PARSING AND CODE GENERATION TECHNIQUES TO DEAL WITH UNCERTAINTY:
EXPERIENCES FROM HIGHLY-EVOLVING AND COMPLEX SYSTEMS

Cedric Lemaire, cedric.lemaire@bnpparibas.com
BNPParibas, Paris - France

Abstract

Fuzzy or changing requirements may seriously impact the viability of complex IT systems when there is no way to obtain better requirements for a start, due to a lack of time in a competitive sector or of an immature perception about the domain. A successful development process will have to rapidly integrate any improvement or reworking of requirements.

This paper reports lessons learned from the practice of Domain-Specific Modeling in unstable or evolving domains. It shows that its efficiency in bringing agility in the development process depends to a large extent on the place it occupies by overlapping Domain and Application Engineering and on the techniques it uses to reduce the harmful effects of changes.

INTRODUCTION

Most papers on Domain-Specific Modeling (DSM) focus on the modeling and metamodeling aspects. Here we look more at the equally vital aspect of code generation and at the possibilities of parsing. We tackle parsing as a manner to use Domain-Specific Languages (DSL) and to mine existing legacy code, where the mining serves to promote code into models or integrate it better.

Hostile conditions such as immaturity and instability of a domain strongly complicate the proper unfolding of Domain Analysis, which propagates its vagueness in Domain Design and Implementation. At this level, any rectification may radically change the source code to be produced. The effect on source code may eventually be amplified by some indecisions and about-turns on the technical requirements or in the integration of third party components prone to modify their requirements too.

An evolving domain also causes similar but attenuated troubles as it provokes impacts on Domain and Application Engineering. It demands the introduction of new constructs and the refactoring of reusable components, Domain-Specific Frameworks and software. These improvements suffer from the major weakness of being too close to get the big picture, certainly leading in the future to express the concepts of the domain differently, requiring once more a refactoring or even a total rewriting of earlier changes.

To overcome such obstacles for smooth software or product line development, agile methodologies stand out as a good source of inspiration, but the Domain-Specific Modeling proves to be particularly well-adapted to reacting to changing requirements as it takes up a concrete point of junction between modeling and coding.

The Equity Derivatives area of BNPParibas, a world-wide Financial and Banking company, has acquired experience in bringing agility to modeling and coding, to protect its developments from the harmful effects of changes and to react efficiently. For instance, Domain-Specific Languages and source code generation are intensively used to build a

software product line [5] intended for performing trading on electronic markets and more. Parsing, code generation, source-to-source translation and program transformation are widely applied through CodeWorker [4], a parsing tool and source code generator.

AGILITY IN MODELING TASKS

DSM has the beneficial effect of raising the level of abstraction using concepts familiar to the application domain. It requires a medium where the domain experts and the IT participants share a same formal language, graphical or textual, where they are able to express business constructs and knowledge. They write models capturing the business knowledge and the IT team extends the meta-models each time a new construct occurs so that they can represent knowledge related to this construct in the models.

In an immature or unstable domain, it is important for the IT team to react instantly to keep the momentum of exploring the domain and reformulating preceding discoveries. Stakeholders will sharpen their abstraction of the domain mostly proceeding by trial and error, so the more attempts they accumulate, the faster they converge. Highly-evolving domains have the same demands of reactivity, to prevent the IT team from being overwhelmed and the business losing ground against competitors.

Building DSLs tolerant to rearranging

In CodeWorker, an extended Backus-Naur Form (BNF) script defines the grammar of DSLs in a declarative syntax close to BNF. It works like a recursive-descend parser to interpret DSL files for building a customized syntax tree. It provides several built-in non-terminals among those commonly used, such as the reading of C-like identifiers or floating-point numbers, but also offers some regular expression facilities. It may ignore insignificant characters using default rules available for consuming whitespaces and some types of comment (C++, XML and others), or accepting the definition of a production rule for the most exotic formats.

The paradigms of object-oriented programming do not really contribute to the reusability and reworking of declarative production rules, except in the building and maintenance of the parse tree. On the other hand, some other paradigms proposed in CodeWorker have turned out to be particularly useful. The overloading of production rules, for instance, greatly participates the reuse and specialization of existing grammars, while the principle of parameterized production rules appears well-adapted for reacting to changes in the syntax of the DSL (extension or correction).

