Using MetaScribe to prototype a UML to C++/Ada95 code generator

Conference Paper *in* Proceedings of the International Workshop on Rapid System Prototyping · February 2000

DOI: 10.1109/IWRSP.2000.855209 · Source: IEEE Xplore

2 authors:

Dan Regep

5 PUBLICATIONS **31** CITATIONS

SEE PROFILE

F. Kordon

Sorbonne Université

226 PUBLICATIONS **1,416** CITATIONS

SEE PROFILE

Using MetaScribe to prototype an UML to C++/Ada95 code generator

Dan Regep
PhD student at CS TELECOM,
28, rue de la Redoute, BP 74
92263 Fontanay-aux-Roses Cedex, France
E-mail: Dan.Regep@lip6.fr

Fabrice Kordon
LIP6-SRC
Université P. & M. Curie
4 place Jussieu, 75252 Paris Cedex 05, France
E-mail: Fabrice.Kordon@lip6.fr

Abstract : *The use of program generation from graphical representations like UML is increasing in software projects. The notion of hypergenericity is raising up to improve program generators. This paper presents MetaScribe, a tool designed to build program generators providing guidelines to program generator designers and having enhanced facilities for reusability. An example illustrates the use of MetaScribe: the construction of program generators from UML Class diagrams to C++ and Ada95.*
Key word: *Prototyping, Meta-data description, Semantic transformation, Program generation, Hypergenericity.*

1. Introduction

Program generation from complex graphical representations such as HOOD **[6]** or UML **[13]** is now widely used in Computer Aided Software Engineering Environments (CASEE) such as Rational/Rose **[14]** or Objecteering **[16]**.

Now, most CASEEs propose such functionnalities that ease the development of software, increase its security and reduce its cost. Version management capabilities of program pieces to be inserted in the generated programs are supported at the input specification level (i.e. HOOD or UML). Predefined classes also ease interfacing with the execution environment.

Objecteering, implement an interesting notion: hypergenericity **[4, 5]**. It is a way to parameterize the program generation strategy. Thus, users can define templates that are processed for the generation of programs or documentation from the input specification. The use of hypergenericity is of interest to experiment various code generation strategies, to adapt a code generator to quality criteria, or to refine a program generation strategy. This is becoming an essential issue while more code is being produced automatically (user interfaces, object interfaces etc.).

Implementation of hypergenericity can be achieved using a meta-program generator. Such a tool must provide flexible representation techniques, such as the one found in XML **[10]** or MOF **[11]**. Moreover, transformation rules may be defined at a semantic level. Such rules can be used for the generation of a transformation engine that transforms a data representation into another one.

To experiment distributed application generation techniques and ease their implementation, we have implemented a transformation engine generator: MetaScribe **[9]**.

In this paper, we present the use of MetaScribe in the implementation of a program generator from UML to a target object oriented language. This example assessed the use of such a tool. We also got good evaluations of how it could be of interest to generate a program generator.

Section 2. introduces MetaScribe. Then, Section 3. presents the case study we use in Section 4. to illustrate the use and advantages of MetaScribe before concluding remarks.

2. MetaScribe

2.1. MetaScribe Architecture

MetaScribe is designed to enhance reusability in the implementation of transformation engines. Our main goal is to provide an environment allowing the definition of a transformation semantics separately from the syntactic sugar that has to be applied (Figure 1). Then, discrete syntactic sugar may be applied on one semantically defined transformation (i.e. when generation rules from UML class diagram to OO languages are defined, several OO languages having discrete syntax can be easily targeted).

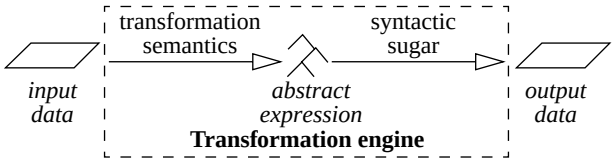

Figure 1 : *Structure of a generation engine.*

The transformation semantics is defined using rules applied on a polymorphic representation of the input data to be processed. These rules, grouped in a *semantic pattern* aim to produce an internal abstract expression semantically describing the output. Semantic constructions in these expressions are represented by semantic constructors on which syntactic rules, grouped in a *syntactic pattern*, are applied to produce the output data in an appropriate format.

