Translation of Visual Prolog programs into UML models

Ivan Andrianov
System Programming Department,
Computational Mathematics and Cybernetics Faculty,
Moscow State University
andrianov.sp.cmc@gmail.com

1 Introduction

Modification and support of systems written in logical programming languages is a difficult task because of the lack of software engineering tools for logical programs. Currently there are a lot of software engineering tools for object-oriented programs. Therefore the translation of a logical program into an object model task is really urgent. The task solution allows us to get models corresponding to logical programs. For these object models we can use object-oriented software engineering tools in full, e.g., modify object models and generate the object-oriented language code from them.

1990s were the time of the active research on the logical programs analysis and modification topic. In that time algorithms resolving the predicate argument type inference (e.g., GAIA [1]) and the predicate mode inference (e.g., MDDAA [2]) problems were developed for standard Prolog programs. These algorithms were used for Prolog interpreter and compiler optimization purposes. In this article logical programs are analyzed for other purposes. The task is a transition from logical programs terms to object models terms. The 1990s algorithms can't applied for the task.

In this article we deal with logical programs written in the Visual Prolog [3] language. Consider features of Visual Prolog differing it from standard Prolog dialects.

```
1  domains:
2    department_type = financial, logistics, technical.
3    date = d(integer, integer, integer).
4    person = p(string, date).
5    department = dep(string, department_type).
6  ...
7  predicates:
8    male(?person).
9    age(+person, ?integer).
10   works(?person, ?department).
11 ...
12 clauses:
13   male(p("Andrey", d(21, 02, 1986))).
14   age(p(_, d(_, _, Year)), Age) :- Age is 2012 - Year.
15   works(p("Andrey", d(21, 02, 1986)), dep("ISP RAS", technical)).
16   works(p("Elena", d(13, 01, 1987)), dep("ISP RAS", technical)).
17 ...
```

Listing 1: The Visual Prolog program example.

The listing 1 program is a model database containing information about employees and their departments. We can query the database to get all male employees working in the technical department as follows *works(P, dep("ISP RAS", technical)), male(P)*. The query result is P = *p("Andrey", d(21, 02, 1986))*. The program is split to different sections: *domains*, *predicates*, *clauses*. The program types are declared in the *domains* section. The type is a list of terms-constants (e.g., *department_type* at the line 2), a functor with defined argument types (e.g., *person* at the line 4) or a list type (*element_type**). The program predicates with their argument types are defined in the *predicates* section. Each argument has a mode: *+* for input arguments, *-* for output arguments and *?* for input-output arguments. Facts and rules for predicates are placed in the *clauses* section.

We created a list of rules providing the translation of logical programs into UML [4] models and implemented an automatic system performing the translation accordingly to these rules.

The created translation rules were designed to provide conservation of program semantics in the model. It means that there are translation rules of program query into model query so that the results of both queries are equal.

The translation rules are grouped by their destination. The first group contains rules related to types, the second one — to predicates, the third one — to queries.

2 Type translation rules

- String and number primitive types must be converted to the corresponding UML type.
- Each primitive type consisting of term-constants must be converted to an enumeration type with the same group of constants. For example *department_type* at the line 2 is converted to the *DepartmentType* enumeration.
- Each type corresponding to a set of terms with a common functor and arity must be converted to a class. Each primitive-typed argument must be converted to a class attribute. Each user-typed argument must converted to a directed association between corresponding classes. For example *person* at the line 4 is converted to the *Person* class, the string argument — to the attribute *arg1*, the *date*-typed argument — to the directed association between *Person* and *Date* classes.
- Each list type with a primitive element type must be converted to a class with an attribute of the corresponding type with the multiplicity ***.
- Each list type with a user element type must be converted to a class linked to the element class by an association with the multiplicity *** at both ends.