Parameterized Rules

The notion of parameterized production rule partly draws its inspiration from generic programming, and consists of switching a BNF non-terminal to the proper instantiation of a production rule at runtime, the selection depending on a parameter linked to the non-terminal. The following example shows a sample of the grammar for a little scripting language, where parameterized production rules describe the parsing of statements:

```
// the rule for matching an instruction
instruction ::=
  // read an ident assigned to sKeyword
  #readIdentifier:sKeyword
  // switching to the rule taking
  // charge the parsing of the
  // instruction stored in 'sKeyword'
  statement<sKeyword>
  ;

// several instantiations of the
```

```
// production rule, one for each
// statement of the language
statement<"while"> ::= ...;
statement<"for"> ::= ...;
```

There are two main advantages of proceeding with parameterized production rules. Firstly, if an unknown keyword appears at the beginning of an instruction, the BNF engine will not be able to match an instantiation of the 'statement' rule and so will trigger an accurate error message. Secondly, extending the DSL amounts to no more than adding a new production rule instantiation. In the heat of the action, it happens frequently that new constructs of the DSL are expressed in the specifications before the grammar has changed. Consequently, if the developers have omitted to integrate these constructs in the grammar, the BNF engine will come up against a new keyword, for instance, and will throw an error message. The developer will have to react simply by adding the production rule instantiation, which will handle the new construct.

Reusability

To come back to reusability facilities, overloading is a way to specialize the behaviour of a production rule. The transformation of existing production rules is another way which has noticeable advantages when the original rule definition is liable to vary, but does not suit in its form without some adjustments. In practice, for instance, it may consist of reusing a production rule intended to scan a part of the input for populating a parse tree. Here is an example of a rule scanning a comparison:

```
comparison_expr ::=
  arithmetic_expr
  [
    ['<' | '>' | "==" | "!="]:op
    arithmetic_expr
  ]?
  ;
```

The approach consists of detecting some invariant pieces of BNF code and applying some transformations around these reference marks. An extended-BNF script includes the original grammar and embodies the directive #transformRule.

This directive filters interesting rules by their name, then transforms the signature and the body of each selected production rule. A translation script performs the transformations. The following example shows the result of transforming 'comparison_expr', so that it populates a parse tree:

```
comparison_expr(expr : node) ::=
  arithmetic_expr(expr.operands[0])
  [
    ['<' | '>' | "==" | "!="]:op
    => insert expr.op = op;
    arithmetic_expr(expr.operands[1])
  ]?
  ;
```

Once all transformation rules are specified, the scanner becomes the only BNF script to maintain (a scanner applies the grammar but does not build a parse tree). The parser follows automatically behind. Note that CodeWorker does not especially reclaim the writing of a scanner, unless a program-transformation script reuses it.

Mining of existing software

It happens that a DSM approach has to compromise with existing software. The developers intervening on it may be resistant to the DSM principles. One reason may be that the software has strong reliability requirements and the existing code counts several hundred thousand lines, dissuading its developers and the management from initiating a refactoring or change of development process towards Model-Driven Software Development [3], even progressively. Possibly, being particularly reactive in the integration of evolutions, they will not be convinced of adopting a new way of coding.

A piece of code is not well suited as a medium for promoting interchange of ideas or knowledge. It does not focus on the business knowledge, but rather scatters it among gluing code aware of particularities of the language, the technical framework and the implementation context (database layer, GUI, unmarshalling, computation...). Sometimes however the source code appears as the only available representation of the Business Model and the first impacted each time an evolution or a change occurs in the domain.

In equity derivatives at BNPParibas for instance, the pricer plays a central role in IT systems, as it must compute the price of any financial product and evaluate the exposure to market variations, the results interesting most of the software behind and explaining why it is the first to integrate new products or readjustments on old ones. The demands on number crunching strongly imprint the design of the application, to such an extend that the data structures do not reflect a business modeling of all recognized financial products but rather the willingness to optimize the algorithms for speed.

