MetaScribe, an Ada-based Tool for the Construction of Tranformation Engines

Fabrice Kordon

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

Abstract : This paper presents MetaScribe, a generator of transformation engine designed to help the implementation of program generators or transformation of a specification to another one. MetaScribe defines a meta-data description scheme suitable for the internal representation of various graphical and hierarchical description.
MetaScribe is fully implemented in Ada and uses the language facilities to enforce type checking and handling of errors in the manipulated descriptions.

Keywords : Meta-data description, Semantic transformation, Code generation.

1 Introduction

Software engineering methodologies rely on various and complex graphical representations such as OMT, UML, etc. They are more useful when associated to CASE (Computer Aided Software Engineering) tools designed to take care of constraints that have to be respected. Such tools help engineers and facilitate the promotion of such methodologies.

Now, CASE tools gave way to CASE environments that may be adapted to a specific understanding of a design methodology. A CASE environment can be defined as follow **[12]** : it is a set of cooperative tools. CASE environments are built on a platform that allows tool plugging. Communication and cooperation between tools must subsequently be investigated.

Implementation of CASE environments is a complex task because they need various functions like graphical user interface, database and communication facilities. Experimentation over large projects has outlined the difficulty to maintain them, especially when tools come from various origins. In a project like Ptolemy **[10]** , the software bases for the project have largely changed in order to ease maintenance as well as new development. Such work (in particular, the Tycho interface system **[6]**) takes into account the definition of evolutionary interfaces between major components.

To implement methodologies and experiment them on large case studies, we have elaborated FrameKit **[7]** , a framework for the quick implementation of CASE environments. It is parameterized in order to provide a framework for the customization of CASE environments dedicated to a given method (Fig. 1). FrameKit is mostly integrated in Ada (a small amount of C is used to interface Unix) and

provides enhanced Ada Application Program Interfaces (API) to operate this customization procedure.

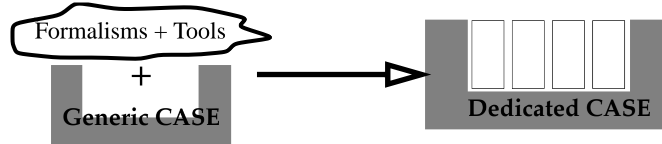

Fig. 1 From a Generic CASE to a dedicated one.

We have used FrameKit to implement CPN-AMI2, a Petri net based CASE environment with satisfactory results. To do so, we have described the Petri net formalism and declared a set of coherent tools that can be applied on Petri net specifications **[8]** . Doing so, we have noticed there is a major problem to provide data sharing facilities between tools coming from various origins. This problem is even more difficult for an environment like FrameKit where we cannot use specificity of a formalism to ease the implementation (FrameKit is supposed to be generic). There is a need to provide FrameKit with facilities to easily manage *secondary standards*. Secondary standards in FrameKit are communication-based data representations dedicated to a formalism (for example the description of Petri net results potentially exploited by other tools).

CPN-AMI2 implements the MARS method **[3]** in which both transformations (from a representation to another one) and program generators have to be implemented. We noticed there were a lack of tools to ease such implementations. Parser generators are useful to help the syntactical analysis of input specification but the transformation rules have to be described using a programming language. It is then difficult to reuse transformation rules. On another hand, Expert systems provide good facilities for the definition of transformation rules but are poor for I/O and efficient management of complex data structures. So, there is also a need for a tool providing facilities to the implementation of *transformation engines*. A transformation engine is a program that parses an input specification and produces an output specification by applying transformation rules.

This paper presents MetaScribe, a tool suitable for the two purposes outlined above. MetaScribe is fully implemented in Ada and uses the language facilities to enforce type checking and handling of errors. Section 2 presents MetaScribe and Section 3 discuss its implementation.

