Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 238 (2009) 25–34

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Code Generation with the Exemplar Flexibilization Language

R. Heradio[1] , J. A. Cerrada[2] , J. C. Lopez[3] , J. R. Coz[4]

Departamento de Ingenieria de Software y Sistemas Informaticos
Universidad Nacional de Educacion a Distancia
Juan del Rosal 16, E-28040, Madrid, Spain

Abstract

Code Generation is an increasing popular technique for implementing Software Product Lines that produces code from abstract specifications written in Domain Specific Languages (DSLs). This paper proposes to take advantage of the similitude among the products in a domain to generate them by analogy. That is, instead of synthesizing the final code from scratch or transforming the DSL specifications, the final products are obtained by adapting a previously developed domain product. The paper also discusses the capabilities and limitations of several currently available tools and languages to implement this kind of generators and introduce a new language to overcome the limitations.

Keywords: Code Generation, Domain Specific Language, Software Product Line.

1 Introduction

Code Generation is an increasing popular technique for implementing Software Product Lines (SPLs) [1] that produces code from abstract specifications written in Domain Specific Languages (DSLs) [2,3]. The next paradox usually comes up when a DSL compiler is developed. A DSL is a specialized, problem-oriented language. From the point of view of the DSL user, it is interesting that DSL is as abstract as possible (supporting the domain terminology and removing the low-level implementation details). On the other hand, from the point of view of the compiler developer, the DSL abstraction makes harder to build the compiler. That is, the further DSL specifications are from the final code, the more difficult is to transform them into final code.

[1] Email: rheradio@issi.uned.es
[2] Email: jcerrada@issi.uned.es
[3] Email: jlopezvilanova@gmail.com
[4] Email: jrcoz@isdefe.es

doi:10.1016/j.entcs.2009.05.004

We propose to solve this paradox by taking advantage of a common property to all DSL compilers: the similitude among the final products[5]. Instead of synthesizing the final code from scratch or transforming a distant input specification, we suggest to obtain the final products adapting a previously developed domain product to satisfy the input DSL specifications. We will refer to this initial product as the domain *exemplar* [4]. Figure 1 illustrates this approach, where the generator of a DSL compiler is another compiler which is used to adapt an exemplar according to the DSL source specifications. The figure also represents a possible decomposition of this subcompiler into subgenerators responsible of different sorts of variability.

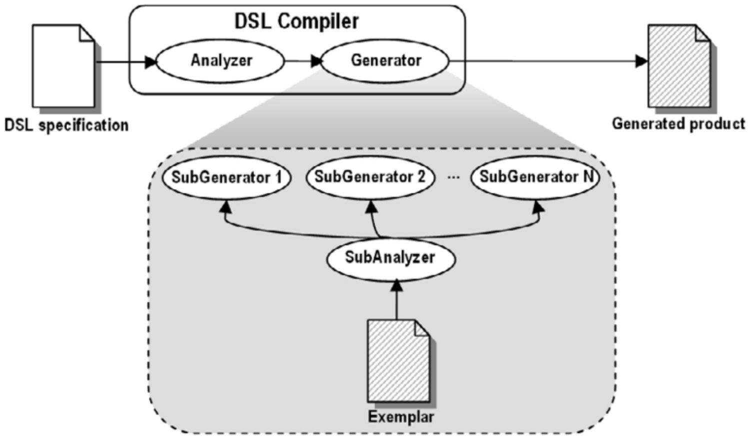


Fig. 1. DSL compiler based on the transformation of a domain exemplar.

Template languages, such as XPand of openArchitectureWare [5] or XVCL [6], use implicitly this approach, since a text template can be viewed as a piece of an exemplar with "holes". The exemplar code that is common to all the domain products is maintained in the template, whereas the variable code is replaced by holes, that are filled with *metacode* which specifies how code must change. Unfortunately, code and metacode are strongly coupled in templates. Indeed, as argued in [7], some domain variability should be implemented as *crosscutting concerns*. When a template engine does not support Aspect Oriented Programming (AOP) [8], templates may suffer *metacode tangling* (multiple variable concerns implemented simultaneously in a template) or *metacode scattering* (a variable concern implemented in multiple templates).