Parsing Source Code

Hopefully, the attributes of financial products are dispersed into the unmarshalling functions implementing the in-house communication layer embedded into the pricer. Below is an example of elementary financial product called vanilla (the programming language of the pricer is Ada [1]):

```
function Unmarshall_Vanilla(
  D : in Data_Source;
  Name : in String -- product name
  E : in Equity) return Vanilla_Product
declare
  K : Get_Double(D, Name&".strike");
  T : Get_Date(D, Name&".maturity");
  CP : Get_Call_Put_Or_Default(D,
         Name&".call_put", "call");
begin
  if K <= 0.0 then
    Error("positive strike expected");
  end if;
  return Create_Vanilla(Name,E,K,T,CP);
end;
```

The recognition of some coding patterns indirectly lead to the extraction of an underlying class diagram. The same mining approach could be applied on other types of unmarshalling functions, to extract market data representations or available pricing objects. The coding patterns are written as production rules in CodeWorker which populates a parse tree in accordance with the output of the DSL parser. For instance:

```
products["Vanilla"]
  |- name = "Vanilla"
  |- attributes["strike"]
       |- type = "double"
```

```
  |- attributes["maturity"]
       |- type = "date"
  |- attributes["call_put"]
       |- type = "Call_Put_Enum"
       |- default = "call"
```

Export to the DSL

The parse tree comes from a piece of source code, but it may have another origin, like the output of a Domain-Specific Modeling tool. Part of the model may have been described in a graphical DSL and may be required to couple with another part, perhaps written in a textual DSL.

From a parse tree, CodeWorker can generate the translation to a DSL representation. Depending on the nature of this DSL, it will enhance the design model, but perhaps also a coordination or an implementation model. Below is a possible output generated for the vanilla product:

```
product Vanilla {
  double strike;
  date maturity;
  Call_Put_Enum call_put = call;
}
```

Here is the template-based script querying the parse tree and generating this piece of DSL:

```
product <%this.name%> {
<%
foreach i in this.attributes {
  %> <%i.type%> <%i.key()%> <%
  if i.default {
    %> = <%i.default%><%
  }
  %>;<%endl()%><%
}
%>}
```

Extraction of an Underlying Meaning

An extended-BNF parse script of CodeWorker extracts the parse tree of the function bodies too. The parse tree undergoes some transformations, after which only remains the constraints between attributes, issued from the consistency checks done during the unmarshalling. Then a template-based script generates the constraints in JavaScript simply by traversing the parse tree. For instance, after skipping the declaration of a vanilla class, the translation of constraints in JavaScript becomes:

```
// returns an error message or null
function checkVanilla(Prd : Vanilla) {
  var K = Prd.strike;
  var T = Prd.maturity;
  var CP = Prd.call_put;
  if (K <= 0.0) {
    return "positive strike expected";
  }
  return null;
}
```

These constraints may interest an implementation model because they only use the classical statements of the language and do not call the standard Application Programming Interface

(API), reducing JavaScript here to play the role of an imperative constraint language. The advantage of choosing JavaScript in this project stems also from the ease of its integration in Web browsers used as lightweight clients or in several general-purpose programming languages [15][16].

Consequently, the effort of updating the design model consists of running the exploration of the source code at any time. Of course, the operation of revealing parts of the design model by mining go against the tide and give partial satisfaction only, particularly when some hard-to-detect subtleties, perhaps prone to frequent changes, cannot be raised to the models. Hopefully, it may initiate a refactoring process for replacing some handmade pieces of source code by code generation and progressively modify the development process to adopt the use of models.

AGILITY IN CODING TASKS

Changes in requirements may heavily impact the existing source code. A revision of meta-models has a consequence on the structure of the domain constructs' representation, while the choice of a new implementation design, programming language or collaboration mechanism between components leads to refactoring or complete rewriting. Hand-typing of the whole source code seems to be a colossal task, where frequent reworking will be turned out highly time-consuming, daunting and error prone, demotivating the developers writing here throwaway source code. At first sight, code generation appears as a good candidate to relieve the human actors.

When balancing between commonalities and variabilities of the domain, code generation does not act necessarily as the main contributor for bringing reactivity in coding. Some alternatives exist, depending on the features provided by the programming language. Introspection and generic programming appear very powerful in handling some types of variation. Hybrid solutions also deserve some attention. An example of this is when the generated code combines calls to the reflection API and instantiation of template classes.