2 MetaScribe

MetaScribe solves problems similar to the ones identified in hardware/software codesing of embedded systems where specific processors are built in very limited series. There is a need for designing specific compilers at low-cost. Retargetable compilers are designed for this task and can be classified as follow **[1]** :

- Automatically retargetable compilers: they contain a set of switches that need to

be set to specify the target architecture. Essentially, all possible target architectures that the compiler is intended to be used for are already built in;

- User retargetable compilers: the user specifies the target architecture to the compiler-compiler in some form. Typically, this is a representation of the instructions in terms of some primitive operations. The compiler-compiler takes this as input and generates a compiler for the specified architecture;
- Developer retargetable compilers that is a way to handle machine specific optimizations that go beyond instruction selection is to permit the developer to modify the compiler to target the given architecture. The difference between retargeting and writing a new compiler for any architecture is rather low. For a compiler to be considered retargetable in this scenario, no new processor dependent optimization capabilities are added to the compiler during retargeting.

MetaScribe fits the needs outlined by the two last points. Like a parser generator such as flex/bison **[5, 9]** , it enables the use of rules applied on input specifications according to a customized scheme. However, because it focuses on the management of hierarchical and graphical like specifications, its philosophy is quite different from the one of a parser generator and the transformation scheme is made of rules contained in a semantic pattern. Like a retargetable compiler, it enables the application of a customized output according to a given syntax format. It is then possible to associate discrete syntactic patterns to a given semantic pattern (e.g. apply Ada or Java code on an object program description). Its parameterization procedure is then similar to the concept of hypergenericity **[2]** .

2.1 Architecture of MetaScribe

In order to increase reusability of transformation engine's elements, input description, as well as semantical and syntactical aspects of a transformation are separately defined. To be operated, MetaScribe requires three elements to be described (Fig. 2):

- Formalism definition: it is expressed using the MSF meta-description language. Users have to declare any entity that can be found in the formalism;
- Semantic pattern: it is expressed using the MSSM language. Users define the transformation rules to be applied on the associated formalism;
- Syntactic rules: it is expressed using the MSST language. Users define the syntactic representation associated to constructors declared in the corresponding semantic pattern.

The semantic pattern is composed of rules that produce a polymorphic semantic representation similar to the one of ASIS **[11]** called *semantic expression-trees*. Semantic expression-trees are expression trees expressing the semantic of a description without syntactic information (like a parsed program in a compiler).

Semantic patterns are not dedicated to a given programming language and can be customized. They declare a list constructors that make the connection with a syntactic pattern describing how these constructors will be represented.

Input specifications must be written using the MSM data description language automatically customized according to the entities declared in the MSF description. Checks are enforced at execution time of the transformation engine.

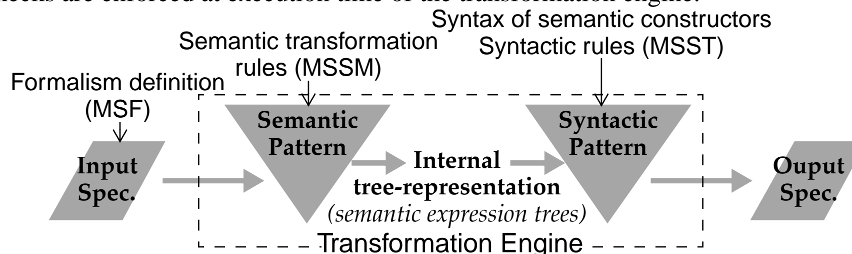

Fig. 2 Structure and components of a transformation engine generated with MetaScribe.

MetaScribe generates a transformation engine from a triplet <*input formalism, semantic pattern, syntactic pattern*>. Such a mechanism enables the reuse of components in any of the involved elements. Thus, MSF, MSSM and MSST languages allow separate definition of pattern components

