Declarative Meta Programming for a Language Extensibility Mechanism

Johan Brichau*
Johan.Brichau@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussel, Belgium

March 24, 2000

1 Introduction

Today, many dialects of traditional programming languages exist [6]. In most cases, they add some programming paradigm or functionality to the original language. A major drawback of these dialects is that they have hard coded the extensions into the compiler.

In an open mechanism, programmers may introduce extensions to the language to fit their own development needs. Otherwise every extension requires a re-implementation of the language compiler, which is not a flexible solution. It has already been demonstrated that meta programming is a solution to this problem [10, 11]. For example, OpenJava is an extensible dialect of Java where the meta programs specify a transformation from the extended language to the original language. These meta programs are normal Java programs that make use of the OpenJava MOP to inspect and modify the code. In such a system, a meta program that adds accessor methods to a class looks more or less like this (in pseudo-code):

```
fields = aClass.getFields();
for(i=1;i < fields.length();i := i+1) {
    method = ...construct new method...
    aClass.addMethod(method) }
```

In this paper, we propose declarative meta programming as a more convenient medium to describe these transformations from the extended to the unextended base language. In casu, we will use a logic language as an instantiation of a declarative language. As suggested in the small example above, a transformation process mostly boils down to 3 phases:

- The retrieval of the parts participating in the transformation process (introspection)
- Performing some computations
- Actually performing the transformation and adding the transformed parts to the base level (absorption and intercession)

Therefore we propose to use a transformation language that benefits from the pattern-matching and unification algorithms present in a logic language to automate this pattern of transformation. Secondly, we believe that for many possible language extensions (e.g. language support for design patterns), one should allow the use of global information in any transformation.

Our approach boils down to a macro-mechanism with declarative meta programming as an expansion mechanism.

*Research Assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

2 Declarative Meta Programming

Declarative meta programming is a meta programming technique where the base programs are represented by logic propositions in the meta level. Using a logic (Prolog-like, [4]) programming environment, one can manipulate and reason about these propositions. We will illustrate the notion of logic propositions representing the structure of a program with an example:

```
superclass(square,graphic)
superclass(circle,graphic)
superclass(Class,object)
```

The above 3 facts state that *graphic* is a superclass of *square* and *circle* and that *object* is a superclass of all classes. *Class* is a logic variable because it starts with a capital letter.

At our lab, several experiments have already been performed that show the expressiveness of this kind of meta programming: SOUL [2] is such a system that has shown us that declarative reasoning about base code is a powerful paradigm. Some of its many uses are writing logic rules to detect certain design patterns or to check the compliance to coding conventions. TyRuBa [1] is another system that uses declarative meta programming, however, here the technique is used to generate base code for a program that was described at the meta level by a set of logic propositions. Our system actually emerged in an attempt to merge these two approaches [12].

2.1 The Meta level Representation

The meta level representation must reify every element of the base language, otherwise some language extensions cannot be expressed. Therefore, we based our experimental representation on the parsetree of the base program: we just chop up the parsetree in different logic propositions. In our experimental system, these propositions have the following form:

```
node(<id>,<kind>,[<list of subterm id's>]).
```

Every proposition is a Prolog-like clause of the name node[1], containing 4 subterms. Hence, every proposition describes the node of the parsetree.

- <id> is a unique identifier of the proposition. This makes it possible to refer to this proposition in another proposition. It is a result from using the parsetree as the base for generating the representation. When following these links throughout the propositions, one can walk down the parsetree.
- <kind> is the name of the construct that it represents, e.g. class, statement, modifier, identifier,...
- <list of subterm id's> is a list of identifiers of references to the other propositions that are a subparts of this proposition. e.g.: a class will contain a reference to propositions that describe its modifiers, methods, superclass, etc...

Hence, a program is represented by a database of logic propositions.

3 The Transformation Language

By means of a macro-like specification mechanism, we can add new syntactic elements to the language and specify transformations from the program in the extended language to the corresponding program in the unextended base language. There are two aspects about the definition of an extension, the first is the part that extends the syntax and hence, also the parser. The second part is the meta program which implements the transformations.

[1] Every proposition has been named "node" because of practical details. This system was build for experimental reasons, the "node"-representation was easy to use because of the genericity it presents: every proposition has the same form. This facilitated some implementation aspects.

3.0.1 Syntax Extensions

Our experiments have been conducted in a Prolog environment, where one has access to the definite clause grammars. These allow a programmer to write down grammar production rules which closely match the Context Free Grammar rules. The most interesting feature, is that these production rules in Prolog actually give us a complete parser, without any extra work. In this notation, it is very easy to extend a parser with extra production rules in order to add new syntactic elements to the language.

It is possible to implement an extensible parser using other techniques. One of these is discussed in [9].

