EUGENE: a STEP-based framework to build Application Generators

Alain Plantec, Vincent Ribaud

EUGENE: a STEP-based Framework to build Application Generators

A.Plantec and V.Ribaud
LIBr & Syseca

LIBr: Faculté des Sciences, BP 809, 29285 Brest Cedex, France
Syseca: 34 quai de la Douane, 29200 Brest, France

E-mail: {plantec,ribaud}@univ-brest.fr

April, 30, 1998

Abstract

Application generators translate specifications into products (programs, documentations). They parse specifications statements into data structures (called *dictionaries* or *meta-models*), from which desired products can be derived.

An application generator builder offers a way to define specification languages and associated parsers, to describe and traverse the meta-models structure and to specify the derivation on this structure. In most builders, derivation is specified with templates (or skeletons) of code. Templates contain a mixture of commands operating on the meta-models and "real" code directly put in the products.

STEP is an ISO 10303 standard developed to facilitate product information sharing by specifying sufficient semantic content for data and their usage. Parts of ISO 10303 are intended to standardize conceptual structures of informations either generic, or within a subject area (e.g. mechanics). Standardized parts are expressed with a dedicated technology, mainly an object oriented modelling language *EXPRESS* and a data access interface.

STEP technology offers very useful software tools and can be applied for the design and the implementation of application generators. We used this capabilities in a tool *EUGENE*, that is intended for the building of application generators. The meta-models are defined with EXPRESS schemata, and code templates are programmed in EXPRESS with embedded target code.

STEP aims to standardize consensus data specifications, called *application protocols*, exchanged and shared throughout an engineering community. The STEP standardization process relies on successive data models definition, based on standardized constructs reuse.

We defined a method inspired from this process, and its implementation with *EUGENE*. The goal is to build a meta-model close to the target product, formally derived from others meta-models. Templates are then applied to this meta-model. As benefits of this method, the final meta-model offers a higher abstraction of the generation process, and the complexity of code templates is greatly reduced.

Introduction

Application generators translate source specifications into target products as programs or documentations. They use data structures called *dictionaries* or *meta-models* to store the source specifications and intermediate representations. As in compilers, there are two main functions in a generator: *parsing* of the source specifications and the target code *generation*.

An application generator builder offers a means to define specification languages and associated parsers, to describe and traverse the meta-model structure and to specify the derivation on this structure. In most builders, derivation is specified with templates (or skeletons) of code. Templates contain a mixture of commands operating on the meta-models and "real" code directly inserted into the products.

STEP is an ISO 10303 standard developed to facilitate product information sharing by specifying sufficient semantic content for data and their usage. Parts of ISO 10303 are intended to standardize conceptual structures of information which are either generic or within a subject area (e.g. mechanics). Standardized parts are expressed with a dedicated technology, mainly an object-oriented modelling language called EXPRESS and a standard data access interface called SDAI.

Application generator building can benefit from the STEP technology at specification and implementation levels [11, 10]. We used these capabilities in a tool, called EUGENE, that is intended for the building of application generators [12]. The meta-models are defined with EXPRESS schemata; code templates are programmed in EXPRESS and are interpreted by a STEP data access interface.

STEP description and implementation methods

The EXPRESS language [1] is an object-oriented modelling language. The application data are described in schemata. A schema has the type definitions and the object descriptions of the application called *Entities*. An entity is made up of attributes and constraint descriptions.
The constraints expressed in an entity definition can be of four kinds: **(1)** the *unique* constraint allows entity attributes to be constrained to be unique either solely or jointly, **(2)** the *derive* clause is used to represent computed attributes, **(3)** the *where* clause of an entity constraints each instance of an entity individually and **(4)** the *inverse* clause is used to specify the inverse cardinality constraints. Entities may inherit attributes and constraints from their supertypes.

The STEP physical file format defines an exchange structure using a clear text encoding of product data for which a conceptual model is specified in the EXPRESS language. The mapping from the EXPRESS language to the syntax of the exchange structure is specified in [2].