2.2 Formalism description

The MSF meta-description language allows users to declare entities potentially found in a type of specification. MSF (MetaScribe Formalism) is an object based description language where classes are either nodes or links. A node is a piece of specification. A link relates some nodes together. Both nodes and links may carry local information stored in attributes. Global information is specified in attributes that are related to the specification (and not to a specific entity). Attributes contain typed expressions (characters, strings, integers, etc.).

```
formalism ('NETWORK_DSC');
// nodes and linkss
entity_list
   COMPUTER : node,
   HUB_16 : node,
   CABLE : link;
// Global attributes
global_attributes
   attribute string  : NET_ID;
end;
// parsed expressions for IP-address description
construction_list (DYNAMIC, STATIC);
// description of nodes and linkss
node (COMPUTER) is
   attribute_list
      attribute string  : NAME;
      attribute expression : IP_ADDRESS;
   end ;
   connectability_list
      with CABLE
         direction in ,
         maximum 1 ;
   end ;
end COMPUTER ;

node (HUB_16) is
   attribute_list
      attribute string  : NAME;
      attribute expression : IP_ADDRESS;
   end ;
   connectability_list
      with CABLE
         direction out ,
         maximum 16 ;
       with CABLE
         direction in ,
         maximum 1 ;
   end ;
end HUB_16 ;

link (CABLE) is
   attribute_list
      none
   end ;
end CABLE ;
```

Fig. 3 Example of formalism description with MSF.

Let us illustrate MSF possibilities with a small example. Fig. 3 presents a simple formalism description: a network composed of computers and hubs related by means

of communication cables. Both computers and hubs are reference using a name and an IP-number. The name attribute is a string and the IP-number attribute is an expression tree composed of a root (tagged `STATIC` or `DYNAMIC`) and, if `STATIC`, four sons representing the four parts of an IP-address. Nodes define connectability rules by accepting to be related to some links. Here, a `HUB_16` node class may accepts 16 output connections via a `CABLE` and only one input connection with a `CABLE`.

MetaScribe uses the MSF description to customize the description of specification using this formalism. This goal is achieved by the MSM data-description language.

Let us consider a network description expressed using the MSF description of Fig. 3 (Fig. 4). The network is composed with two computers connected to a hub via two cables. The hub and one computer have static IP-addresses and the second computer has a dynamic one.

```
formalism ( 'NETWORK_DSC' ) ;
// Global attributes
where (attribute NET_ID => 'my_net') ;
// hub list
node 'hub_1' is HUB_16
  where (attribute NAME => 'hub_one',
         attribute IP_ADDRESS => sy_node (STATIC:
                                    sy_leaf (10),
                                    sy_leaf (10),
                                    sy_leaf (10),
                                    sy_leaf (10)));
// host list
node 'host_1' is COMPUTER
  where (attribute NAME => 'host_one',
         attribute IP_ADDRESS => sy_leaf (DYNAMIC)) ;
node 'host_2' is COMPUTER
  where (attribute NAME => 'host_two',
         attribute IP_ADDRESS => sy_node (STATIC:
                                    sy_leaf (10),
                                    sy_leaf (10),
                                    sy_leaf (10),
                                    sy_leaf (11)));
// connections
link 'cable_1' is CABLE
  where (none)
  relate HUB_16:'hub_1' to COMPUTER:'host_1';
link 'cable_2' is CABLE
  where (none)
  relate HUB_16:'hub_1' to COMPUTER:'host_2';
```

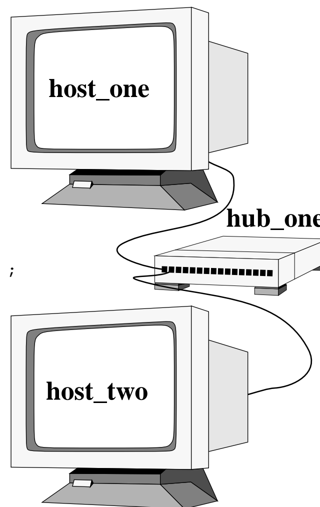

Fig. 4 Description of a specification using teh formalism defined in Fig. 3.

The major advantage of MSM (MetaScribe Model) is to clearly describe a data-structure that can have a memory equivalent. It is a polymorphic data-description language because it does not take side on any aspect of the specification and only carries out its description.