To overcome the templates coupling problem, the metacode should be kept out of the exemplar code. In this case, the exemplar might be processed at:

- lexical level, using *regular expressions*. Unfortunately, though regular expressions can manage text in an agile way [9], they have serious limitations because are internally implemented as state machines without memory and cannot manage nested or balanced constructs [10].

[5] Note that domain product commonalities are the main reason to develop the products jointly as a family, instead of one by one.

- syntactical level, using a *metaparser* such as ANTLR [11] or a *transformation language* such as Stratego [12] or Tom [13]. However, in most cases the simplicity of the exemplar changes does not justify to waste time either defining the exemplar language grammar or working with Abstract Syntax Trees (ASTs).

This paper introduces an intermediate solution, the *Exemplar Flexibilization Language (EFL)*, that provides new operators to overcome the regular expressions limitations. EFL also supports the integration with parsers to manage marginal complex exemplar modifications. Besides, EFL supports the implementation of *crosscutting generators*, that manage variability scattered over the exemplar, and the decomposition and combination of generators.

The rest of the paper is structured as follows. Section 2 summarizes EFL. Section 3 introduces the EFL capabilities to overcome important regular expression limitations. Section 4 lists successful applications of EFL to solve several examples taken from the generative programming literature and to develop real SPLs. Finally, the section 5 summarizes the presented work.

2 Overview of the Exemplar Flexibilization Language

A technique for developing a DSL interpreter quickly is embedding it into a *dynamic* general purpose language [14]. This way, all the host language capabilities are implicitly available from the DSL. Unfortunately, the pay-off is that the DSL concrete syntax has to fit in the host language concrete syntax. EFL is currently implemented applying this technique: it is a library of the Ruby object oriented language[6] [15]. As we will see, thanks to the Ruby extensibility, the EFL concrete syntax is reasonably usable.

2.1 Defining Generators

Figure 2 shows a simplified EFL metamodel. EFL supports the writing of generators that transform input exemplar files into output final product files according to input DSL specifications. EFL generators are written as Ruby classes that extend from the `Generator` class. This way, the generators can be easily reused by mean of the Ruby composition and inheritance capabilities. Alternatively, there is available the next *syntactic sugar* to write generators as objects of the `Generator` class:

```
my_generator = generator {
    << generator definition >>
}
```

A generator definition is composed of *substitutions*, *productions* and *generations*:

(i) A **substitution** describes the interchange of an exemplar code pattern, expressed with a regular expression[7], to new code. Crosscutting generators

[6] EFL is freely available at `http://rubyforge.org/projects/efl`

[7] Due to EFL is embedded in Ruby, regular expressions are written in the Ruby notation (delimited with the `/` symbol). For example, `my_regexp = /code/`.

often apply the same substitutions over different exemplar files. To avoid the repetitive writing of substitutions and support their reuse, substitutions are independent from the exemplar files and the final product files. The main `Generator` methods to define substitutions are:

- `sub(reg_exp, text, name`[8] `= nil)`
- `gsub(reg_exp, text, name = nil)`

A *local* substitution (`sub`) expresses the interchange of the first occurrence of the `reg_exp` regular expression to the `text` string. A *global* substitution (`gsub`) expresses the interchange of all the `reg_exp` occurrences. Additionally, the `Generator` class provides the next methods[9]:

- `del` and `gdel` to delete code from the exemplar.
- `before` and `gbefore` to insert code before the `reg_exp` occurrences.
- `after` and `gafter` to insert code after the `reg_exp` occurrences.

(ii) A **production** describes the application of a substitution list to an exemplar file to produce a final product file. `Generator` provides the next method to define productions:

- `prod(input_file, output_file, sub_list`[10] `= nil, name = nil)`

EFL supports the detection of undesirable overlaps among the code patterns of the `sub_list` substitutions.

(iii) A **generation** executes a list of productions. `Generator` provides the next method for generations:

- `gen(prod_list`[11] `= nil)`

EFL supports the detection of undesirable collisions among the productions of a generation.