The Standard Data Access Interface (SDAI) [3] defines an access protocol for *EXPRESS*-modelled databases and is defined independently from any particular system and language. The representation of this functional interface in a particular programming language is referred to as a language binding in the standard. As an example, ISO 10303-23 is the STEP part describing the C++ SDAI binding [4].
The five main goals of the SDAI are: **(1)** to access and manipulate data which are described using the EXPRESS language, **(2)** to allow access to multiple data repositories by a single application at the same time, **(3)** to allow commit and rollback on a set of SDAI operations, **(4)** to allow access to the EXPRESS definition of all data elements that can be manipulated by an application process, and **(5)** to allow the validation of the constraints defined in EXPRESS.
An SDAI can be implemented as an interpretor of EXPRESS schemata or as a specialized data interface. The interpretor implementation is referred to in the standard [3] as the SDAI late binding. An SDAI late binding is generic in nature. The specialized implementation is referred to in the standard as the SDAI early binding.

References

[1] ISO 10303-11. *Part 11: EXPRESS Language Reference Manual*, 1994.
[2] ISO 10303-21. *Part 21: Clear Text Encoding of the Exchange Structure*, 1994.
[3] ISO DIS 10303-22. *Part 22: Standard Data Access Interface*, 1994.
[4] ISO CD 10303-23. *Part 23: C++ Programming Language Binding to the SDAI Specification*, 1995.

STEP aims to standardize consensus data specifications, called *application protocols*, exchanged and shared throughout an engineering community. The standardization process relies on successive data model definition, based on high level and standardized construct reuse.

This paper argues that this process can be applied for the design and the implementation of application generators. The goal is to build a meta-model close to the target product, formally derived from other meta-models. Templates are then applied to this meta-model. As benefits of this method, the final meta-model offers a higher abstraction level, the complexity of code templates is greatly reduced and the reusability of meta-models is increased.

In order to illustrate the concepts, a practical example is given. The problem covered by the example is described in section 1. Section 2 shows how a generator is built with EUGENE. A first solution resolving the example problem is given. In section 3 we focus on the STEP standardization process. Section 4 shows how this process can help to improve application generator design. Then an improved solution is given.

1 A concrete application

The problem concerns the cascade in referential integrity as it is defined in the ANSI/ISO SQL92 standard. A *primary key* is a set of columns in a table called the *parent table*, which have a different value for each row of the table. A *foreign key* is the same set of columns included in another table called the *child table*: for each row in the child table, the value in the foreign key matches a value in the corresponding primary key from the parent table. The matching row in the child table is *dependent* on the matched row in the parent table.

One of the actions that can be performed on the dependent rows in a child table is the cascade when the referenced parent key value is updated or the referenced parent row is deleted. It means that, in the child table, the matching value should be updated or the matching row should be deleted.

Triggers can be used to enforce the cascade rule. Triggers represent typical data management operations that can be automatically generated from table descriptions. The remainder of this article examines the delete trigger from the table Parent to a dependent table Child. The delete trigger code is shown in figure 1.

```
CREATE TRIGGER del_Child_from_Parent
    BEFORE DELETE ON Parent
    FOR EACH ROW
    BEGIN
        DELETE FROM Child
        WHERE Child.<foreign_key> = :OLD.<primary_key>
    END;
```

Figure 1: The trigger code example

2 Building application generators with EUGENE

2.1 Overview

STEP is intended to deal with product data. STEP toolkits provide tools to specify data structures with EXPRESS and to manage databases through the SDAI. As part of conventional STEP projects, we made a STEP-based toolkit starting from NIST implementations [8].

This paper presents EUGENE, an application generator builder. Built application generators use STEP capabilities in an original way. In our research, STEP technology is used in order to manage meta-data and to produce software components.

An application generator built with EUGENE uses meta-data in order to generate a target textual representation. Meta-data come from a source specification analysis and are stored in a so-called *source language meta-model*. Derivations are programmed with imperative functions, called *translation functions*. They are made of fixed parts that are mainly either meta-model traversal routines or string constants directly put into the target and made of variable parts that are values fetched from the *source language meta-model*.

