1 contains a little piece of code written in a C-like language. The grammar is very simple, instructions between curly braces are only allowed to be assignments and expressions can only be built by using identifiers, numbers, the four basic arithmetic operators and parantheses.

```
{
  b = 4;
  a = ( 8 + b ) * 5;
}
```

Fig. 1. A sample program

ProGen is a language independent tool. Thus, when dealing with a program like the one in figure 1 it needs the grammar for the respective language. We present the grammar in figure 2 using an EBNF notation ([ebn]).

```
Block ::= '{' InstrList '}'

InstrList ::= ( Instr ';' )*

Instr ::= Assignment

Assignment ::= <ident> '=' Expr

Expr ::= Term ( AdditiveOp Term )*

Term ::= Factor ( MultiplicativeOp Factor )*

AdditiveOp ::= '+' | '-'

Factor ::= <ident> | <constant> | '(' Expr ')'

MultiplicativeOp ::= '*' | '/'
```

Fig. 2. A sample grammar

The Prolog metamodel is listed in figure 3. The clauses of the metamodel are simply non-terminals of the grammar only in downcase, since Prolog treats identifiers starting with an uppercase as variables. The father-son relationships are modeled by using numeric identifiers for each clause. Thus, each clause is associated with a numeric identifier which is unique in the system and also with the numeric identifier of its parent clause, as specified by the grammar. The first clause will have a special value as the parent identifier (-1) since that clause will normally have no parent.

```
block(#ID, -1).
instrlist(#ID , #blockID).
instr(#ID , #instrlistID).
assignment(#ID , #instrID).
expr(#ID , #assignmentID).
term(#ID , #exprID).
additiveop(#ID , #exprID).
factor(#ID , #termID).
multiplicativeop(#ID , #termID).
```

Fig. 3. The Prolog metamodel

Terminals of the grammar will be modeled by using a special clause, called *atom*. The first parameter of such a clause will be the numeric identifier of the parent of that atom and the second parameter will be the actual value of the atom.

The output of ProGen will be a Prolog knowledge base that models the code in figure 1 using clauses from the metamodel in figure 3 as dictated by the grammar in figure 2. A listing of that knowledge base is presented in figure 4 and an equivalent graphical representation is depicted in figure 5. The dotted line follows the AST border and visits the atoms in the actual order they were discovered in the input file.

```
block(10000, -1).
atom(10000,'{').
instrlist(10001,10000).
instr(10002,10001).
assignment(10003,10002).
atom(10003,'b').
atom(10003,'=').
expr(10004,10003).
term(10005,10004).
factor(10006,10005).
atom(10006,'4').
atom(10001,';').
instr(10007,10001).
assignment(10008,10007).
atom(10008,'a').
atom(10008,'=').
expr(10009,10008).
term(10010,10009).
factor(10011,10010).
atom(10011,'(').
expr(10012,10011).
term(10013,10012).
factor(10014,10013).
atom(10014,'8').
additiveop(10015,10012).
atom(10015,'+').
term(10016,10012).
factor(10017,10016).
atom(10017,'b').
atom(10011,')').
multiplicativeop(10018,10010).
atom(10018,'*').
factor(10019,10010).
atom(10019,'5').
atom(10001,';').
atom(10000,'}').
```

Fig. 4. The Prolog model

III. XML Representation of UML Models

As mentioned earlier, XML proved to be a difficult candidate for the approach presented in section II since XML already contains a hierarchy of entities (tags) at the textual level. ProGen translates XML to Prolog by using the hierarchy of entities specified in the XML grammar, and the two hierarchies are not always compatible because the XML grammar introduces unnecessary levels of indirection between entities. This is not an issue for a random programming language where no other hierarchy can be derived, but is annoying for XML and we decided to write a special version of ProGen that will only deal with translating XML content to Prolog.

A. Translation of basic XML files

We will study the translation details on a comprehensive example. Figure 6 contains an example of basic XML content.