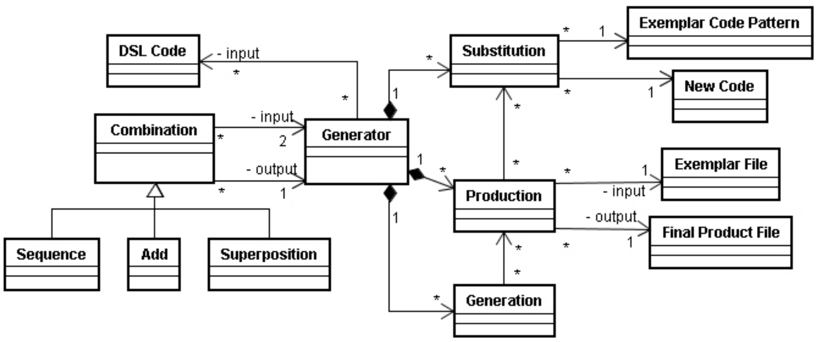


Fig. 2. Simplified EFL metamodel.

[8] Optionally, substitutions, productions and generations can be named using the `name` string.

[9] Besides, the EFL substitution capabilities can be easily extended adding the correspondending methods to the `Generator` class.

[10] The order of the substitutions in `sub_list` is irrelevant. If the `sub_list` is not specified, it will contain implicitly all the substitutions defined before the current production.

[11] The order of the productions in `prod_list` is irrelevant. If `prod_list` is not specified, it will contain implicitly all the productions defined before the current generation.

2.2 Combining Generators

For writing complex exemplar transformations, EFL provides the next binary operators to combine two generators `g1` and `g2`[12]:

(i) **Sequence**. Executes `g1` first and `g2` later:

```
g1.gen
g2.gen
```

(ii) **Add**. Returns a new generator which substitutions and productions are the union of the substitutions and productions of `g1` and `g2`:

```
(g1 + g2).gen
```

(iii) **Superposition**. Updates the substitutions and productions of `g1` with the substitutions and productions of `g2`. Those with the same name are overwritten and the remaining ones are added:

```
(g1 << g2).gen
```

3 EFL Capabilities to Overcome the Regular Expressions Limitations

3.1 The Zoom Operator

There are two fundamentally types of regular expressions engines: the *Deterministic Finite Automaton* (DFA) and the *Nondeterministic Finite Automaton* (NFA). Being irrelevant for DFA engines how the regular expressions are written, the behaviour of NFA engines, however, depends on the representations of the regular expressions[13]. According to Jeffrey E. F. Friedl [9], most of the programming languages[14] implement NFA engines because give more control to the programmer, since the representation of a regular expression sets the way the NFA engine backtracks during the matching resolution. Besides, NFA engines provide interesting features, such as capturing parentheses and the associated backreferences (`$1, $2...`), and lazy quantifiers.

Writing a complex and time-efficient regular expression for an NFA engine may be quite hard. To simplify this work, EFL provides the *zoom operator* (`>`) that supports the step-by-step writing of regular expressions. Thanks to this operator, regular expressions can be chained to specify progressively a text pattern; i. e., the expression:

```
regexp1 > regexp2 > regexp3 > ... > regexpN
```

matches the `regexp2` against the text matched by the `regexp1`, the `regexp3` against the text matched by the `regexp2`, etcetera.

[12] Of course, these operators can be combined among them. For example, you can write: `((g1 << g2) + g3 + g4).gen`.

[13] For example, an NFA engine follows different ways to match the equivalent regular expressions `regexp1 = /to(ni(ght|te)|knight)/` and `regexp2 = /tonite|toknight|tonight/` against the “tonight” string.

[14] Perl, Ruby, Python, Java, .Net languages...

3.2 Anti-patterns

Sometimes it is useful to express a pattern in negative terms: instead of specifying the features we are interested in, describing characteristics to exclude some matching candidates. To support the writing of such *anti-patterns*, many regular expression engines provide the next constructs:

- The *negated character class* `[^...]`, which matches any character that is not listed into the character class.
- The negatives *look-ahead* `(?!...)` and *look-behind* `(?<!...)`[15]. These look-around constructs do not actually "consume" any text, but they look forward or backward to "see" if their subexpressions cannot be matched. For example, the evaluation of the `/Ruben (?!Heradio)/` regular expression against the "Ruben Garcia" string only matches the text "Ruben" (i.e. the negative look-ahead queries if anything different of "Heradio" follows "Ruben", but does not consume "Garcia").

EFL provides two new constructs for writing anti-patterns:

(i) **Complement** (`o`) is an unary-operator that inverts the matching of a regular expression. That is, `o(regexp1) > regexp2` matches the `regexp2` out of the text matched by the `regexp1`.