A transformation engine generated with MetaScribe first parses the MSM description of a model and maps it to data structures in memory. Then, actions defined in both semantic and syntactic patterns are applied on this memory representation.

2.3 Semantic patterns description

The MSSM description language allows users to define transformations to be applied on an input specification. Such a description is related to a formalism (i.e. a MSF

description) where entities to be manipulated are described. A MSSM (MetaScribe SeMantic) descriptions is composed with three elements:

- *Constructors* that are links to a given syntactic pattern,
- *Rules* that process entities found in the input specification (nodes, links, attributes),
- *Static trees* that corresponds to constant semantic expression-trees.

MSSM is a functional based language. Program units are rules (actions to be applied on the memory image of an input specification) and static-trees. It is possible to build a semantic pattern from separate files.

The goal of a semantic pattern is to produce semantic expression trees. Semantic expression trees are trees where nodes contains at least one of the three following fields:

- A constructor,
- A string,
- An integer.

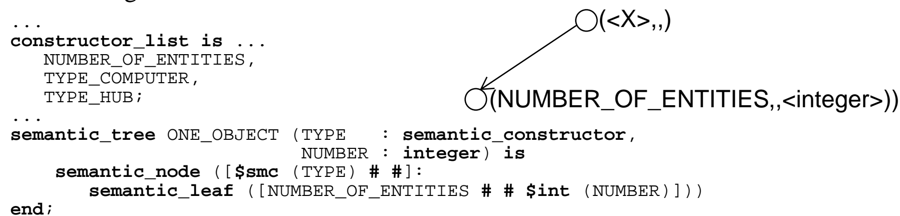

Fig. 5 Example of a static-tree in a semantic pattern.

Fig. 5 presents an example of semantic expression tree and its corresponding description as a static-tree. Let us assume that, to produce a textual display of a network description, three semantic constructors have been declared: NUMBER_OF_ENTITIES, TYPE_COMPUTER and TYPE_HUB. The semantic-tree ONE_OBJECT corresponds to the tree on the upper right where <*X*> is a predefined constructor (either TYPE_COMPUTER or TYPE_HUB) and <*int*> a natural value. Thus, the definition of a syntactic form for such an expression can be easily performed. Here, the tree's root only contains a predefined constructor and its son a predefined constructor and an integer.

Fig. 6 presents an example of semantic rule and shows how semantic and syntactic patterns are connected. The rule produces an expression-tree that references all computers in a network. First, it creates the root of a result expression-tree and applies the rule A_RULE to any COMPUTER node in the description. Then, it invokes the syntactic pattern and applies it to the resulted tree. Note that the syntactic pattern is not explicitly named (the association <*MSF description, MSSM description, MSST description*> is set by users when they invoke MetaScribe to build a transformation engine). The application of the syntactic pattern is written in the file a_file as asked in the generate directive.

```
semantic_rule LIST_COMPUTERS (none) return void is

TREE : semantic_tree;

begin
   // Create the root of the result expression-tree
   TREE := create_sm_tree ([ANALYSIS_RESULT #
                                $atrv_str (attribute NET_ID)#]);
   // linking a computer to the description
   message ('Analysis the network...');
   if nb_node_instance (COMPUTER) > 0 then
      for COMP in 1 .. nb_node_instance (COMPUTER) do
         TREE := add_sm_son ($smt (TREE),
                            sm_rule A_RULE (get_node_reference (COMPUTER,
                                                          $int (COMP))));
      end for;
   end if;
   // Applying the syntactic pattern to the result expresion-tree
   generate $smt (TREE) in 'a_file';
   message ('Done...');
   return;
end;
```

Fig. 6 Example of a rule in a semantic pattern.