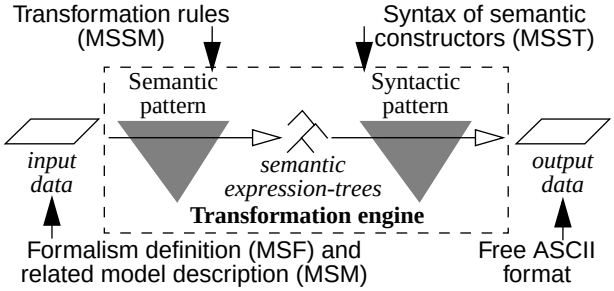

Figure 2 : *Elements to produce a transformation engine.*

MetaScribe requires three elements to be operated (Figure 2):

- **The formalism definition** is expressed using the MSF (**M**eta**S**cribe **F**ormalism) meta-description language. Users declare any entity that can be found in the input data to be processed, MSF allows to type a model description using the MSM polymorphic data description

language. An MSM description is translatable into a memory representation to be manipulated by instructions of the semantic pattern.

- **The semantic pattern** is defined using the MSSM (**M**eta**S**cribe **S**e**M**antic) language. Users declare a set of constructors that are located in semantic expression-trees and define the transformation rules that are applied on models having the format declared in the input formalism.
- **The syntactic pattern** is expressed using the MSST (**M**eta**S**cribe **S**yn**T**actic) language. Users define the syntactic representation associated to constructors declared in the corresponding semantic pattern.

Semantic expression-trees are produced by the semantic pattern and serve as a link with the corresponding syntactic pattern. The semantic expression-trees structure is very similar to the one proposed in the Ada Semantic Interface Specification (ASIS) **[7]**. Moreover, it is polymorphic and can express any concepts. Semantic expression-trees are typically a basis on which a «decompilation» process can be performed to dress expressions with a syntax (a given language instructions or calls to some runtime provided by an execution environment).

MetaScribe generates a transformation engine from a triplet <*input formalism*, *semantic pattern*, *syntactic pattern*>. This transformation engine is implemented as an Ada95 **[1]** program that can be compiled and then able to process any input description in MSM format if it respects the corresponding MSF formalism description.

2.2. Reusability with MetaScribe

The separation of the three aspects manipulated by MetaScribe (input formalism, semantic pattern and syntactic pattern) enables the reuse of components in any of the involved elements.

Let us consider an UML specification described using the MSM data description language (Figure 3).

Let a first semantic pattern be dedicated to the transformation of this UML description into an object-oriented like program. If the OO constructions produced by these rules are not dedicated to some language characteristics, the semantic pattern can be associated with several syntactic patterns (for example, C++, Ada95 and Java).

Input	semantic pattern	syntactic pattern	output
UML descriptions (MSM format)	**Object languages**	C++	*C++ programs*
		Ada95	*Ada95 programs*
		Java	*Java programs*
	Procedural languages	C	*C programs*
		Pascal	*Pascal programs*

Figure 3 : *Reusability in MetaScribe.*

Let a second semantic pattern be dedicated to the production of procedural-like code. Similarly, it could be associated with several syntactic patterns (for example, C and Pascal).

The definition of both semantic and syntactic patterns is very modular. Then, some rules of the OO semantic pattern can be reused in the procedural semantic pattern (like the generation of object's methods). The same observation can be done between some parts of the C, C++ and Java syntactic patterns (idem with some aspects of Ada95 and Pascal).

2.3. Comparison with XML

MetaScribe has many similarities with XML that is now an emerging standard for data interchange. In both cases there is a meta-data description mechanism (DTD versus MSF), associated with a customizable data description language (XML versus MSM).