Regardless of the XML grammar, the hierarchy is obvious from the textual level. Thus, the *course* entity descends from the *courses* entity and *name*, *year* and *students* descend from *course*. Each XML tag will be translated as a Prolog clause

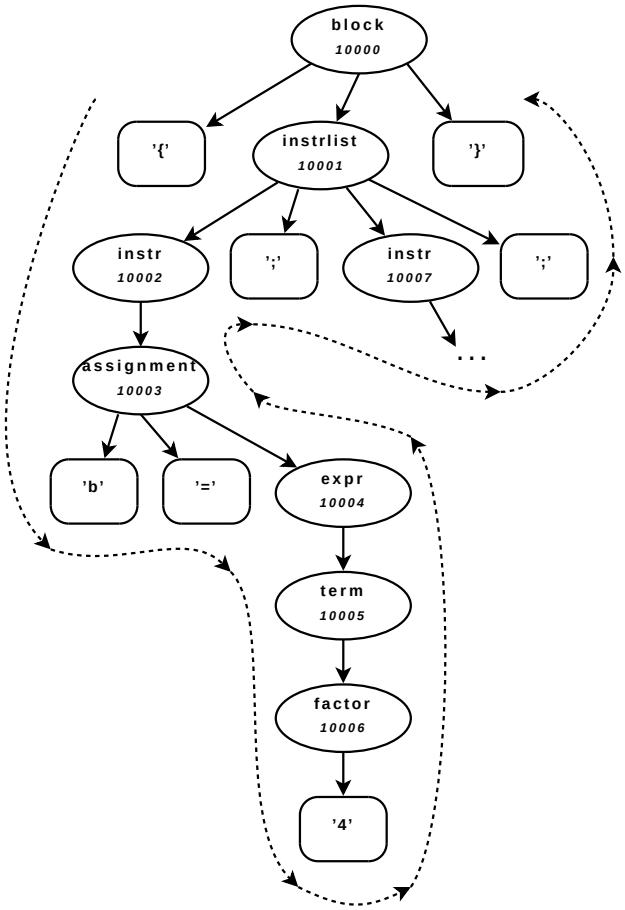

Fig. 5. AST representation

```
<courses>
  <course id=100>
    <name>
      Computer graphics
    </name>
    <year>
      2
    </year>
    <students min=30 max=60 />
  </course>
  <course id=101>
    <name>
      Artificial intelligence
    </name>
    <year>
      3
    </year>
    <students min=40 max=80 />
  </course>
</courses>
```

Fig. 6. A sample XML file

and they will be linked by numeric identifiers according to the XML hierarchy. If an XML tag has arguments, they will be added as further arguments in the corresponding Prolog clauses. The Prolog output is presented in figure 7.

Since the XML to Prolog translator is custom-built and not a particular case of a generic translator (like ProGen), we decided to add bidirectional navigation in Prolog. This is not available in normal ProGen output, where only navigation

```
courses(10000, -1, [10001, 10005]).
course(10001, 10000, [10002, 10003, 10004], '100').
name(10002, 10001, [], 'Computer graphics').
year(10003, 10001, [], '2').
students(10004, 10001, [], '30', '60').
course(10005, 10000, [10006, 10007, 10008], '101').
name(10006, 10005, [], 'Artificial intelligence').
year(10007, 10005, [], '3').
students(10008, 10005, [], '40', '80').
```

Fig. 7. The Prolog model (XML)

from children to parent is easily achieved. Bidirectional navigation in figure 7 can be observed, for example, in the second clause:

course(10001, 10000, [10002, 10003, 10004], '100').

10001 is the numeric ID of the *course*, *10000* is the numeric ID of the parent and the following list contains the IDs of the *course*'s children. They belong to a *name* clause, a *year* clause and a *students* clause, thus allowing navigation from the *course* to its children (downwards) and also from the *course* to its parent (upwards).

B. Translation of XMI files

All topics about modeling XML in Prolog are aimed at a greater purpose. XML is the best file format to exchange information between application and thus it comes as no surprise that UML modeling tools use XML as a suitable way to share data. For that, a special XML dialect was introduced by the Object Management Group, called XMI. XMI is still XML, but its main purpose is as an interchange format for UML models. All major UML modeling tools have the option of exporting UML models to XMI. The XML to Prolog translator comes then as a first step in a greater process of logic based analysis of UML models, which is the whole purpose of the approach we present here.

An exhaustive presentation of XMI is beyond the scope of this article. However, for clarity purposes we should present a small excerpt from an XMI file and the details of translating XMI to Prolog. Although XMI is still XML and translation of XML to Prolog was covered in the previous subsection, there are a few issues that have to be treated with special attention when dealing with XMI files.