However, code generation is a process which strengthens the interest of choosing DSM and confirms the pivotal role it occupies in the Adaptive Process. The code generator intends to traverse the specific model semantics issued from the modeling stage and to produce the source code. A refactoring amounts to reworking the generator, while the production of source code in a new programming language leads to extending the generator.

Building code generators tolerant to changes

By nature, enrichment of models does not affect code generators as long as it does not touch on meta-models. They also easily admit the implementation of some variants in the output. On the other hand, code generators are sensitive to changes applied on meta-models, as they have to explore the specific model semantics [14] differently or translate new constructs in the target language. Also, revisions on how the outputs have to interact with the Domain-Specific Framework may rearrange the generator.

In CodeWorker, a code generator is composed of one or several template-based scripts. Changing the generator consists of modifying some existent scripts or adding new ones. A template-based script takes charge of the generation of one output. The script embodies both rough text directly, as it should figure in the output, and scripting statements for handling variant factors. A tree structure represents the models to query. A tree is generally issued from the parsing of DSLs, but it may also come, for instance, from a database plugin.

To facilitate the propagation of changes from modeling to coding, where the tree arises as the main vehicle, the code generator should be able to detect any change in the tree structure or in the semantics it conveys. An evident manner to take precaution against modifications in

the parse tree is to type its nodes so that they belong to a structured type. This allows the interpreter to report the inconsistencies at compile-time between the parse tree structure and how the template-based script uses it.

Parameterized Functions

Parameterized functions also contribute significantly to the flexibility against changes among the range of node types the tree may have to store. Close to the notion of parameterized production rule, the principle consists of switching a function call to the proper instantiation of a function definition at runtime. The selection depends on a parameter linked to the function call. The following example shows a sample of a template-based script translating a little constraint language to C, where parameterized functions describe the generation of expressions, looking up the parse tree:

```
#include "<%this.name%>.h"

int execute(Param** params, int nb) {
  return <%
// switching to the proper function
expression<this.expr.op>(this.expr);
%>;
}
<%

// generation of the equality
function expression<"==">(expr : node) {
  expression<expr.left.op>(expr.left);
  %> == <%
  expression<expr.right.op>(expr.right);
}
... [skipping other instantiations]
```

The advantages of working with parameterized functions are similar to those of parameterized production rules. Firstly, if the interpreter cannot match an instantiation of the 'expression' function while resolving the function call, it raises a clear error message. Secondly, the implementation of a new instantiation of the parameterized function naturally extends the code generator. This appears similar to polymorphism in a object-oriented approach. These functions behave like methods overloading a member. Here, the developer proposes a function implementation for each type of expression.

Code Expansion

A template-based script may have to generate only one output, like a factory, an unmarshalling process or specific implementation of the visitor design-pattern [6] (in serialization for instance). A traditional code generation approach will generate the source code, including the commonalities. The commonalities may have to undergo some improvements. The developer will tend to intervene directly on the source code through its EDI, especially during debugging stages, rather than to report its modifications in the template-based script. As a side effect, the next source generation will erase the hand-typed code.

A solution consists of writing the source code, including markups where some variabilities are expected. A template-based script describes the variabilities specific to each markup. Then, the code generator scrutinizes the source code and executes the template-based script on each markup encountered. The script generates the appropriate outputs and the code generator injects them on markups at the same time. Below is an example of C++ source code

implementing a factory. It contains two markups, one for including the declaration of Business classes and another for implementing a builder function for each:

```
//##markup##"INCLUDES"

class Factory {
  public:
//##markup##"BUILDERS"
};
```

The template-based script describes how to generate the INCLUDES and BUILDERS section:

```
<%
switch(getMarkupKey()) {
  case "INCLUDES":
    foreach i in this.products {
      %>#include "<%i.name%>.h"
<%
    }
    break;
  case "BUILDERS":
    foreach i in this.products {
        %> static <%i.name%>* build<%i.name %>(const std::string&
name) {
    return new <%i.name%>(name);
  }
<%
    }
    break;
}
```

Then, the source code of the factory becomes:

```
//##markup##"INCLUDES"
//##begin##
#include "Vanilla.h"
//##end##

class Factory {
  public:
//##markup##"BUILDERS"
//##begin##
  static Vanilla* buildVanilla(const std::string& name) {
    return new Vanilla(name);
  }
//##end##
};
```

The injected code remains attached to its markup thanks to the delimiters ##begin## and ##end##, so the code generator is not going to confuse among hand-typed and generated code the next time source code will be expanded.