Using EUGENE, an application generator results from two activities. The first activity aims to define together in EXPRESS the *source language meta-model* and the *translation functions*. The second activity aims to produce the application generator, mainly made of an SDAI built from the results of the first activity.

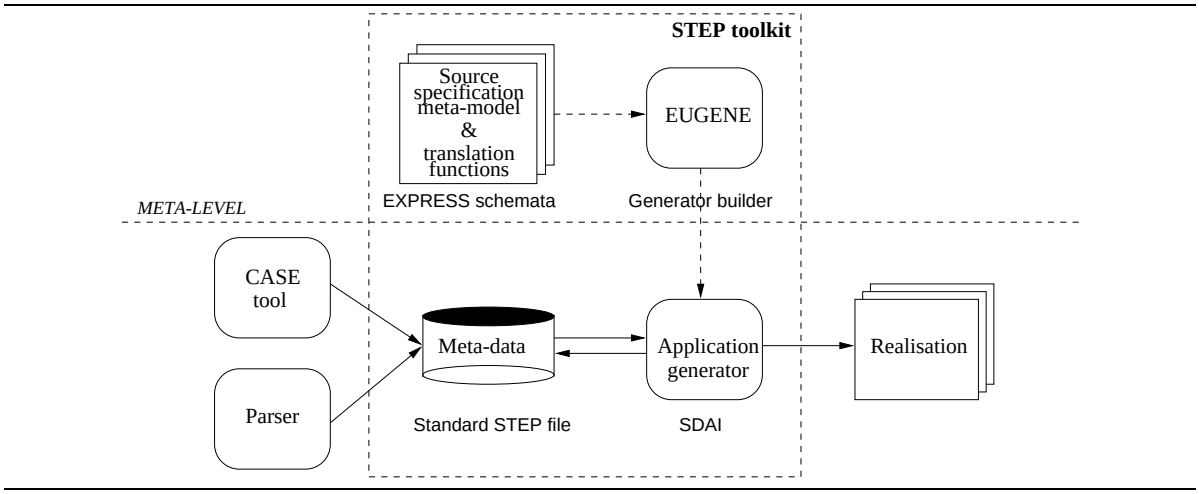

Figure 2: The building and the using of an application generator

Figure 2 shows that an application generator built with EUGENE is only a process that consumes meta-data and produces some realisation. The meta-data are themselves produced by other processes or tools that can use the automatically built SDAI in order to write standard STEP files.

2.2 The specification of the generator

2.2.1 The source language meta-model

As pointed out in [4], recognizing where an application generator can be used is difficult and often occurs too late in the life cycle. Another difficulty is to convince project managers to

invest efforts in generator development rather than concentrate the forces on the project itself. We believe that introducing a new specification language increases these difficulties. So we use the specification languages in turn on the projects as source languages that are those specified from classical application design: OMT class models, database data definition language (DDL) schemata or EXPRESS schemata.

The source language meta-model consists in a set of EXPRESS schemata that describes the source language data constructs. The main components of a source language meta-model are types and entities, describing concepts that can be used with the source language. Entities provide buckets to store meta-data while global and local EXPRESS constraints are used to ensure meta-data soundness.

Considering the example of section 1, the source language is SQL and figure 3 shows a simplified SQL meta-model. The table definition is related to a list of columns. The referential integrity constraint is referred to in the entity *column* as the attribute *foreign*: a child column that is constrained by a referential integrity rule has a foreign column relating the child column to the parent column. The inverse attribute *owner* points to the table of the columns.

```
SCHEMA sql_dictionary;

  ENTITY simple_type ABSTRACT SUPERTYPE OF (ONEOF(real_type, integer_type, string_type));
  END_ENTITY; ...

  ENTITY table;
      name : STRING;
      columns : LIST [1:?] OF column; ...
  END_ENTITY;

  ENTITY column;
      name : STRING;
      domain : simple_type;
      foreign : OPTIONAL column; ...
    INVERSE
      owner : table FOR columns;
  END_ENTITY; ...
END_SCHEMA;                                                    -- sql_dictionary
```

Figure 3: An example of source meta-model: a simple SQL dictionary

2.2.2 The translation model

The *translation functions* are written in EXPRESS and are specified in the *translation model* schema. The specification of translation functions is a programming activity in which EXPRESS is used as an imperative language. A typical translation function returns a string and is parameterized with types that are entities defined in the source language meta-model. The resulting string represents part of the target textual representation. Because of the nature of parameter types, this activity is often called *meta-programming* [2, 7].