(ii) **Minus** (`-`) is a binary-operator that excludes candidates for matching. That is, `regexp1 - regexp2` captures the text that is matched by the `regexp1` but not matched by the `regexp2`.

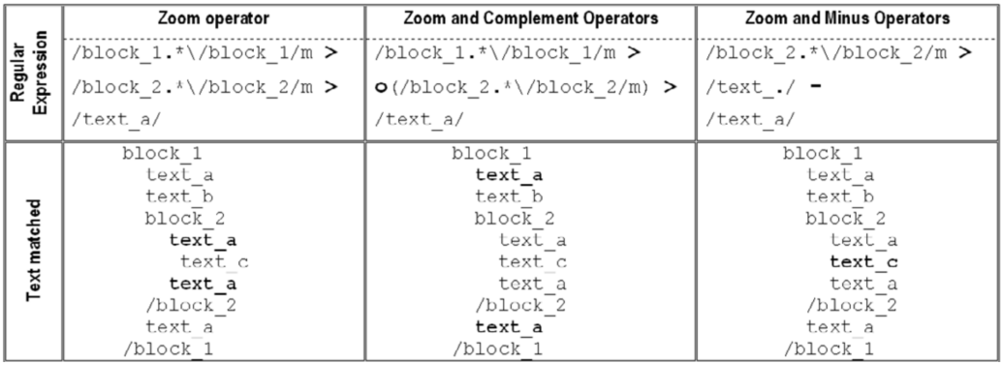
Sometimes is quite hard "to find the precise regular expression", general enough to match all the text of interest and particular enough to ignore the rest. Using the minus operator, this problem can be solved in several steps: first, a more general regular expression is written without worrying about catching some undesirable text and, then, the matching is progressively adjusted by subtracting one or more particular regular expressions.

Figure 3 illustrates several examples of the zoom, complement and minus operators. The top row shows several regular expressions built combining these operators and the bottom row highlights the result of matching the regular expressions against a given text.

3.3 Managing Nested Constructs

As it was mentioned in the introduction, regular expressions cannot actually manage nested or balanced constructs because they are internally implemented as state machines without memory. For example, a regular expression for matching any number of balanced parentheses cannot be written, because when the state machine finds the first close-parenthesis, is not able to "remember" how many open-parentheses has processed before. However, it is possible to write a regular expression for matching

[15] Unfortunately, Ruby does not support the negative look-behind construct.



	Zoom operator	Zoom and Complement Operators	Zoom and Minus Operators
Regular Expression	/block_1.*\/block_1/m > /block_2.*\/block_2/m > /text_a/	/block_1.*\/block_1/m > o(/block_2.*\/block_2/m) > /text_a/	/block_2.*\/block_2/m > /text_./ - /text_a/
Text matched	block_1 text_a text_b block_2 **text_a** text_c **text_a** /block_2 text_a /block_1	block_1 **text_a** text_b block_2 text_a text_c text_a /block_2 **text_a** /block_1	block_1 text_a text_b block_2 text_a **text_c** text_a /block_2 text_a /block_1

Fig. 3. Examples of the zoom, complement and minus operators.

until a fixed number of balanced parenthesis. For example, the next three regular expressions match until one, two and three balanced parentheses respectively:

(i) `/[(]([^()])*[)]/`

(ii) `/[(]([^()] | [(]([^()])*[)] )*[)]/`

(iii) `/[(]([^()] | [(]([^()] | [(]([^()])*[)] )*[)] )*[)]/`

Writing a `/[(] ...[)]/` regular expression for each particular case is quite hard and repetitive. Fortunately, this work can be automatized using the Ruby meta-programming capabilities. For example, the next `nested_parentheses` method receives a `levels` number of balanced parentheses and generates the corresponding regular expression [16]. Internally, this method makes a string that contains the Ruby code for the corresponding regular expression and, then, calls the `eval` method for asking to the Ruby interpreter to evaluate the string [17].

```
def nested_parentheses(levels)
    eval('@level0 = "[(]([^()' + '])*[)]"')
    (1..(levels-2)).reject {|i|
     eval("@level#{i}" + '= "[(]([^()' +
             '] | #{@level' + "#{i-1}" + '} )*[)]"')
    }
    if levels > 1 then
        eval("@level#{levels-1}" +
             '= /[(]([^()' +'] | #{@level' +
             "#{levels-2}" + '} )*[)]/mx')
        eval "return @level#{levels-1}"
    else
        eval "return /#{@level0}/mx"
    end
end
```

[16] For example, a regular expression for ten balanced parentheses would be obtained with `nested_parentheses(10)`.