2.4 Syntactic patterns description

The MSST description language allows users to associate a syntactic expression to any constructor declared in the semantic pattern. MSST (MetaScribe SynTactic) is a functional language composed with two types of rules:

- *External rules* are associated to predefined constructors. Such rules can be either implicitly invoked according to the constructor tag of a semantic expression-tree node or explicitly invoked;
- *Internal rules* are not associated to predefined constructors. Thus, they can only be explicitly invoked from external rules. Usually, an external rule is a "front end" for several internal rules.

```
syntactic_rule TYPE_COMPUTER is
begin
   put ('number of computers in the network :');
   apply ($1);
end;

syntactic_rule TYPE_HUB is
begin
   put ('number of hubs in the network      :');
   apply ($1);
end;

syntactic_rule NUMBER_OF_ENTITIES is
begin
   put_line ($str_int (0));
end;
```

Fig. 7 Abstract of a syntactic pattern.

Fig. 7 contains a set of external rules dedicated to textual display of the semantic tree defined in Fig. 5. Any declared constructor has to be related to a syntactic rule in the pattern. `TYPE_COMPUTER` and `TYPE_HUB` implicitly refer to rule `NUMBER_OF_ENTITIES` using the apply instruction. It invokes the rule associated to the predefined constructor found in the root of the tree transmitted as a parameter.

Fig. 8 presents the execution scheme of the rules presented in Fig. 7 on a tree that respect the required format. `TYPE_NUM` implicitly invokes the rule associated to the semantic constructor located in the first son of the semantic expression tree transmitted as a parameter (here, `NUMBER_OF_ENTITIES`). This invoked rule converts the integer value into a string that is written in the output declared in the corresponding semantic pattern (via the generate directive). The result for this expression-tree should be:

```
number of hub in the network       : 4
```

Of course, the «apply» directive can be used to explicitly apply a syntactic rule to a semantic-expression-tree. Then, the rule identifier is also transmitted.

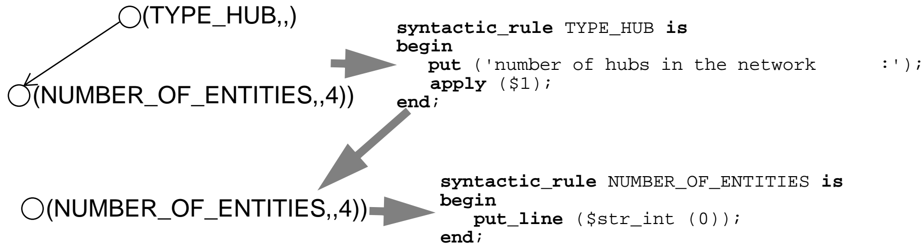

Fig. 8 Execution of the syntactic pattern on a given expression-tree.

3 About MetaScribe implementation

Implementation's principles of MetaScribe are rather simple: it is a program generator. We first investigated an interpreted approach but it raises two problems:

- it should be slower,
- execution time checks have to be implemented.

Thus, transformation engines produced by MetaScribe are specific programs implementing a particular transformation. Then, the choice of Ada is obvious: it provides good mechanisms for execution time type checking. Then, all execution time checks are supported by the Ada runtime. Exceptions are caught in a handler and the propagation mechanism is used to provide the program stack (soon or later, this function should be supported by all compilers).

3.1 Structure of a generated transformation engine

The structure of a transformation engine generated by MetaSCribe is shown in Fig. 9. There are six components (the Ada runtime is not a part of MetaScribe):

- the MetaScribe runtime that performs input/output operations and defines all data structures suitable for a transformation engine. For example, it contains a generic semantic expression-tree manager to be instantiated for the constructors defined in the semantic pattern;
- a MSM parser that interpret the input specification and built a memory representation on which semantic rules will operate;

- a MSF parser to dynamically interpret the input formalism's description;
- the implementation of the semantic pattern;
- the implementation of the syntactic pattern;
- the MetaScribe starter.

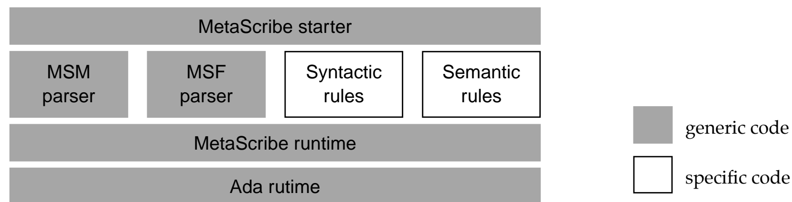

Fig. 9 Structure of a tranformation engine produced by MetaScribe.