Figure 4 shows three translation functions related to the paper example. Functions *del_trigger_of_column* and *del_trigger_of_table* can be used in order to produce the definitions of delete triggers for, respectively, a *column* and a *table*. The function *del_trigger_name* computes the name of a trigger from *column* entity. These three functions represent three kinds of

functions usually found in the translation model:

- *del_trigger_of_table* is a traversal function that iterate over all columns of a table; this kind of function is directly related to the tree structure of the source language meta-model; because the attribute *foreign* of the entity *column* is optional, the value of this attribute has to be tested with the EXPRESS built-in function *EXISTS* for all columns of the table,
- *del_trigger_of_column* is a derivation function that builds a part of the target representation. It consists of a concatenation of fixed string values and variables taken from the source language meta-model,
- *del_trigger_name* specifies the way in which a trigger name is built; it is a private function providing some readability and some re-usability.

The reader should note that only *del_trigger_of_column* function fully depends on the example problem. The traversal function *del_trigger_of_table* partly depends on the source language meta-model structure and the private function *del_trigger_name* depends on the way a trigger name is built.

```
FUNCTION del_trigger_name (col : column) : STRING;
  RETURN ('del_' + col.owner.name + '_from_' + col.foreign.owner.name);
END_FUNCTION;                                                    – get_del_trigger_name

FUNCTION del_trigger_of_column (col : column) : STRING;
  RETURN (
    'CREATE TRIGGER ' + del_trigger_name(col) + ' BEFORE DELETE ON ' + col.foreign.owner.name
    + ' FOR EACH ROW BEGIN DELETE FROM ' + col.owner.name + ' WHERE ' + col.owner.name + '.'
    + col.name + ' = :OLD.' + col.foreign.name + ' END;');
END_FUNCTION;                                                    – del_trigger_of_column

FUNCTION del_trigger_of_table (table : table) : STRING;
  LOCAL str : STRING := ''; END_LOCAL;
  REPEAT no := LOINDEX(table.columns) TO HIINDEX(table.columns);
    IF EXISTS(table.columns[no].foreign) THEN
      str := str + del_trigger_of_column(table.columns[no]);
    END_IF;
  END_REPEAT;
  RETURN (str);
END_FUNCTION;                                                    – del_trigger_of_table
```

Figure 4: First version of translation functions that create SQL delete triggers

2.3 The resulting application generator

The resulting application generator is made up of two parts:

- the first part is a lexer and a parser for the source language analysis; it produces meta-model instances;
- the second part is a code generator that processes translation functions; this process consumes meta-model instances and produces target codes.

Managing instances and processing EXPRESS functions are the main goals of an SDAI. Thus an SDAI dedicated to the meta-model is the main component of the resulting application generator. An SDAI is either generic or automatically produced from EXPRESS schemata, hence only the first part is hand-made, typically with a *lex/yacc* component relying on SDAI services.

3 The STEP design framework

A fundamental concept of STEP is the definition of consensus data specifications that describe the data to be exchanged or shared and that cover some particular application domain. These data specifications are called *Application Protocols* [9, 5].