3.0.2 Transformations

The transformations actually are logic rules that describe a mapping from the base program's representation to the resulting program's representation. A transformation declaration boils down to the following:

> **FROM** <abstract syntax>
> **TO** <abstract syntax>
> **WHEN** <conditions/logic meta program>

This transformation always converts some kind of abstract syntax construction to another (or the same) abstract syntax construction only when the conditions hold or the logic meta program can be successfully executed. Actually, the WHEN-part of the transformation contains queries that are launched in the logic environment containing the meta level representation. It can be used to investigate the representation of the base program, to call other logic programs to make computations or to use some primitives that are native to the transformation system (this will be explained later on). Every transformation has access to the entire meta level representation of the program, enabling changes based on non-local information.

A small example:

> **FROM** automatic(interface(Modifiers,identifier(anInterface),Extends,Body))
> **TO** interface(Modifiers,identifier(aNewInterface),Extends,Body)
> **WHEN** node(Id,class,[ModifiersId,NameId,ExtendsId,BodyId]), node(NameId,identifier,aClassName).

> *This transformation specifies that an abstract syntax, which is a logic term, that matches the abstract syntax description in the FROM-clause should be replaced by the abstract syntax in the TO-clause. Mind that variables are noted with a starting capital letter, which means that "Modifiers" is a variable that will be bound to the actual abstract syntax of the modifiers and it's value will be present in the new abstract syntax. The name of the interface should be "anInterface" and the new name will be the "aNewInterface". This transformation only occurs when there exists a class called "aClassName".*

The advantage of these transformations is that they are logic rules that detect some patterns in the code and transform these into some other patterns. They rules make explicit use of pattern-matching and unification. One might wonder why we are reasoning about abstract syntaxes. Clearly, this is exactly what we want to do: transform a program in an extended syntax to the traditional syntax of the programming language.

3.0.3 Primitives and other meta programs

In the meta program part of the transformations, the programmer can use some primitives which not only allow more convenient meta programming, but are often necessary to express certain transformations. A short overview:

- *doassert*: A meta program should be able to construct new statements that need to be added to the base program. Therefore, the doassert primitive takes a description of a part

of an abstract syntaxtree as its argument, generates the necessary meta level representation for it and outputs a reference to it. This is how one generates new code that should be inserted somewhere in the program. This is a necessary primitive since our transformations do not allow to dynamically generate code. The TO-clause in the transformations is a static structure, without the doassert primitive, it is impossible to specify a transformation where a list has to be generated whose length can only be determined at transformation-time.

- *dolookup*: This primitive can be used to search for a certain pattern in the program's structure. It makes it easier for the meta programmer to investigate the meta level representation since the pattern should be described as an abstract syntax.

- *env_add, env_lookup*: In a transformation rule, we have no means for knowing the position of the part we are transforming in the entire program. Sometimes, transformation rules should only occur at certain positions. Therefore, we introduce an environment that is available in each rule through the primitives env_add and env_lookup. It is statically scoped with respect to the transformation of the base program. In other words, bindings that were added to the environment when transforming a certain construct (e.g. a class), remain present in the environment of the transformations of the sub-constructs (e.g. methods, data members of this class) and are no longer present in the environment of the transformations of other constructs (e.g. other classes).

- The programmer can also write logic programs that can be used or called from the transformation's WHEN-part. We already included some small programs that lookup all the methods of a class, lookup all the classnames, ...

3.0.4 The Transformation Process

We define a 2-layered system where one layer contains the representation of the base program as it was parsed (containing extra syntactic constructions) and where another layer contains the representation of the base program as it should be generated by our system. We can use the transformation rules, specified by the programmer, to "copy" the propositions from the first layer to the second. The meta level representation is important to allow easy detection of the propositions that need to be transformed. Therefore, using an abstract syntaxtree as basis for our representation makes it fairly easy to detect the abstract syntaxes that are the trigger for a transformation (the FROM-clause in the transformation specification). This means that every transformation is triggered by the existence of some part of the abstract syntax in the representation.

4 Experiment: Automatic Interfaces

In this extension, a class will automatically be a subtype of an interface if this interface was declared to be automatic and if the class defines the necessary methods. An interface that can be automatically implemented by a class must be proceeded with the 'automatic' keyword. The transformations we need to consider in order to implement the automatic interfaces are:

- *Omitting the keyword "automatic" proceeding an interface declaration*:

```
FROM automatic(interface(Modifiers,InterfaceName,Definitions,Super)).
TO   interface(Modifiers,InterfaceName,Definitions,Super).
WHEN
```

- *Adding the automatic interface's names to the class' implements-clause*:

```
FROM class(Modifiers,identifier(atom(ClassName)),Definitions,Super,Implement).
TO   class(Modifiers,identifier(atom(ClassName)),Definitions,Super,NewImplement).
WHEN automaticInterfaceRule(ClassName,Implement,NewImplement).
```

To complete our transformations, we define the predicate that is used in the second transformation. This predicate investigates if a certain class has all the necessary methods (based on their signature) to implement an automatic interface. The automaticInterfaceRule expands an implements-proposition if necessary.