To format some output, XSL **[17]** provides facilities to express style sheets on an XML description. For example, it is possible to specify the representation of a class of XML documents by describing how an instance of the class is transformed into an XML document that uses the formatting vocabulary. This mechanism is similar to the one provided in MetaScribe syntactic patterns. However, The MSST language provides more flexibility.

MetaScribe management of semantic transformation is also more flexible than the one of XML. XSL mainly proposes a one-pass transformation of the input data. The MSSM language allows navigation over the input data, definition and use of variables, and manipulation on the input data such as the definition of temporary flags. Such functions are suitable to do any kind of transformation, even the one including the use of mechanisms similar to cross-reference tables (i.e. a transformation based on several passes).

The very new draft standard XSLT **[18]** seems to provide more possibilities than XSL. Basically, it allows to group together several XSL descriptions to produce better transformations. However, in current drafts, it appears that some possibilities of MetaScribe (like the management of variables and dynamic flags on the input description) have no equivalent yet.

XML users have to manage themselves reusability at a semantic level. No guidelines are provided. Such guidelines are present in MetaScribe with the separation of semantic and syntactic aspects.

3. A case study

This section illustrates on an example the construction, using MetaScribe, of a code generator from an UML class diagram to an object oriented language. UML class diagrams represent the skeleton above which any object oriented application is built. C++ and Ada95 are both object oriented languages but their phylosophy is quite different. Therefore, the re-factoring of the UML-to-C++ transformation engine from to an UML-to-Ada95 one is not natural and should illustrates the capabilities of MetaScribe.

Implementation of UML descriptions does not represent the main purpose of this paper, which is to illustrate the use of a tool such as MetaScribe. More argued details about generating C++ code from UML models can be found in **[15]**.

The Ada95 generated code is semantically equivalent (as far as possible) to the C++ code in the sense of **[8]**. Another comparison of the Object-Oriented features of Ada95 and Java can be found in **[3]**. It can be used as an extrapolation from Java to C++ due to the similarities of these languages.

We use a small example provided in Figure 4. Associated C++ and Ada95 code fragments are presented in Section 4. We selected this example because it covers most of the possibilities of an UML class diagram. We used it to

validate our code generator. Due to space reasons we will give only fragments of the C++ and ADA95 generated code which exemplify the proposed translation technique.

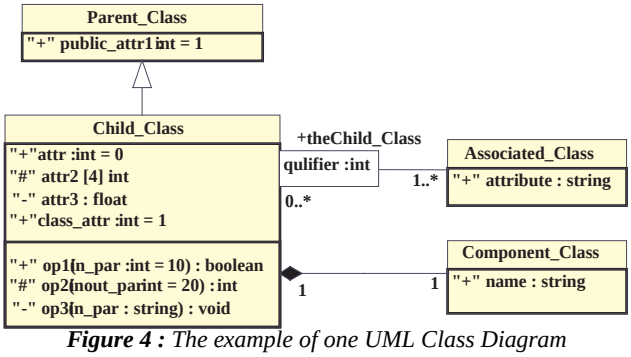

Figure 4 : *The example of one UML Class Diagram*

3.1. From UML to C++

UML classes are directly projected to C++. Our code generation scheme improves the Rational/Rose **[14]** approach:

- All UML instance attributes are projected to equivalent C++ private attributes.
- Access to these attributes is made using two primitives: "*Set*" and "*Get*".
- Primitive visibility is directly derived from the attributes (public, protected or private).
- Class-wide attributes or operations are mapped using "*static*" C++ attributes.

UML operations are also projected to C++ functions. Single or multiple class inheritance is directly supported. Compared with the Rational/Rose UML mapping scheme, we support attributes and operation parameters with an arbitrary multiplicity. Moreover, the mapping of the operations parameters access modes (IN, OUT or INOUT) is also provided. IN access mode parameters are projected to a "by value" C++ parameter. OUT and INOUT access mode parameters are projected to a "by reference" C++ parameters (using pointers).