[17] *We are writing Ruby code that: 1) writes more Ruby code and 2) executes the new code.*

3.4 EFL Integration with Parsers and Text Template Engines

Sometimes, regular expressions are not the best way to write certain exemplar changes. You may want to work at syntactical level (i.e., against a AST) or to use a text template.

Thanks to EFL is embedded in Ruby, EFL generators can integrate parsers[18] and text templates[19]. Figure 4 shows how to do this inside substitutions, maintaining the EFL support for detecting undesirable overlaps among the substitutions of a production and the possible collisions among the productions of a generation. The first substitution parameter is a very general regular expression that sets the exemplar scope for the parser or the template. The second parameter calls the parser or the template engine for processing the scoped text and producing the new code. Note that the scope is captured with parentheses and then is passed to the parser or the template through the associate backreference (`$1`).

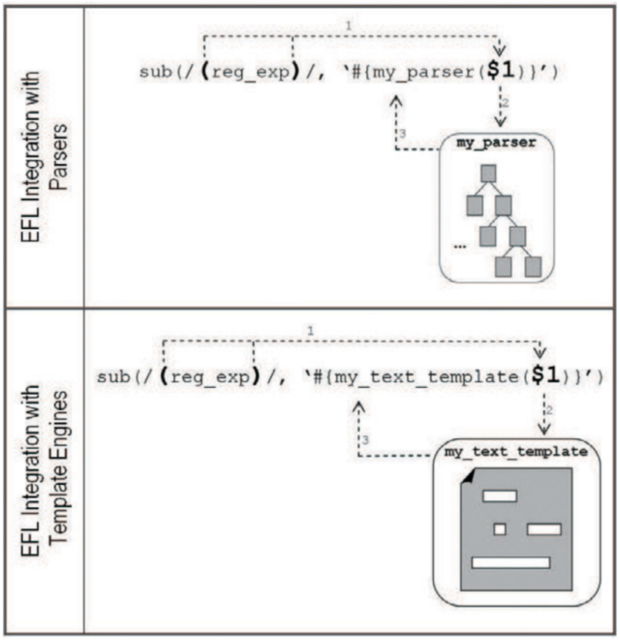

EFL Integration with Parsers
sub(/(reg_exp)/, '#{my_parser($1)}')
1
2
3
my_parser
...
EFL Integration with Template Engines
sub(/(reg_exp)/, '#{my_text_template($1)}')
1
2
3
my_text_template

Fig. 4. Example of how to integrate a parser or a text template with EFL.

4 Practical Experience and Results

At the moment, EFL has been successfully applied for developing[20]:

(i) Several examples taken from the generative programming literature (see chapters 4, 5 and 6 of [4]), including:

[18] Racc [16] and Rockit [17] are two currently available metaparsers for Ruby.
[19] ERB [18] is a valuable text template engine for Ruby.
[20] The code of many of these examples is freely available at
http://www.issi.uned.es/miembros/pagpersonales/ruben_heradio/rheradio_english.html

(a) A tool that interprets documentation embedded in SQL and turns it into external documentation in HTML (example proposed in the chapter 6 of [2]).

(b) A generator that receives abstract definitions for file formats and produces Java libraries to read the files (example proposed in the chapter 9 of [2]).

(c) The "List Container" problem proposed in [19].

(d) The "Dictionary" example proposed in the chapter 1 of [20].

(ii) A generative model that produces stored procedures in Transact SQL to load a Data Warehouse (see section 6.3 in [4]).

(iii) The m2unit tool that generates Modula-2 test cases from embedded code (see section 6.4 in [4]).

(iv) A Data Acquisition SPL for the Astrophysics Institute of the Canary Islands [21].

(v) A generative model that produces, from abstract specifications, change notifications written in PL/SQL for Oracle databases [22].

5 Conclusions

In this paper we have proposed the next shortcut for developing DSL compilers: instead of synthesizing the final code from scratch or transforming the DSL specifications, we suggest to obtain the products of a family by adapting a previously developed domain product.