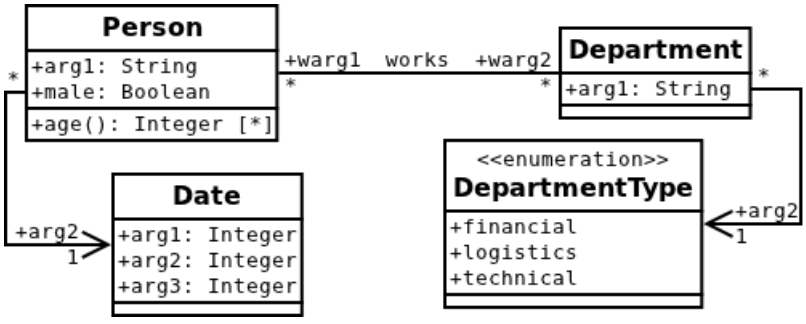

Figure 1: Classes of the model corresponding to the example program.

3 Predicate translation rules

Predicates defined by facts without variables are further called *fact*-predicates. Other predicates are further called *rule*-predicates.

- Each *fact*-predicate having two or more user-typed arguments and no primitive-typed arguments must be converted to an association between the argument classes with the multiplicity * at both ends. A link is set up between objects corresponding to the fact argument terms. For example *works* at the line 10 is converted to the *works* association between *Person* and *Department* classes. The link is set up between *p1* and *dep1* (line 15), *p2* and *dep1* (line 16).

- Each *fact*-predicate having one user-typed argument and no primitive-typed arguments must be converted to a boolean attribute of the argument class. The attribute is *true* for objects corresponding to the fact argument terms and *false* for others. For example *male* at the line 8 is converted to the boolean *male* attribute of the *Person* class. The attribute is *true* for the *p1* object (line 13) and *false* for the *p2* object.

- Each *rule*-predicate having one primitive-typed and one user-typed argument must be converted to an operation of the argument class with the return type of list of the corresponding primitive type. The operation body written in the OCL [5] language must be generated by analogy with the query generation for right parts of the predicate rules (see section 4). For example *age* at the line 9 is converted to the *age* operation of the *Person* class with an OCL description *context Person::age(): Bag(Integer) body: Bag 2012 - self.arg2.arg3* (line 14).

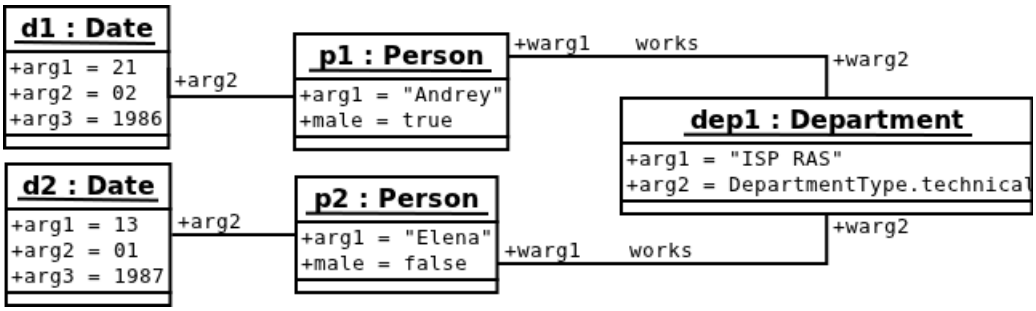

Figure 2: Objects of the model corresponding to the example program.

- Each *fact*-predicate having two or more user-typed arguments and at least one primitive-typed argument must be converted to the association class between the argument classes with the multiplicity * at both ends. Each primitive-typed argument must be converted to an attribute of the association class with the corresponding type. The link is set up between the fact argument terms.

- Each *rule*-predicate having one primitive-typed and one user-typed argument must be converted to an attribute of the argument class with the corresponding type and the multiplicity *.

- Each *fact*-predicate having one user-typed argument and two or more primitive-typed arguments must be converted to an association between the argument class and an auxiliary class with the multiplicity * at both ends. Each primitive-typed argument must be converted to an attribute of the auxiliary class with the corresponding type. The link is set up between the fact argument terms.