Associations are simulated using one or more class-instance auxiliary attributes associated to two navigation functions ("*Set_Role*" and "*Get_Role*") named according to the distant associated class role. The type of these auxiliary attributes is built using the Standard Template Library like in Rational/Rose.

There is no difference when mapping associations, aggregations or compositions because we generate only the header files and do not provide any implementation of the C++ Constructor/Destructor functions yet.

3.2. From UML to Ada95

The UML to Ada95 code generation scheme is more difficult due to some semantic differences with UML. In particular, Ada95 does not support multiple inheritance. Our strategy follows the Ada95 semantics by not providing multiple inheritance. A reference about mapping C++ to Ada95 can be found in **[8]** and several workarounds to implement multiple inheritance can be found in **[2]**.

In Ada95 classes are represented by two features: a *package* having a container role and, a *tagged type* used to implement per-instance attributes **[3]**.

We group all per-instance attributes in a *controlled record type* specified in the private part of the package, hidden from other components view.

Equivalence between the UML, C++ and Ada95 visibility is presented in Table 1.

UML	C++	Ada95
public	public	the public part of the package specification
protected	protected	private part of the package specifiation
private	private	package body

Table 1: *Visibility equvalences between UML,C++ and Ada95*

UML per-instance private attributes are represented in Ada95 using an access to an incomplete type defined later in the package body.

We manage access to the per-instance attributes using two primitive operations named "*Get_*" and "*Set_*". The first parameter (named "*This*") of any per-instance operation is an explicit reference parameter to the actual class instance.

Class-scope attributes and operations are declared as normal Ada95 package variables and procedures.

All UML operations are mapped to Ada95 procedures. If the UML operation provide a return value then "*This*" is mapped to an "OUT" access mode parameter named "*returns*".

The proposed implementations do not claim to be in the best ones. However, it appears to be satisfactory for our specific needs. An overview of equivalencies between UML, C++ and Ada95 features is presented in Table 2.

UML features	C++	Ada95
Class	Class	Package and tagged type
Single and multiple inheritance	Direct single and multiple inheritance	Single inharitance using the derived tagged type
Class accesibility	Using # include	Using with and use clause
Class initialisation	Class Constructor and Destructor	Controlled types with Initialize, Adjust and Finalize procedures
Class-instance reference	Implicit through the C++ class-instance reference mechanism	Explicit using an instance parameter named "**This**"
Class attributes	Private atributes and primitive acces functions	Access to a controled record type and primitive access procedures and functions
Class operations	Functions	Procedures
Operation parameters access modes : **IN**, **OUT** or **INOUT**	Call "**by value**" for IN access mode parameters and, "**by reference**" for OUT and INOUT access mode parameters	Using **IN**, **OUT** or **INOUT** parameter access mode

Table 2: *Equivalence between UML, C++ and Ada95 features*

4. Using MetaScribe to experiment the case study

4.1. The MSF representation of UML

UML is a graphical modeling language. To translate it, we use a concrete representation based on a MSF description. UML Class Diagram models are represented using a MSM description in concordance with this MSF description.

To generate programs from an UML Class Diagram, we first identify all entities that might be present in one model. We view these diagrams as a collection of nodes (model classes) connected by means of links representing the rela-

tional aspects (generalizations, associations, compositions or aggregations). This is an interpretation of the detailed UML definition provided in **[12]**.

UML Classes (or Interfaces) are mapped to MSF nodes and UML Generalizations or Associations to MSF model links (Aggregations and Compositions are considered as special Associations). These MSF entities contain one expression-tree for which structure follows the UML 1.3 notation guide.

```
node (CLASS) is
   attribute_list
      attribute expression : THE_CLASS;
   end;
   connectability_list
      with GENERALISATION
        direction non_oriented,
        maximum none;
      with ASSOCIATION
        direction non_oriented,
        maximum none;
   end;
end CLASS;
```

Figure 5 : *The MSF definition of a Class*

Figure 5 shows the MSF specification of an UML class. All classes components are described in the expression-tree THE_CLASS attribute. The MSF description also provides connectability rules with an arbitrary number of generalization or association links (direction and maximum of connections).