The MetaScribe starter basically performs the following operations: a) parsing the MSM input description (with dynamic checks of the input specification according to the MSF description[1]), b) «launching» of the semantic pattern. «launching» means that MetaScribe systematically applies some rules to the current description. Such rules are called «main rules», of course, there must be at least one main rule in a semantic pattern.

Transformation engines designed with MetaScribe are ready to be plugged in FrameKit, our generic CASE environment. Thus, it is a simple way to prototype transformation tools as well as program generators and experiment them in a given methodology.

3.2 Implementation of a semantic pattern

The semantic pattern is composed with two packages : the first one defines data structures, the second one implements the semantic pattern.

```
with GEN_RTM_SEM_TREE;
package STATISTICS_SEM_TREE_MNGR is
    -- ====================================================================
    -- The type that contains all the predefined semantic constructors
    type STATISTICS_SEMANTIC_CONSTRUCTOR is (XXX_SEM_TREE_NO_ITEM,
                                             STATISTICS_RID_ANALYSIS_RESULT,
                                             STATISTICS_RID_NUMBER_OF_ENTITIES,
                                             STATISTICS_RID_TYPE_COMPUTER,
                                             STATISTICS_RID_TYPE_HUB);
    -- ====================================================================
    -- Instanciation of the generic unit that defines semantic expresion-trees
    package INTERNAL_STATISTICS_SEM_TREE_MNGR is
            new GEN_RTM_SEM_TREE (STATISTICS_SEMANTIC_CONSTRUCTOR);
    -- ====================================================================
    -- The exception used to generate the propagation stack of errors
    PROPAGATE_ERROR : exception;
end STATISTICS_SEM_TREE_MNGR;
```

Fig. 10 Example of generated code: the main data structure to operate the semantic.

Fig. 10 provides the specification of the package defining data structures. It declares an enumerative type containing all constructors (STATISTICS_SEMANTIC_CONSTRUCTOR)

1. To speed up execution, it is possible to disable these execution time check.

and uses it to instance the semantic expression-tree manager. This enables the use of primitives dedicated to the handling of semantic tree-expressions for a specific transformation engine.

The package implementing the semantic pattern groups procedures and functions corresponding to rules and static semantic-trees. The body of these primitives is located in a separate file in order to avoid big files. Fig. 11 shows the code generated for the static semantic-tree presented in Fig. 5. primitives of the instantiated semantic expression-tree manager are used to build a specific tree. Exception handlers are generated to either signal a problem or propagate it. In debug mode, traces are automatically generated at both the beginning and the end of the subprogram.

```
separate (STATISTICS_SEM_ENTITIES)

function SMT_STATISTICS_ONE_OBJECT (TYPE : in STATISTICS_SEMANTIC_CONSTRUCTOR;
                                    NUMBER : in INTEGER) return RTM_SEM_TREE is
   TMP_TAB : SUB_SEM_TREE_STORAGE;
begin
   TMP_TAB := CREATE_RTM_SEM_TREE (CREATE_RTM_SEM_NODE (ITS_TYPE => TYPE,
                                                        ITS_INT_VALUE => NUMBER));
   TMP_TAB := ADD_SON_TO_CURRENT_NODE (TMP_TAB, CREATE_RTM_SEM_TREE (
              CREATE_RTM_ST_NODE (ITS_TYPE => STATISTICS_RID_NUMBER_OF_ENTITIES)));
   return TMP_TAB;
exception
   when PROPAGATE_ERROR =>
      FK_PUT_MSG (MESSAGE => "propagated in ->SMT_STATISTICS_ONE_OBJECT");
      raise ;
   when others  =>
      FK_PUT_MSG (MESSAGE => "Huge problem in ->SMT_STATISTICS_ONE_OBJECT");
      raise PROPAGATE_ERROR;
end SMT_STATISTICS_ONE_OBJECT;
```

Fig. 11 Example of generated code: static semantic-tree ONE_OBJECT (Fig. 5).