- Each *fact*-predicate having no user-typed arguments and any count of primitive-typed arguments must be converted to a class. Each argument must be converted to an attribute of the class with the corresponding type.
- Each *rule*-predicate having one user-typed argument and no primitive-typed arguments must be converted to a boolean operation of the argument class. The operation body written in OCL must be generated by analogy with the query generation for right parts of the predicate rules (see section 4).
- Each predicate of any other type must be converted to an operation. The return type of the operation is boolean if the predicate has only input arguments and bag of tuples with element types corresponding to the output and input-output predicate arguments. If the predicate has at least one user-typed argument then the operation is added to the corresponding class and the argument is removed from the operation (becomes the implicit self argument). If the predicate has at least one output or input-output argument the operation is added to the corresponding class as static. Otherwise the operation is added to the auxiliary class *Predicates*. The non-deterministic choice of a class is resolved by a metric *class with the least count of operations*. The operation body written in OCL must be generated by analogy with the query generation for right parts of the predicate rules (see section 4).

4 Query translation algorithm

Consider a query getting all male employees of the given department *works(P, dep("ISP RAS", technical)), male(P)*. As it was said above the query result is *P = p("Andrey", d(21, 02, 1986))*. The query from the listing 2 corresponds to the given Prolog query. Consider the structure of the given OCL query. There we iterate over all instances of the *works* association and the *Person* class. The choice of collections is caused by the predicates from the Prolog query. Such and only such instances of the *Person* class that are the first argument of any instance of the *works* association, the second argument of which is the *dep1* object of the *Department* class, and has *true* as the value of the *male* attribute are added to the result bag. The choice of conditions is caused by the *dep("ISP RAS", technical)* constant, the *P* variable and the *male(P)* predicate. The query result is *Bag* {*Tuple* {*P = @p1*}}. It almost evident that the results of Prolog and OCL queries are equal up to the differences in logical and object terms. Indeed the *p1* object of the *Person* class corresponds to the *p("Andrey", d(21, 02, 1986))* term.

```
works::allInstances()->iterate(e1: works;
    res1: Bag(Tuple(P: Person)) = Bag {} |
  Person::allInstances()->iterate(e2: Person;
      res2: Bag(Tuple(P: Person)) = res1 |
    if e1.warg2 = dep1 and e1.warg1 = e2 and e2.male
      then res2->including(Tuple {P = e2})
      else res2
```

Listing 2: The OCL query.

We have to note that it is possible to construct a simpler equivalent query. It is *dep1.warg1 ->select(male)*. But the main purpose of the query translation algorithm is to be applicable to all queries and to guarantee the equal result property and therefore the preservation of semantics in the translation rules for programs. At this stage of the research in the translation of Visual Prolog programs into UML models domain query complexity and performance problems aren't examined. But in future this branch of the research must be considered.

The logical query result computation is a backtracking procedure. Therefore the corresponding OCL query can be constructed as a collections iteration procedure with filtering. For each query we will examine all its predicates and add needed elements to the iteration list and the filtering condition.

- If the p predicate is converted to the class, association class or association then *p::allInstances()* must be added to the iteration list. The equality of the collection element attributes to constants or variables must be added to the filtering condition.

- If the p predicate is converted to the boolean attribute or the operation without arguments (the predicate with the only user-typed argument corresponding to the T class), the argument contains variables then *T::allInstances()* must be added to the iteration list. The value of the attribute or the operation must be added to the filtering condition. If there is no variables in the argument the iteration list must be the same and the value of the attribute or operation must be added to the filtering condition.

- If the predicate p is converted to the attribute (operation) of the primitive type with the multiplicity * (the predicate with two arguments: one has the user type corresponding to the class T, another is primitive-typed), the user-typed argument contains variables, *T::allInstances()* and p (*p()* for the operation) must be added to the iteration list. If there is no variables in the user-typed argument *o.p* (*o.p()* for the operation) where o is an object corresponding to the user-typed argument term must be added to the iteration list. In both cases the equality of the collection element to constants, variables must be added to the filtering list.