Figure 6 shows the structure of the THE_CLASS expression-tree. This is an internal standard defined when designing the MSF description of UML and thus, all class node instances in a model have to respect this structure.

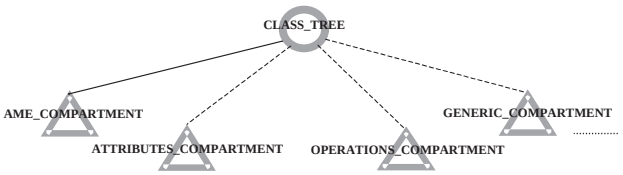

Figure 6 : *The Class syntax tree structure in MSM.*

In the graphical notation used in Figure 6, imposed parts are connected using a continuous line while optional parts are connected using a dotted line. Sub-trees are marked as triangles and nodes as circles. Tree nodes can hold: a construction labels (marked with capital letters), a string (prefixed with $) and/or an integer (prefixed by #). Alternative sub-trees are separated by the word "or".

As in UML, the structure of the THE_CLASS expression-tree is divided in compartments. The "Name Compartment" has to be defined. Figure 7 shows the structure of the "Name Compartment" expression tree (referenced in the class syntax-tree of Figure 6).

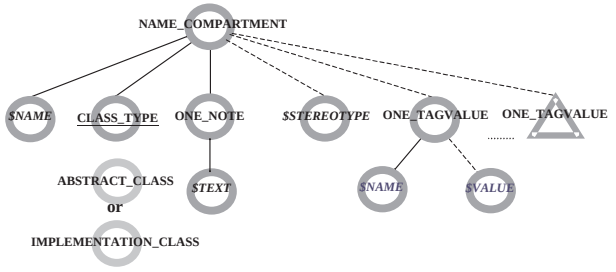

Figure 7 : *The "Name compartment" structure in MSM.*

Both "Attributes Compartment" and "Operations Compartment" hold the class attributes and operations. The "Generic compartments" provides an extension mechanism suitable for extending UML classes (for example, the description of exceptions that may be raised by the class).

The MSM representation of our example (Figure 4) contains 7 entities: 4 nodes corresponding to the 4 UML classes and 3 links corresponding to the UML Generalization, the UML qualified Association and the UML Composition. (see Figure 8).

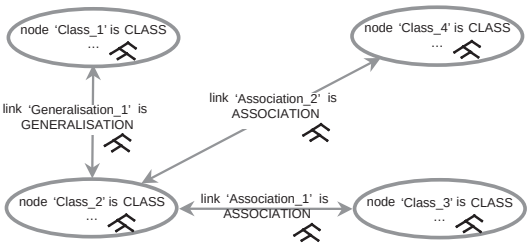

Figure 8 : *A graphical representation of the MSM entities*

Figure 9 presents the MSM description of the "Child_Class" in Figure 4.

```
node 'Class_2' is CLASS where (
attribute THE_CLASS =>
  sy_node (CLASS_TREE:
    sy_node (NAME_COMPARTMENT:
      sy_node (NAME:
        sy_leaf ('Child_Class')
      ),
      sy_leaf (IMPLEMENTATION_CLASS),
      sy_node (ONE_NOTE :
        sy_leaf ('A NOTE example.')
       )
    ),
    sy_node (ATTRIBUTES_COMPARTMENT:
    ....
  );
```

Figure 9 : THE_CLASS *attribute of node "Child_Class"*

The definition of Generalisation expression-trees and association expression-trees follows the same strategy.

4.2. Building the semantic pattern

Semantic expression-trees are an abstract internal representation of the output formalisms. Therefore, they are built according to its semantic. When building the semantic pattern, we must take care about the semantic expressions found in the output formalism and about what is the input formalism information used to build these expressions.