While these logic programs might seem complicated, once one has learned the declarative programming style, they become easy to read and write. Note that the following program implements the lookup of methods, the generation of their signatures, the comparison and the insertion in a classbody if necessary.

```
automaticInterfaceRule(ClassName,Implement,NewImplement) :-
   methods(ClassName,ClassMethods),
   findall([InterfaceName,InterfaceMethods],
           automaticInterfaceMethods(InterfaceName,InterfaceMethods),
   Result),
   doMatching(Result,ClassMethods,Implement,NewImplement).

doMatching([],_,_I,_I).
doMatching([[InterfaceName,InterfaceMethods] | Rest],ClassMethods,
                                    OldImplement,NewImplement) :-
   doAutomaticMatch(InterfaceMethods,ClassMethods),
   doassert(identifier(atom(InterfaceName)),InterfaceRef),
   doMatching(Rest,ClassMethods,OldImplement,Result),
   listcons(InterfaceRef,Result,NewImplement).
doMatching([_|Rest],ClassMethods,OldImplement,NewImplement) :-
   doMatching(Rest,ClassMethods,OldImplement,NewImplement).

automaticInterfaceMethods(InterfaceName,Methods) :-
   node(automatic,_,[InterfaceN]),
   node(interface,InterfaceN,[_,InterfaceNameN,_,BodyN]),
   node(atom,InterfaceNameN,InterfaceName),
   node(list,BodyN,DefsListN),
   lookup_methods(DefsListN,Methods).

doAutomaticMatch([],_).
doAutomaticMatch([Method1 | Rest],OtherMethods) :-
   doAutomaticMatch_sub(Method1,OtherMethods),
   doAutomaticMatch(Rest,OtherMethods).

doAutomaticMatch_sub(Method,[Car | Rest]) :-
   modifiers(Method,MethodModifiers),
   modifiers(Car,CarModifiers),
   compare(CarModifiers,MethodModifiers),
   signature(Method,MethodSig),
   signature(Car,CarSig),
   MethodSig = CarSig.

doAutomaticMatch_sub(Method,[Car | Rest]) :-
   doAutomaticMatch_sub(Method,Rest).
```

5 Further Work

We observe that our chosen representation is not very easy to use. Walking down the parsetree is not what we want to do in a declarative transformation system. In this position paper, we have chosen a syntaxtree as basis for the representational mapping used in our prototype system. This is a representation that can be straightforwardly used and implemented.

A more semantic mapping might be more useful and clear to understand. It should describe the structure of a program in a more 'human-like' fashion. This means that the mapping should describe the classes, their methods, their variables, etc ... and not the structure of the parsetree. E.g.:

```
class(Stack).
method(pop,Stack).
method(push,Stack).
arguments(pop,Stack,[]).
arguments(push,Stack,[arg(type(Object,0),identifier(Element)]).
returntype(pop,Stack,Element).
returntype(push,Stack,void).
```

The representation above is part of the description of a *Stack* class. It gives an idea about what is meant by a more semantic and easy-to-understand mapping. Further research should look into this topic and try to reach more flexible propositions that describe the representational mapping.

6 Conclusion

In this paper we proposed a declarative meta programming approach for an extendible language implementation. We argued that the transformation process can benefit from the pattern-matching and unification algorithms. Another key element is the availability to investigate the entire meta level representation in every transformation. A necessary improvement is the design of a more semantic meta level representation which should improve the declarative nature of the code investigation.

References

[1] Kris de Volder, *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998

[2] Roel Wuyts, *Declarative Reasoning about the Structure of Object-Oriented Systems*, Programming Technology Laboratory, Vrije Universiteit Brussel. Published in Proceedings of TOOLS-USA '98.

[3] Rateb Abu-Hamdeh, James R. Cordy and Patrick Martin, *Schema Translation Using Structural Information*, Dept. of Computing and Information Science, Queen's University, 1994. Published in Proceedings of CASCON '94.

[4] Peter Flach, *Simply Logical, Intelligent Reasoning by Example*, John Wiley & Sons, 1994.

[5] Don Batory, *Software System Generators, Transformation Systems, and Compilers*, Department of Computer Sciences, The University of Texas.

[6] *Programming languages for the Java virtual machine*. http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html

[7] Isabel Michiels, *Using Logic Meta-Programming for Building Sophisticated Development Tools*, licentiaatsthesis, Vrije Universiteit Brussel, 1998.

[8] R. Wuyts, K. Mens, *Declaratively Codifying Software Architectures Using Virtual Software Classifications*, Programming Technology Laboratory, Vrije Universiteit Brussel. In Proceedings of TOOLS-USA '99.

[9] Yuuji Ichisugi, *Modular and Extensible Parser Implementation using Mixins*, Information Processing Society of Japan, Transaction on Programming, vol. 39 No.SIG 1 (PRO 1) pp.61-69, December 1998. (website: http://www.etl.go.jp/~epp)

[10] Michiaki Tatsubori, *An Extension Mechanism for the Java Language*, Master of Engineering dissertation, Graduate school of Engineering, University of Tsukuba, Ibaraki, Japan, Februari 1999.

[11] Shigeru Chiba, *A Metaobject protocol for C++*, in Proceedings of the OOPSLA-Conference, pages 285-299, October 1995.

[12] Johan Brichau, *Syntactic Abstractions for Logic Meta Programming*, Licentiaatsthesis, Vrije Universiteit Brussel, 1999.