- Otherwise if the p predicate has at least one output or input-output argument $o_1.p(o_2, \ldots, o_n)$ or $T :: p(o_1, \ldots, o_n)$ where $o_1, \ldots, o_n$ are terms corresponding to the input predicate arguments, T is a class where the operation corresponding to p is added must be added to the iteration list. The equality of the collection element to constants, variables must be added to the filtering list. If the predicate has only input arguments (therefore the corresponding operation return type is boolean) the iteration list must be the same and the operation call must be added to the filtering condition.

5 Software tool

The developed software tool which provide the translation of Visual Prolog programs into UML models is based on the rules described above.

ANTLR [6] is used to create an analyzer of the input program. This tool gets LL(k) grammars and produces the source code of a program analyzer written in one of languages including Java. During the translation tool development we created a LL(1) grammar of the Visual Prolog language inclusive the program tree description. This grammar was given to the ANTLR tool and the syntactic analyzer source code written in Java was produced.

ANTLR allows its users to create a so-called tree grammar where it is possible to specify the code that handles the given tree node. This code usually creates objects of the program internal representation classes and links these instances among themselves. During the translation tool development we implemented a number of classes describing the internal representation of Visual Prolog programs and created a tree grammar which uses these classes. This grammar was given to the ANTLR tool and the semantic analyzer source code written in Java was produced.

To generate UML models as XMI files we used Eclipse MDT [7]. This tool is one of the best UML 2 metamodel implementations. It is implemented as an Eclipse IDE plugin. During the

translation tool development we implemented a class that analyzed the given internal representation of the program. Firstly the information about types is analyzed then the information about predicates is. If any part of the internal representation matches one of translation rules this class uses Eclipse MDT API to store a new UML model element.

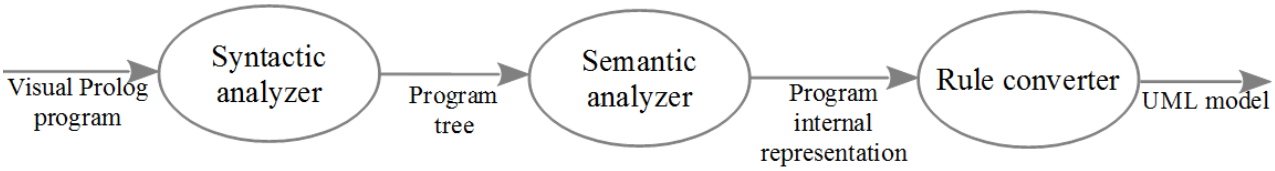

Figure 3: The tool architecture.

The tool architecture is described at the figure 3. We have to note that this scheme allows us to change the translation rule set easily because only the converter module depends on them.

6 Conclusion

Research results consist of the created set of translation of Visual Prolog programs into UML models rules which provide program semantics preservation and the software tool which automates translation based on the rules. The set of the rules includes translation rules for queries so that the query result equality property is true. The research results can be used for re-engineering of programs written in object-oriented dialects of Prolog.

References

[1] Pascal van Hentenryck, Agostino Cortesi, and Baudouin le Charlier. Type analysis of Prolog using type graphs. *Journal of logic programming*, (22):179–209, 1993.

[2] Saumya K. Debray. Static inference of modes and data dependencies in logic programs. *ACM transactions on programming languages and systems*, (11):418–450, 1989.

[3] Randall Scott. *A Guide to Artificial Intelligence with Visual Prolog.* Outskirts Press, 2010.

[4] Grady Booch, Jim Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 2nd edition, 2004.

[5] Jos Warmer and Anneke Kleppe. *Object Constraint Language, The: Getting Your Models Ready For MDA.* Addison-Wesley, 2nd edition, 2003.

[6] Terence Parr. *The definitive ANTLR Reference.* The Pragmatic Bookshelf, 2007.

[7] Richard Gronback. *Eclipse Modeling Project: A Domain-Specific Language Toolkit.* Addison-Wesley, 2009.