Because we use several output formalisms (here, Ada95 and C++) the semantic expression trees produced by the semantic pattern have to be compatible with both semantics. This remains possible because, despite their difference, they belong to the same formalism class (here, OO languages).

In the next part of the section, we first present the structure of a semantic-tree associated with an UML class and then we will explain the reasons that lead us to such a representation.

The semantic expression-tree structure provided in Figure 10 corresponds to the description of a class. Nodes are marked as an ellipse and sub-trees as triangle. Imposed sub-trees are linked with continuous lines while optional are with dotted lines. Nodes may contain one or more of the following elements:

- a semantic constructor (marked in capital letters). When

underlined, it contains one out of several values (e.g. in Figure 10, the TYPE constructor can have one of the following values, INPLEMENTATION_CLASS or ABSTRACT_CLASS),
- a string value (marked "string"),
- an integer value (marked "int").

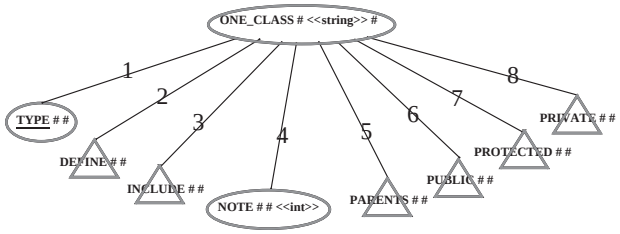

Figure 10 : *The structure of the "Class" semantic expression tree.*

According to Table 2 packages are the Ada95 equivalents of C++ compilation units (classes). For that reason, the constructor located in root of the semantic-expression tree of Figure 10 identifies an UML class instead of a language-based one:
- 1-The first sub-node specifies the class type (ABSTRACT or IMPLEMENTATION).
- 2-The second sub-node sets constants and elements that are useful to establish the C++ inclusion system, mainly the "#define" directive. Such information is ignored for a language like Ada95, which provide a better mechanism than the preprocessor-based one.
- 3-The third sub-node sets the relation with other units. In C++, this will correspond to the " #include " preprocessor directives. In Ada95 it will be translated into " with " and " use " clauses.
- 4-The note sub-tree contains class specific comments.
- 5-The fifth sub-tree contains the list of all inherited parent classes.
- 6,7,8-The three last sub-trees (PUBLIC, PROTECTED and PRIVATE) have the same structure defining a list of attributes, operation and related association ends for the corresponding visibility.

```
semantic_rule PRODUCE_ALL (none) return void is
  SEM_TREE   : semantic_tree;
  CLASS_NAME : string;
  NB_CLASSES : integer;
begin
  NB_CLASSES := nb_node_instance (CLASS);
  for I in 1 .. nb_node_instance (CLASS) do
   CLASS_NAME := sm_rule GET_CLASS_NAME
      (REF => get_node_reference (CLASS,
                                  $int (INDEX)));
   SEM_TREE := sm_rule WORK_ON_A_CLASS
      (REF => get_node_reference (CLASS,
                                  $int (INDEX)),
       CLASS_NAME => $str (CLASS_NAME));
   generate $smt (SEM_TREE) in
                     {$str (CLASS_NAME) & '.h'};
  end for;
  return;
end;
```

Figure 11 : *The main MSSM semantic rule*

Figure 11 shows one of the main semantic rules. It is the entry point of our transformation engine. It processes all MSM nodes having CLASS type and performs the following actions:
- extraction of the class name,
- construction of an expression-tree describing the class implementation in OO languages (this is done in the rule WORK_ON_A_CLASS),
- application of the syntactic pattern to this semantic-tree. The result is written in a file named after the class name and its format depends on the associated syntactic pattern. Here, we build two transformation engines, one for C++, the other one for Ada95 that will reuse the same semantic pattern. As mentioned in Section 2.2., the same semantic pattern could be reused for other OO languages such as Java of Effeil.