Preserved Areas

Often, the same template-based script is used to generate several outputs. The script factorizes their requirements, but it may happen that outputs reclaim some specialized implementations at well-determined locations. For instance, it may be the implementation of some methods whose skeletons were generated, but not the bodies. It occurs when the effort

of raising the level of abstraction about the behaviour of these methods seems too long and too complicated, when their implementation is too low-level. Therefore, the behaviour has not been transferred to the models and the generator cannot translate it to source code (or anything else).

Consequently, there are pieces of code to report in some outputs. The models may hold them, establishing a dependence with the choice of the target language, platform and framework. Ordinarily, the code generators can preserve the Domain Design from such implementation details, prone to radically change and thus, to impact the models in return.

Another way could be to keep these pieces of code separate from the outputs. The code generation process will have to merge them properly, but the developer will suffer from the same trouble described in the previous section when he corrects, refactors or adds new features in the source code. He might forget to report the modifications and if he does not he could make mistakes.

Like some other tools, CodeWorker proposes to embed the hand-typed code of specialized implementations in preserved areas. A well-defined comment delimits both the beginning and the end of the area. The code generator recognizes the areas and takes care to not eliminate them. This technique does not release the developer from the burden of reporting the modifications of the generated part to the template-based script, translating them in term of commonalities and variabilities.

Code generation versus dynamic execution

Code generators do not necessarily stand out as an unavoidable software asset of the Domain Implementation. Among the software family, there may exist some which require only a subset of the models, related to stable parts of the meta-models. The subset may grow and undergo some reworking, but the underlying meta-models remains invariant. If Object-Oriented Programming, Generic Programming, Reflection or any other paradigms of the programming language are sufficient to implement the variant factors, the concerned software or code components can load the models subset as data and then adapt their behaviour dynamically.

In Financial Engineering for instance, a Graphical User Interface (GUI) application may have to build forms dynamically to create, edit and validate any existing derivatives product. It is possible to write a generic GUI engine, loading the description of products at startup. The products are composed of a set of attributes, restrained to a triplet of string values: the name, type and attached documentation. The constraints between attributes are expressed as a JavaScript function (see the example of the section touching on mining), linked to each product description as a string value. This description allows the dynamic building of dialog boxes (appropriate input fields, tooltips for the documentation...) for each product, and the validation behind if the application embeds a JavaScript interpreter.

Because the source code of dialog boxes is not generated, the IT team does not have to deliver a new release of such software each time the subset changes in the model. Here, code generation is pushed into the background, requested only for translating models to a reduced form that suits the needs of the involved code components and no more. Updating the software consists of generating a new data file/record in a database and deploying it.

Maintenance of multiple translations

Application generators and reusable components issued from the Domain Implementation rarely exist for several target languages, otherwise their parallel maintenance becomes seriously hard to manage. Changing a component or application generator available for a

given target language obliges them to report after translation in components or application generators for all other target languages. Doing it by hand is slow, tedious and error prone.

However, it happens that code components and the Domain-Specific Framework were originally written in a programming language well-adapted at the time to the requirements. Code components have grown considerably and are widely used. However, some new kind of software has appeared, using a different programming language, and wishing to integrate the module within the application, rather than as a remote component.

Hopefully, source-to-source translation may partially help to automate the updating of versions available in other target languages. An automatic translation of the whole original code component is arduous to obtain. Whereas classical statements generally require a rather straight-forward transformation, the access to low-level primitives and the way they are combined must be detected by the recognition of coding patterns.

Some coding patterns are frequent and do not really vary from one project to another, like the iteration of containers, implemented differently in C++ and Java. Some others are highly dependent of how code has to collaborate with the Domain-Specific Framework, which may vary strongly from one programming language to another. Effectively, the implementation of a framework has to take advantage of the main strengths of the programming language, including the paradigms, but also the available API, standard or provided by third parties.

Specific coding patterns are tedious to list exhaustively and hard to maintain. Generally, the more time has elapsed, the more stable are the commonalities. If so, the participants can consider the rewriting of the largest part of commonalities by hand. In practice, the developers will write new template-based scripts and new components of the framework and, because of the stability of commonalities, they will not really suffer reporting adjustments from one target language version to another.