3.1 Application protocols

An application protocol (AP) is a part of STEP that defines the context, scope and information requirements for designated domain(s) and specifies the STEP resource constructs used to satisfy these requirements [9]. APs were first proposed as a means of ensuring that STEP would provide a more reliable way of exchanging product data. APs define the form and contents of a block of data that is to be exchanged in such a way that claims of conformance to the standard for particular software products can be properly tested [1]. In order to avoid overlapping between APs, STEP standardizes common entities and common usages, called *integrated resources.*

3.2 Application protocol development process

The goal of an AP development process is the definition of an *application interpreted model* (AIM). An AIM consists of a selected set of integrated resources which are specialized, constrained or completed to satisfy the information requirements of the domain. The AIM specification is based on the reuse of integrated resources: an AIM is an EXPRESS schema that selects the applicable constructs from the integrated resources as baseline conceptual elements. Thus, this schema is specialized with additional constraints, relationships and entities which inherit from common constructs. This process is called *application interpretation*; it assigns a meaning to integrated resources in the context of a particular domain. The *application interpretation* process is a formal and well established part of the AP development process.

4 Improving the building process with STEP

4.1 Revisiting the code generation process

As in the compiler field, intermediate representations, close to the target, reduce the code generation work. As source representation, an intermediate representation is stored in a meta-model. The structure of this meta-model is related to the target language. The method used to obtain this meta-model, called *interpreted meta-model* (IMM), is similar to the method used to define an AIM from integrated resources.

The generated application is obtained through three steps (see figure 5): analysis of source specification, building of the IMM, generation of the target representation. The method is

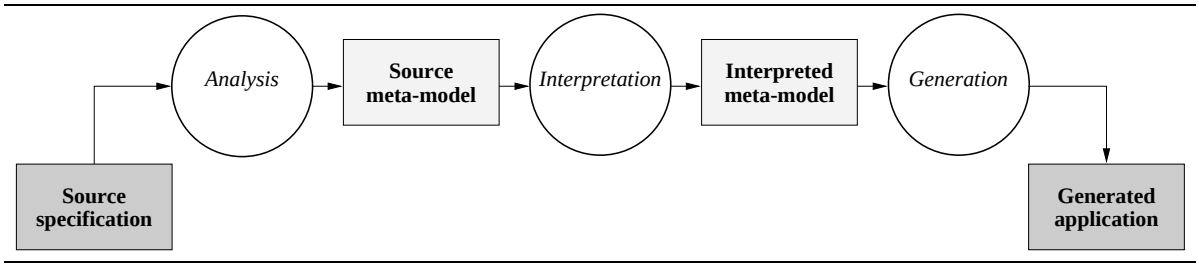

Figure 5: Generation with an intermediate meta-model

revisited to deal with the description with EXPRESS of the target languages data constructs and of the description of *translation parameters* that shall be used by the translation process. Together with the source meta-model, all these schemata are used in the interpreted meta-model. The IMM serves as the basis for the processing of *translation functions*.

4.2 Data models

4.2.1 Source and target language meta-model schemata

The source and target language meta-models consist in a set of EXPRESS schemata that describe the source and target language data constructs. Parallel to the idea of STEP *Application Protocol*, the source and target language meta-models are considered as *integrated resources*.

Considering the example of section 1, the target language meta-model is shown in figure 6. This schema contains the entities *delete_trigger* that describes all needed data in order to produce a delete trigger and *table_triggers* that contains the list of all triggers related to a table.

```
SCHEMA trigger;
  USE FROM sql_dictionary;
  ENTITY trigger;
      col : column;
      name : STRING;
  END_ENTITY;

  ENTITY delete_triggers;
      trig_list : LIST [0:?] OF trigger;
  END_ENTITY;
```

Figure 6: The target meta-model

4.2.2 The translation parameters schema

The *translation parameters* schema consists of a set of entities describing other data that are used by the translation process. It aims to describe data useful for the naming of target programming constructs such as the class or type names. It can also contain target system descriptions such as the name of basic classes used by produced classes. This schema is considered as a description of a part of programming rules usually described and used for building and for integration of an application within a target system.