With the experience, the definition of a semantic pattern usually follows these steps:
- identification of the pertinent semantic expression-trees structure,
- identification of the relevant semantic constructors in these expression-trees,
- definition of the semantic rules building these semantic expression-trees.

4.3. Building C++ and Ada95 syntactic patterns

The syntactic pattern is the element that dresses semantic expression-trees to produce the appropriate output. Theoretically, it is the only part to be rewritten when changing the output formalism.

```
#ifndef Child_Class_h
#define Child_Class_h 1

#include "Parent_Class.h"
#include "Component_Class.h"
#include "Associated_Class.h"

class Child_Class : public Parent_Class {...}
```

Figure 12 : *The C++ structure of "Child_Class".*

It is connected to the semantic pattern by means of the semantic constructors described. Each one corresponds to a syntactic rule. Extra rules (that do not correspond to a semantic constructor) may be defined for convenience. However there must be one rule per declared semantic constructor.

```
with Ada.Finalization; use Ada.Finalization;
with Parent_Class_Pkg; use Parent_Class_Pkg;
with Component_Class_Pkg; use ...
with Associated_Class_Pkg; use ...

package Child_Class_Pkg is
 type Child_Class is new
       Parent_Class with private;
  ...
end Child_Class_Pkg;
package body Child_Class_Pkg is
  ...
end Child_Class_Pkg;
```

Figure 13 : *The Ada95 structure of "Child_Class"*

Figure 12 and Figure 13 respectively provide an example of C++ and Ada95 structures corresponding to "Child_Class" in the example from Figure 4. By shake of place, we cannot provide bigger examples.

Figure 14 shows the Ada95 syntactic rule associated with the ONE_CLASS semantic constructor (root of the semantic-tree presented in Figure 10). It refers to several other rules dedicated to the generation of an element of the class. $N represents the N^{th} son in the implicit expression-tree provided as a parameter (it will then be the implicit parameter of the invoked rule). $0 represents the complete expression-tree itself.

```
syntactic_rule ONE_CLASS is
begin
  put ('with Ada.Finalization;');
  put_line ('use Ada.Finalization;');
  apply WHITH_USE ($2);
  apply NOTE ($4);
  apply PACKAGE_DECLARATION ($0);
  apply PACKAGE_BODY ($0);
end;
```

Figure 14 : *the ONE_CLASS syntactic rule.*

Figure 15 provides another example of syntactic rule: the one associated to the WITH_USE constructor (root of the PARENT sub-tree in Figure 10). It illustrates the functional-like use of recursivity to go through all the expression-tree sons. Here, $N* corresponds to the initial expression tree from which the N-1 first sons have been deleted.

```
syntactic_rule WITH_USE is
begin
  if $# > 0 then
     apply ONE_WITH_USE ($1);
     if $# > 1 then
       apply WITH_USE ($2*);
     end if;
  end if;
end;
```

Figure 15 : *The "WITH_USE" syntactic rule.*

4.4. Metrics

The main goal of prototyping is the fast implementation of a tool. MetaScribe clearly meets this goal. We have been able to perform a fast implementation of three code generators from UML class diagrams to respectively C++, Ada95 and Java. The last one was developed between the paper acceptance and the final submission. Therefore, it is not presented in this paper.

The elaboration of a MSF description of UML took about four days. Building the semantic pattern took about one week and the construction of the C++ syntactic pattern about three days. The elaboration of the UML-class diagram to C++ took about two weeks. The develommement of the syntactic pattern for ADA95 cost only three days. The construction of the Java syntactic pattern was done by reusing the C++ version and was performed in two days.

5. Conclusion

We have described MetaScribe, a tool for the prototyping of transformation engine. A transformation engine is a program transforming an input description to an output one. Code generators, more and more used in CASE environment to develop programs from high level specification, are examples of transformation engines.

To illustrate the use of MetaScribe and quantify the advantages of using it, we have developed a case study: program generation from an UML class diagram in C++, Ada95 and Java. This study was accomplished quite easily in a reasonable time.