We have discussed the capabilities and limitations of some currently available tools and languages (such as text templates, regular expressions, metaparsers and transformation languages) to implement this kind of generators. We have introduced the Exemplar Flexibilization Language (EFL) which overcomes some important limitations of the studied tools and languages. In addition, we have shown that, instead of being an exclusive alternative to these tools or languages, EFL can easily be integrated with many of them.

Finally, we have shown a summary of successful applications of EFL to solve several examples taken from the generative programming literature and to develop real SPLs.

References

[1] Pohl, K.; Bockle, G,; Linden, F. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[2] Herrington, J. *Code Generation in Action*. Manning, 2003.

[3] Pohl C. et al. *Survey of existing implementation techniques with respect to their support for the requirements identified in M3.2*. AMPLE Consortium, Version 1.2, 7/30/2007. URL: http://ample.holos.pt.

[4] Heradio Gil, R. *Metodologia de desarrollo de software basada en el paradigma generativo. Realizacion mediante la transformacion de ejemplares*. Ph. D. Thesis, Departamento de Ingenieria de Software y Sistemas Informaticos de la UNED, Spain, April 2007. URL: http://www.issi.uned.es/miembros/pagpersonales/ruben_heradio/rheradio_english.html.

[5] openArchitectureWare website.URL: http://www.eclipse.org/gmt/oaw.

[6] Swe, S. M.; Zhang, H.; Jarzabek, S. *XVCL: a tutorial*. Proceedings of the 14th international Conference on Software Engineering and Knowledge Engineering, Jul 2002.

[7] Voelter, M.; Groher, I. *Product Line Implementation using Aspect-Oriented and Model-Driven Software Development*. 11th International Software Product Line Conference (SPLC 2007). URL: http://www.voelter.de/services/ple.html.

[8] Kiczales, G. et al. *Aspect-Oriented Programming*. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[9] Friedl, J. E. F. *Mastering Regular Expressions*. Second edition. O' Reilly, 2002.

[10] Aho A. V., Lam M. S., Sethi R.; Ullman J.D. *Compilers: Principles, Techniques, and Tools*. Addison Wesley; 2nd edition (August 31, 2006).

[11] Parr, T. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, May 17, 2007. URL: http://www.antlr.org.

[12] E. Visser. *Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5*. In A. Middeldorp, editor, Rewriting Techniques and Applications (RTA'01), volume 2051 of Lecture Notes in Computer Science, pages 357–361. Springer-Verlag, May 2001.URL: http://www.program-transformation.org/Stratego.

[13] Balland, E.; Brauner, P.; Kopetz, R.; Moreau, P.; Reilles, A. *Tom: Piggybacking rewriting on java*. In RTA 2007, Paris. URL: http://tom.loria.fr.

[14] Fowler, M. *Language Workbenches: The Killer-App for Domain Specific Languages?*. 12 Jun 2005. URL: http://www.martinfowler.com/articles/languageWorkbench.html.

[15] Thomas, D.; Hunt, A. *Programming Ruby. The Pragmatic Programmers' Guide*. Addison Wesley, 2nd edition (October 1, 2004).

[16] Minero Aoki. *Racc: LALR(1) Parser Generator (yacc for ruby)*. URL:
http://i.loveruby.net/en/projects/racc.

[17] Feldt, *R. ROCKIT - Ruby O-o Compiler construction toolKIT*. URL:
http://sourceforge.net/projects/rockit.

[18] Masatoshi Seki. *ERB: an implementation of eRuby*. URL:
http://raa.ruby-lang.org/project/erb.

[19] Czarnecki, K.; Eisenecker, U. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000

[20] Cleaveland, J. C. *Program Generators with XML and Java*. Prentice Hall, 2001.

[21] Lopez Ruiz, J. C. *Analisis de la metodologia "Exemplar Driven Development" y de la herramienta de transformaciones "Exemplar Flexibilization Language". Construccion de una linea de productos para sistemas de adquisicion de datos en astronomia*. Oct. 2007. (send an email to jlopezvilanova@gmail.com)

[22] Coz Fernandez, J. R. *Adaptacion de EDD para incrementar la productividad del desarrollo en PL/SQL en un escenario de notificacion de cambios en bases de datos*. Oct. 2007. (send an email to jrcoz@isdefe.es)