Figure 8 depicts a simple class diagram with 4 classes, modeled in ArgoUML ([arg]): *Shape*, *Line*, *Circle* and *Square*, with *Shape* being the superclass and the others being subclasses.

Figure 9 presents a small excerpt from the XMI document exported from the class diagram in figure 8. The excerpt displays only the XMI content regarding classes *Shape* and *Circle* and the inheritance relation between them.

Figure 9 shows several interesting aspects:

- XML tag names have the common prefix *"UML:"*. There are 2 problems with this prefix that prevent a naive Prolog translation to be valid: first, it starts with a capital letter and second, it contains a colon – thus, the Prolog translator should be instructed to convert all letters to lowercase and replace all non-letter and non-digit characters with an underscore character: '_'. According to this rule, *UML:Class* will be translated in Prolog as *uml_class*

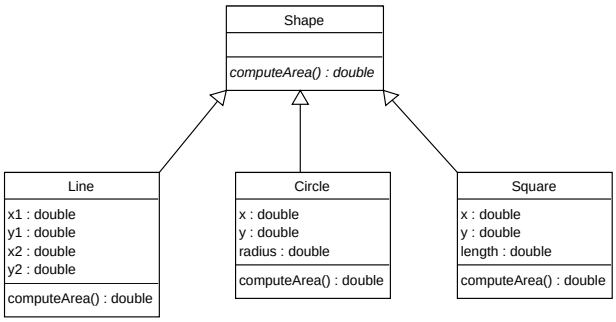

Fig. 8. A sample class diagram

```
<UML:Model xmi.id='B36' name='ClassDiagram'>

  <UML:Class xmi.id='CDE' name='Shape' visibility='public'>
    ...
  </UML:Class>

  <UML:Class xmi.id='CE1' name='Circle' visibility='public'>
    <UML:Attribute ...> ... </UML:Attribute>
    <UML:Operation ...> ... </UML:Operation>
  </UML:Class>

  <UML:Generalization xmi.id='CEB'>

    <UML:Generalization.child>
      <UML:Class xmi.idref='CE1'/>
    </UML:Generalization.child>

    <UML:Generalization.parent>
      <UML:Class xmi.idref='CDE'/>
    </UML:Generalization.parent>

  </UML:Generalization>

  ...

</UML:Model>
```

Fig. 9. An XMI excerpt

- Some of the XMI entities have their own unique ID called *xmi.id* and represented as a string value [1]. For example, the *UML:Model* entity has an associated ID 'B36'. There are other XMI entities that don't have personal IDs, like *XMI:Generalization.child*, for instance. It is not recommended to ignore these already generated IDs because all the relations between XMI entities are described using them (take, for example, the inheritance relationship between *Shape* and *Circle*). As a result, the Prolog translator will be configured to be aware of these IDs
- Some XMI entities have an *xmi.idref* argument. For example, when defining the inheritance relationship in figure 9, the two entities that are part of this relationship are referred using the *UML:Class* tag, but instead of having an *xmi.id* argument (which would have meant a new class declaration), they have an *xmi.idref* argument (which means that the respective class has already been defined earlier and the current entity is only a reference to that class)

[1]The actual string values for the XMI IDs are much longer than 3 characters, but we truncated them to the last 3 characters because of space limitations

According to these aspects, the general translation algorithm is sketched below:

1) The XMI tree is visited in preorder and each node receives a unique numeric ID - the uniqueness could be achieved by incrementing a variable after it is assigned to the current node, for example
2) During the same visit, each node receives a suitable name, as mentioned earlier, by converting the original name of the node to lowercase and replacing unwanted characters with the '_' character
3) The XMI tree is visited the second time in preorder and each XMI node will generate a Prolog clause, as specified in subsection III-A.
4) The name of the Prolog clause is the suitable name received at step 2)
5) The first parameter of the Prolog clause is the numeric ID of the XMI node received during step 1), unless the XMI node has an *xmi.id* argument, in which case the value of this argument is used as the first argument of the Prolog clause. If the XMI node has an *xmi.idref* argument, then the same rule applies but furthermore, the name of the Prolog clause is appended with the *_ref* suffix
6) The second parameter of the Prolog clause is the numeric ID of the parent of the XMI node, with the same exception as above
7) The third parameter of the Prolog clause is the list of numeric IDs of the children of the XMI node, with the same exception as above
8) The next parameters of the Prolog clause are the rest of the arguments of the XMI node