The original code component may contain numerous preserved areas, open to grow with the evolution of the design model. Rewriting commonalities does not require an automatic recognition of coding patterns. This work can be done by hand, but the translation of all preserved areas and those expected in the future are seriously time-consuming. To remain reactive, the developers will benefit from automating this translation process.

Program Transformation

The translation is easier when it consists of changing the way of coding in a code component, but keeping the same programming language. For instance, a financial library for equity derivatives has been written in C++ at BNPParibas. This library was delegating the lifetime management of class instances to the client component. Some software has then appeared, requiring an opaque memory management of instances allocated in the library. A particular reference counting mechanism, very intrusive in the existing code and changing appreciably the API, was specified.

Code generation was building about 90% of the original library and the rest was embodied in preserved areas. A parallel maintenance was required for the two versions of the financial library, as some code components using the library did not intend to integrate the new one. So firstly, the developers have written a second set of template-based scripts, generating a code compatible with the specifications. Secondly, they have written a tool for transforming the code embedded in preserved areas.

In CodeWorker, a source-to-source translation script handles the program transformation. For instance, here is a sample of C++ code to make compatible with the reference counting logic:

```
Product* p = new Vanilla("BNP",56,2007);
```

The source code issued from the transformation of the precedent C++ statement should look like:

```
Product_ref p = Factory::buildVanilla
("BNP",60,2007);
```

CodeWorker accomplishes transformation through a production rule extended with code generation features:

```
init_declaration_1 (scope : node) ::=
  class_name:C1 '*' id:V '=' "new" id:C2
  => {//leave the BNF engine for 3 lines
    scope.addVariable(V, C1 + '*');
    // @ is similar to <% or %>
    @@C1@_ref @V@ = Factory::build@C2@(@
  }
  '(' params(scope) #implicitCopy ')'';'
  ;
```

The original library continues to grow. The developers populate the new preserved area and eventually modify the others. Then the developers apply both the code generation and the program transformation processes to duplicate the library to the reference counting version. If a bug occurs in the latter library, developers have to fix it in the original library too, modifying a preserved area or a template-based script. If they want to extend the library, they integrate the evolutions in the first one and execute the building process once again.

Source-to-Source Translation

To finish without evading the translation of preserved areas to another language, a C++-to-Java translation of the financial library, for instance, could have been processed on this hand-typed code, but after the writing of more specific coding patterns taking into account the differences between these languages.

Coming back to the precedent example about reference counting logic, the translation to Java of the C++ statement should look like:

```
Product p = new Vanilla("BNP",56,2007);
```

The coding pattern for processing such a translation to Java is quite direct:

```
init_declaration_1 (scope : node) ::=
  class_name:C1 '*' id:V '=' "new" id:C2
  => {
    scope.addVariable(V, C1 + '*');
    @@C1@ @V@ = new @C2@(@
  }
  '(' params(scope) #implicitCopy ')'';'
  ;
```

However, this implementation is not representative of the effort of writing a coding pattern, because it does not combine several lines of code plus all their variants, as may happen sometimes when some constructs of a language are difficult to render in another.

RELATED WORK

The FAST (Family-oriented, Abstraction, Specification, and Translation) process [17] proposes to consider a system production as creating different members of a family, to avoid the creation of a new system each time requirements change. They have experienced mainly two methods to migrate the knowledge issued from the analysis stage into useful technology:

the creation of small DSLs and the writing of components libraries. The specifications written in a DSL were translated to code. The automation of this translation was shrinking labour costs and time to market, but the generator screens and underlying code were sometimes expensive to develop [2].

There exists a large number of code generators. Most of them are listed at http://www.codegeneration.net [8]. They can work on the command line or using a GUI, and can generate any kind of output or are intended for existing frameworks. They impose a proprietary format in the writing of models or can recognize standard meta-models, and can be coupled to a parsing tool or not.

A good collaboration between the parsing tool and the code generation engine enhances the propagation of changes initiated by domain variability and uncertainty. Terence Parr has developed a recursive-descend parser generator and a template engine, respectively called ANTLR [12] and StringTemplate [13]. The StringTemplate template engine provides a framework for accessing the extracted data in a formal way. A combination of ANTLR and StringTemplate provides a consistent approach for performing source-to-source translation.