Compared to a "traditional hand made" approach, it appears that MetaScribe provides several advantages:

- Implementation of a transformation engine is at least not longer.
- MetaScribe patterns provides a scheme helpful to the designer of transformation engine.
- Reuse capabilities of MetaScribe allow to build easily and at low cost new transformation engines based on existing ones. Modifications mainly reside in the specification of new syntactic rules for a new output and save development time.
- MetaScribe provides tracking/debug facilities available for both semantic and syntactic rules. Such mechanisms, not presented by lack of place are useful to build and maintain patterns.

We most probably have done the work better than with a traditional approach combining the use of a lexical analyzer and parser generator such as lex and yacc.

It is more difficult to compare benefits of MetaScribe compared to the XML environment. XML is now becoming a standard for data exchange, even if it lacks in the potential of manipulating the semantics of the input formalism to produce some output. It definitely has a good potential but we have to wait for a final release of elements such as XSLT to really decide weather it has capabilities similar to MetaScribe ones.

The use XML as an input to MetaScribe appears to be a natural extension. We plane to study the implementation of such a capability. Then, MetaScribe transformation engines could be able to use either MSM/MSF or XML document/DTD as input data.

6. References

[1] Ada 9X Mapping/Revision Team, "Ada 95 Reference Manual and Rationale, ANSI/ISO/IEC-8652", Intermetrics, Inc. 733 Concord Avenue Cambridge, Massachusetts 02138 1995

[2] J. Barnes, "Programming in Ada95", pp 449-453, Adison-Wesley, 1996, ISBN0-201-87700-7

[3] B. Brosgol, "A Comparison of the Object-Oriented Features of Ada 95 and Java™", White paper avilable on http://www.aonix.com/Pdfs/CSDS/bmbta97.pdf, May 1999

[4] P. Desfray, "Object Engineering: The Fourth Dimension", Addison-Wesley, 1994

[5] P. Desfray, "Automation of design pattern : concepts, tools and practices", UML'98 International Workshop, June 1998

[6] HOOD Technical Group, "HOOD Reference Manual, release 4", June 1995

[7] ISO/IEC, "Ada Semantic Interface Specification (ASIS)", ISO/IEC 15291, 1999

[8] S Johnson, "Ada-95: A guide for C and C++ programmers" , 1995, <http://www.adahome.com/Ammo/cpp2ada.html>

[9] F.Kordon, "MetaScribe, an Ada-based Tool for the Construction of Tranformation Engines", in proceedings of the International Conference on Reliable Software Technologies - Ada-Europe'99, LNCS- vol 1622, pp 308-319, Santander, Spain, June 7-11, 1999

[10] OMG, "Extensible Markup Language (XML) 1.0", W3C recomendation, february 1998

[11] OMG, "Meta Object Facility (MOF) Specification, v 1.3", OMG documentation, 1999

[12] OMG, "OMG Unified Modeling Language Specifiacation", version 1.3, June 1999, <http://www.omg.org/cgi-bin/doc?ad/99-06-09.zip>

[13] J. Rumbaugh, I. Jacobson, G. Booch , "The Unified Modeling Language Reference Manual", Addison-Wesley (Object Technology Series), 1998

[14] Rational Software Corporation, Rational Rose 98 Help Chapter 10 "Rose C++", 1998, <http://www.rational.com/rose/>

[15] S. Si Ahir, "From the Unified Modeling Language (UML) to C++", The Visual C++ Developers Jurnal, 1999, <http://www.vcdj.com>

[16] SofTeam, "Objecteering 4.3 user manual", SofTeam company, 1999

[17] W3C, "Extensible Stylesheet Language (XSL) Specification", http://www.w3.org/TR/1999/WD-xsl-19990421, April 1999

[18] W3C, "XSL Transformations (XSLT) Version 1.0", http://www.w3.org/TR/1999/REC-xslt-19991116, November 1999