3.3 Implementation of a syntactic pattern

The syntactic pattern is composed of a package that groups all syntactic rules. As for semantic patterns code generation and to avoid large files, rules bodies are located in separate units. Because they never return any value (they are used to print strings in an output), syntactic rules are implemented via procedures.

```
with STATISTICS_SEM_ENTITIES,
     SYN_GENERATOR_RUNTIME;
use  STATISTICS_SEM_ENTITIES,
     SYN_GENERATOR_RUNTIME;

separate (SYNT_PATTERN_TEXT_DISPLAY_FOR_STATISTICS)
procedure SYR_STATISTICS_TEXT_DISPLAY_TYPE_COMPUTER (SEM_TREE : in RTM_SEM_TREE) is
begin
   SYN_RTM_PUT (FILE_IN_MEM, TO_VSTRING ("number of computers in the network :"));
   APPLY_SYNTACTIC_RULE (CURRENT_GOTO_SON (SEM_TREE, 1));
exception
   when PROPAGATE_ERROR =>
      FK_PUT_MSG (MESSAGE =>
                  "propagated in ->SYR_STATISTICS_TEXT_DISPLAY_TYPE_COMPUTER");
      raise ;
   when others  =>
      FK_PUT_MSG (MESSAGE =>
                  "Huge problem in ->SYR_STATISTICS_TEXT_DISPLAY_TYPE_COMPUTER");
      raise PROPAGATE_ERROR;
end SYR_STATISTICS_TEXT_DISPLAY_TYPE_COMPUTER;
```

Fig. 12 Example of generated code: syntactic rule for constructor TYPE_COMPUTER (Fig. 7).

Fig. 12 shows the code generated for syntactic rule TYPE_COMPUTER in the syntactic

pattern (Fig. 7). A parameter is necessary that was implicit in the syntactic pattern description: the semantic expression-tree on which the rule is applied. Because rules are sequentially invoked, the output file is not provided, a «current output» is set each time the syntactic pattern in invoked by a semantic rule.

Display functions that put message to a default output set via the generate directive belongs to the MetaScribe runtime (like SYN_RTM_PUT which behaves like PUT). navigation directives (like $1 in the syntactic rule) are implemented by the instantiated semantic expression-tree manager (for example, CURRENT_GOTO_SON implements the $<int> directive).

Exception handlers are set to detect problems in expression-trees construction (they correspond to bugs in the semantic pattern description). A trace mode allows tracking of all input semantic expression-trees of syntactic rules.

```
-- The function that perform dynamicaly the application of the rule tag referenced in the current
-- node of a semantic tree.
procedure APPLY_SYNTACTIC_RULE (SEM_TREE : in RTM_SEM_TREE) is
begin
   case GET_RTM_ST_NODE_TYPE (CURRENT_CONTENT (SEM_TREE)) is
      when STATISTICS_RID_ANALYSIS_RESULT =>
         SYR_STATISTICS_TEXT_DISPLAY_ANALYSIS_RESULT (SEM_TREE);
      when STATISTICS_RID_NUMBER_OF_ENTITIES =>
         SYR_STATISTICS_TEXT_DISPLAY_NUMBER_OF_ENTITIES (SEM_TREE);
      when STATISTICS_RID_TYPE_COMPUTER =>
         SYR_STATISTICS_TEXT_DISPLAY_TYPE_COMPUTER (SEM_TREE);
      when STATISTICS_RID_TYPE_HUB =>
         SYR_STATISTICS_TEXT_DISPLAY_TYPE_HUB (SEM_TREE);
      when XXX_SEM_TREE_NO_ITEM =>
         raise SYNT_PTRN_APPLY_ERROR;
   end case ;
exception
   when PROPAGATE_ERROR =>
      FK_PUT_MSG (MESSAGE => "propagated in ->APPLY_SYNTACTIC_RULE");
      raise ;
   when others  =>
      FK_PUT_MSG (MESSAGE => "Huge problem in ->APPLY_SYNTACTIC_RULE");
      raise PROPAGATE_ERROR;
end APPLY_SYNTACTIC_RULE;
```

Fig. 13 Example of generated code: automatic apply function.