Aspect-Oriented Programming (AOP) [9] appears like an efficient approach to reduce the impact of changes. It restrains the side-effects of a change propagation by the separation of concerns it implies on legacy source code. DSLs might also take advantage of powerful technologies turning around AOP, to propose modularity in the writing of models. These technologies are often focused on a single programming language, and a proposal [7] addresses the goal of creating new weavers from meta-specifications of a language.

CONCLUSION

Domain-Specific Modeling not only enhances the communication between the domain experts and the IT team, but also accelerates the implementation of software. The participants directly handle the constructs of the domain for the description of the problem space, expressed using one or more DSL, or better, a Domain-Specific Visual Language [10][11] if the domain is endowed with a rich graphical language or could adopt one that suits the experts. As much as possible, the IT team translates this description to programming languages automatically, capitalizing on technical and coding skills in code generators for a large extent.

As a side effect, the effort of raising the abstraction level, both in modeling and coding tasks, moves the complexity in the process of transforming models to code. So, this process doesn't fall in with the modification of requirements naturally. But the complexity of systems built today obliges the process to tolerate weaknesses in the understanding of the domain and so, to accept partial reworking or extensions at any time of the development. The same complexity drives the process to accept unusual sources for updating requirements, such as extracting a part of the design by parsing an existing code component.

The practice of parsing and code generation in the financial area of equity derivatives has revealed some useful paradigms, features and guidelines, which improve the reaction of a Domain-Specific Modeling approach against changing requirements in the meta-models or in the Domain-Specific Framework for instance. These functionalities help in spreading these changes along parse tasks and code generators. They are also flexible enough to satisfy new requirements of a system being into its stride but still prone to evolve. An example is the duplication of a reusable component to another target language, under the constraint that the original component will continue to be extended.

REFERENCES

[1] "Ada 95 Annotated Reference Manual", http://www.adaic.org/standards/95aarm/html/AA-TOC.html

[2] Ardis M., Daley N., Hoffman D., Siy H., Weiss D., "Software Product Lines: a Case Study", Software Practice and Experience 30(7), pp. 825-847, June 2000

[3] Bettin J., "Process Implications of Model-Driven Software Development", Original publication in the magazine OBJECTspektrum, 2004

[4] "CodeWorker: a Parsing Tool and a Source Code Generator", http://www.codeworker.org

[5] Czarnecki K., Eisenecker U., "Generative Programming, Methods, Tools and Applications", Addison Wesley, 2000

[6] Gamma E., Helm R., Johnson R., Vlissides J., "Design Patterns - Elements of Reusable Object-Oriented Software", Addison Wesley, 1995

[7] Gray J., "Using Software Component Generators to Construct a Meta-Weaver Framework", 23rd International Conference on Software Engineering (ICSE 2001), May 2001

[8] Herrington J., "Code Generation Network: Code Generation Information for the Pragmatic Engineer", http://www.codegeneration.net

[9] Kiczales G., Lamping J., Mendhekar A., Maeda C., Videira Lopes C., Loingtier J., Irwin J., "Aspect-Oriented Programming", European Conference on Object-Oriented Programming (ECOOP '97), 1997

[10] Luoma J., Kelly S., Tolvanen J.P., "Defining Domain-Specific Modeling Languages: Collected Experiences", OOPSLA DSM workshop, 2004

[11] Nordstrom G., Sztipanovits J., Karsai G., Ledeczi A., "Metamodeling - Rapid Design and Evolution of Domain-Specific Modeling Environments", Proceedings of the IEEE ECBS'99 Conference, pp. 68-74, April 1999

[12] Parr T., "ANTLR Parser Generator", http://www.antlr.org

[13] Parr T., "StringTemplate Template Engine", http://www.stringtemplate.org

[14] Pohjonen R., Tolvanen J.P., "Automated Production of Family Members: Lessons Learned", PLEES'02, 2002

[15] "Rhino - JavaScript for Java", http://www.mozilla.org/rhino/

[16] "SpiderMonkey (JavaScript-C) Engine", http://www.mozilla.org/js/spidermonkey/

[17] Weiss D., Chi Tau Robert Lai, "Software Product-Line Engineering: A Family-Based Software Development Process", Addison-Wesley, 1999