Considering the example, one can think about a prefix information used to build the names of the trigger. The example translation parameters schema is shown in figure 7.

```
SCHEMA traduction_parameters;
  ENTITY naming;
      prefix : STRING; ...
  END_ENTITY; ...
```

Figure 7: The translation parameters schema

4.2.3 The interpreted meta-model schema

The interpreted meta-model (IMM) schema contains all constructs of the target meta-model schema. Subtypes of the entities from the target meta-model schema are created. The creation of subtypes allows more specific attribute definitions to be given in the context of the source language meta-model schema and of the translation parameters schema. The context is represented in the subtypes by associating them with entities from the source language meta-model and from the translation parameters. The goal is to redefine all the attributes of subtypes of the target meta-model schema as derived attributes in order to compute their value in the given context. The idea is very similar to the STEP *application interpretation* process described in section 3.2 in which the AIM is the interpretation of integrated resources in the context of a particular domain.

```
SCHEMA sql2trigger_imm;
  USE FROM trigger; USE FROM traduction_parameters; USE FROM sql_dictionary;

  ENTITY i_trigger SUBTYPE OF (trigger);
      nam : naming;                                    (* interpretation context *)
    DERIVE
      SELF\trigger.name : STRING := nam.prefix + col.owner.name + '_from_' + col.foreign.owner.name;
  END_ENTITY;

  ENTITY i_delete_triggers SUBTYPE OF (delete_triggers);
      tbl : table; nam : naming;                       (* interpretation context *)
    DERIVE
      SELF\delete_triggers.trig_list : LIST [0:?] OF i_trigger := compute_trigger_list(tbl, nam);
  END_ENTITY;

  FUNCTION compute_trigger_list(tbl : table; nam : naming) : LIST OF i_trigger;
    LOCAL i_trigs : LIST OF i_trigger := []; END_LOCAL;
    REPEAT no := LOINDEX(tbl.columns) TO HIINDEX(tbl.columns);
      IF EXISTS(tbl.columns[no].foreign) THEN
        INSERT(i_trigs, i_trigger(tbl.columns[no], ?, nam), 0);
      END_IF;
    END_REPEAT;
    RETURN (i_trigs);
  END_FUNCTION;
```

Figure 8: The interpreted meta-model

Considering the example, the IMM shown in figure 8, contains the entity *i_delete_triggers* that is considered as the interpretation of *delete_triggers*. The context of this interpretation is made up of the *table* and the *naming* entities: in order to compute the attribute *trig_list* of the entity *delete_triggers* used from the schema *trigger*, all needed data are fetched from *table* and *naming* entities.

4.3 The translation model

Programming the translation function is achieved as described in section 2.2.2, except the translation functions operate now on the IMM rather than the source meta-model. Computing instances of the IMM carries out a part of the code generation. Thus there is less work left and translation functions are simpler.

Considering the example, the translation model shown in figure 9, contains now only two functions. Comparing this model with the model of figure 4, the reader should note that the private function *del_trigger_name* is replaced by the derived attribute *i_trigger_name* of the IMM (figure 8), and that the traversal function *del_trigger_of_table* operates on a list of valid *column* instances, hence there is no need to test their existence.

```
FUNCTION del_trigger_of_column (it : i_trigger) : STRING;
  RETURN (
    'CREATE TRIGGER ' + it.name + ' BEFORE DELETE ON ' + it.col.foreign.owner.name
    + ' FOR EACH ROW BEGIN DELETE FROM ' + it.col.owner.name + ' WHERE ' + it.col.owner.name + '.'
    + it.col.name + ' = :OLD.' + it.col.foreign.name + ' END;');
END_FUNCTION;                                              -- del_trigger_of_column

FUNCTION del_trigger_of_table (its : i_delete_triggers) : STRING;
  LOCAL str : STRING := ''; END_LOCAL;
  REPEAT no := LOINDEX(its.trig_list) TO HIINDEX(its.trig_list);
    str := str + del_trigger_of_column(its.trig_list[no]);
  END_REPEAT;
  RETURN (str);
END_FUNCTION;                                              -- del_trigger_of_table
```

Figure 9: Second version of translation functions that create SQL delete triggers

4.4 Benefits

Benefits of this method are discussed in [10]. Translation functions mix up different kind of generation directives. The various schemata are intended to organize the generator design and implementation.

The separation between the *what* and the *how* is enforced, the IMM contains descriptions of the target product (*what* to generate), while translation functions describe *how* to generate.