One point is the implementation of the apply function. The explicit apply corresponds to a standard procedure call. The implicit apply is implemented by means the procedure shown in Fig. 13. It contains a case based on the enumerative type that describes all constructors declared in the semantic pattern (remind that internal rules cannot be implicitly invoked).

4 Conclusion

We have presented in this paper MetaScribe, a tool to quickly produce transformation engines. A transformation engine is a program that parses an input specification and produces an output specification by applying transformation rules. It can be used to build program generators or to ease standardization of data representation in a CASE environment like FrameKit.

MetaScribe is operational, information can be found on `<http://www-src.lip6.fr/metacribe>`. It has been experimented to generated Petri nets or Java programs from high-level semi-formal specification **[13]** . Results of these experimentations are satisfactory. We are currently using it to handle transformation from High Level Agent oriented specification into Petri nets in the ODAC project **[4]** .

References

[1] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sudarsanam, S. Tjiang & A. Wang, "Challenges in Code Generation for Embedded Processors", Chapter 3, pp. 48-64, in "Code Generation for Embedded Processors", P. Marwedel and G. Goossens editors, Kluwer Academic Publishers, ISBN 0-7923-9577-8, 1995

[2] P.Desfray, "Object Engineering, the fourth dimention", Addison-Wesley, 1994

[3] A. Diagne, P. Estraillier & F. Kordon, "Quality Management Issues along Life-cycle of Distributed Applications", in the proceedings of CARI'98, pp 753-763, Dakar, Sénégal, October 12-15, 1998

[4] A. Diagne & M.P. Gervais, "Building Telecommunications Services as Qualitative Multi-Agent Systems: the ODAC Project", in Proceedings of the IEEE Globecom'98, Sydney, Australia, November 1998

[5] C. Donnelly & R. Stallman, "Bison: The YACC-compatible Parser Generator", GNU documentation, <http://www.cl.cam.ac.uk/texinfodoc/bison_toc.html>, November 1995

[6] C. Hylands, E. Lee & H. Reekie, "The Tycho User Interface System", The 5th Annual Tcl/Tk Workshop '97, Boston, Massachusetts, pp 149-157, July 14-17, 1997

[7] F.Kordon & J-L. Mounier, "FrameKit, an Ada Framework for a Fast Implementation of CASE Environments", in proceedings of the ACM/SIGAda ASSET'98 symposium, pp 42-51, Monterey, USA, July 1998

[8] MARS-Team, "the CPN-AMI2 home page", <http://www.lip6.fr/cpn-ami>

[9] V. Paxson, "Flex: A fast scanner generator, Edition 2.5", GNU documentation, <http://www.cl.cam.ac.uk/texinfodoc/flex_toc.html>, March 1995

[10] Ptolemy Team, "The Ptolemy Kernel-- Supporting Heterogeneous Design", RASSP Digest Newsletter, vol. 2, no. 1, pp. 14-17, 1st Quarter, April, 1995

[11] S. Rybin, A. Strohmeier & E. Zueff, "ASIS for GNAT: Goals, Problems and Implementation Strategy", In M. Toussaint (Ed), Second International Eurospace - Ada-Europe Symposium Proceedings, LNCS no 1031, Springer Verlag, pp 139-151, 1995

[12] D.Schefström, "System Development Environments: Contemporary Concepts", in Tool Integration: environment and framework, Edited by D.Schefström & G. van den Broek, John Wiley & Sons, 1993

[13] P. Vidal, "Comparison between implementation and code generation for multi-agent systems : application to the Personnal Travel Assistant", Master thesis in an ERASMUS program, University of Olso and University P. & M. Curie, 1999