According the the previous algorithm, the Prolog translation of the XMI excerpt in figure 9 is presented in figure 10.

```
uml_model('B36', -1, ['CDE', 'CE1', 'CEB'], 'ClassDiagram').
uml_class('CDE', 'B36', [...], 'Shape', 'public').
uml_class('CE1', 'B36', [...], 'Circle', 'public').
uml_generalization('CEB', 'B36', [10000, 10001]).
uml_generalization_child(10000, 'CEB', ['CE1']).
uml_class_ref('CE1', 10000, []).
uml_generalization_parent(10001, 'CEB', ['CDE']).
uml_class_ref('CDE', 10001, []).
```

Fig. 10. The Prolog model (XMI)

IV. Benefits: Analysis of UML Models

Once the XMI translation in Prolog is achieved, we can use the Prolog model to perform a few analyses on the initial UML model that generated the XMI representation. For example, it is easy to detect long inheritance chains in Prolog since an inheritance chain is composed of several *uml_generalization* clauses linked one after another by means of their IDs.

Another easy-to-do analysis is the detection of large classes, which is presented as a bad smell in [Fow00]. Class content is available in XMI, but figure 9 failed at showing it because of space limitations. However, method and attribute information for classes is available inside *UML:Class* tags and can serve to

detect either classes that have a large amount of methods (god classes) and very few attributes or classes that have a large amount of attributes (data classes) and very few methods that are not setters and getters. These amounts can be computed either as absolute values, or relative to each other.

Still, it is not our intention to describe UML model analysis using Prolog in this article. This subject will be treated in greater detail in a future paper.

V. Conclusions and Future Work

In this paper we have proposed a solution to perform analysis on UML models. As a first step, we have observed that UML modeling tools use XML as a standardized way of sharing information. Next, we used our general purpose translator called ProGen to obtain a Prolog representation for XML input. Since the Prolog representation we obtained was not quite satisfactory for reasons we have already presented in the paper, we have modified ProGen to produce a more suitable output on XML input files. Thus, we believe the translator will be ready to perform on XMI files, an XML interchange format for UML models.

The XML to Prolog translator is already under development. As future work, we plan to finish the direct translation and start working on a correspondent Prolog to XML convertor. In the mean time, we also plan to implement as many model analyses as possible and assess their accuracy on large scale models.

References

[arg] Argouml. http://argouml.tigris.org.

[Ciu99] Oliver Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of the Technology of Object-Oriented Languages and Systems*. IEEE Computer Society, 1999.

[CJM08] Ciprian-Bogdan Chirilă, Călin Jebelean, and Anca Măduţa. Towards automatic generation and regeneration of logic representation for object-oriented programming languages. In *Proceedings of the International Conference on Technical Informatics*, University Politehnica Timişoara, 2008.

[ebn] The ISO/IEC 14977:1996(e) standard.

[Fow00] Martin Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley, 2000.

[JCM08] Călin Jebelean, Ciprian-Bogdan Chirilă, and Anca Măduţa. Generating logic based representations for programs. In *Proceedings of the 2008 IEEE 4th International Conference on Intelligent Computer Communication and Processing*, pages 145–151, Cluj-Napoca, Romania, 2008.

[Jeb04] Călin Jebelean. Automatic detection of missing abstract-factory design pattern in object-oriented code. In *Proceedings of the International Conference on Technical Informatics*, Politehnica University in Timişoara, 2004.

[JLB02] Sang-Uk Jeon, Joon-Sang Lee, and Doo-Hwan Bae. An automated refactoring approach to design pattern-based program transformations in Java programs. In *Proceedings of the 9th Asia-Pacific Software Engineering Conference*, 2002.

[Mar04] Radu Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 350–359. IEEE Computer Society, 2004.

[SSL01] Frank Simon, Frank Steinbruckner, and Claus Lewerentz. Metrics based refactoring. In *Proceedings of the Conference on Software Maintainance and Reengineering*, pages 30 – 38, 2001.

[TM03] Tom Tourwé and Tom Mens. Identifying refactoring opportunities using logic meta programming. In *Proceedings of the 7th European Conference on Software Maintainance and Reengineering*, pages 91–100. IEEE Computer Society, 2003.