The translation parameter schema facilitates integration of the generated product into the whole system thanks to the control of the naming and typing rules. This allows the same generator to be reused in different contexts.

5 Applications

5.1 Experiences

EUGENE has been used by several research projects to build the following generators :

- a generator of early binding SDAI in Smalltalk-80 and in Java,
- a Java graphical user interface generator for database management,
- a generator producing a *LaTeX* programmer reference documentation from the analysis of comments contained in sources of programs,

- a generator producing a Smalltalk-80 handler of a Management Information Base (in the field of Telecommunication Management).

These experiences showed that the building of application generator was simplified because only the EXPRESS language has to be learned by EUGENE users. The main task was meta-models specification that required expert knowledge of the domain.

5.2 Pros and cons

Most application generator builders first focus on the formal definition of the source specification concrete and abstract syntax. From these definitions, an internal representation structure and specialized tools as structure editors and parsers are automatically derived. Stage [4], Centaur [3] or the meta-environment described in [6] are examples of such application generator builders. These tools are very powerful, but unfortunately they are few used in an industrial context. The reasons are mainly because they are too complex and because the source specification is not always a formally defined language.

Using EUGENE requires essentially data design and imperative programming. These activities are familiar to software engineers.

The source specification is not constrained to be a textual language described by a LL(1) or LALR grammar.

As instances of EXPRESS schemata, meta-data can be exported and imported with the STEP neutral exchange structure. It enables interoperability between application generators as well between an application generator and another CASE tool (as depicted in figure 2).

When the source specification can be described with a grammar, the lack of formal syntax definition denies EUGENE of automatic parser generation. For the same reason, the consistency of meta-models is more difficult to achieve.

6 Conclusion

Various components of an application can be drawn automatically from formal specifications and design. STEP is an ISO standard (ISO-10303) for the computer-interpretable representation and exchange of product data. Parts of STEP standardize conceptual structures and usage of information in generic or specific domains. The standardization process of these constructs is an evolutionary approach, which builds successive models.

This paper has presented a STEP-based tool *EUGENE* intended to build generators and a method inspired by this standardization process.

References

[1] M.S. Bloor A. McKay and J. Owen. Application Protocols: a position paper. In *Proceedings of European Product Data Technology Days Conference*. Hermes, Paris, March 1993.

[2] Y. Ait-Ameur, F. Besnard, P. Girard, G. Pierra, and J. C. Potier. Formal Specification and Metaprogramming in the EXPRESS langage. In *Int'Conf' on Software Engineering and Knowledge Engineering (SEKE)*, 1995.

[3] P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: the system. In *ACM SIGSOFT'88, Third annual symposium on software development environment*, 1988.

[4] J. C. Cleaveland. Building Application Generators. *IEEE Software*, July 1988.

[5] W. F. Danner. STEP Data Specification Methodology. Technical report, ISO TC184/SC4/WG5 N50, 1993.

[6] Paul Klint. A Meta-Environment for Generating Programming Environments. In *ACM Transaction on Software Engineering and Methodology*, volume 2, 1993.

[7] David A. Ladd and J. Christopher Ramming. A*: A Language for Implementing Language Processors. *IEEE Transactions on Software Engineering*, 21(11), November 1995.

[8] D. Libes. The NIST EXPRESS Toolkit - Design and Implementation. Technical report, National Institute of Standards and Technology, Gaithersburg, Maryland, 1993.

[9] M. Palmer. Guidelines for the development and approval of STEP application protocols. Technical report, ISO TC184/SC4/WG4 N511, 1995.

[10] Alain Plantec. *Exploitation de la norme STEP pour la spécification et la mise en œuvre de générateurs de code.* PhD thesis, Université de Rennes I, 35065 Rennes cedex, France, 1998 (to be published).

[11] Alain Plantec and Vincent Ribaud. Data Management: From EXPRESS Schemata To User Interface. *Journal of Computing and Information*, 2(1), November 1996.

[12] Alain Plantec and Vincent Ribaud. The STEP Standard as an Approach for Design and Prototyping. *Rapid System Prototyping, RSP'98, IEEE*, 1998.