Modeling Untypical Document Instances using Domain-Specific Languages

Verislav Djukić
Djukic – Software Solutions
Nürnberg, Germany
+49 (0)911 4313-686
info@djukic-soft.com

Ivan Luković
University of Novi Sad
Faculty of Technical Sciences
Novi Sad, Serbia
+381 (0)21 4852-445
ivan@uns.ac.rs

Aleksandar Popović
University of Montenegro
Faculty of Science
Podgorica, Montenegro
aleksandarp@rc.pmf.ac.me

Marko Bošković
School of Computing and Information Systems
Athabasca University, Canada
marko.boskovic@athabascau.ca

[1] ***Abstract* — The paper presents an application of Domain-Specific Modeling (DSM), Domain-Specific Languages (DSLs), code generators and appropriate target interpreters in document engineering, specifically in the design and production of electronic and printed documents. In contrast to some other application domains, the design of documents and document templates usually results in a large number of models and model variations with a requirement for their validation to be incremental and exceptionally fast. Another distinction is about requirements concerning spatial arrangement of document elements. To provide modeling document variations and multi-level modeling we deploy the concept of a modifier and an extended concept of deep instantiation. Special care is paid to the synchronization between models and generated code, as well as to the improvement of user interfaces of document modeling tools. In the paper, we illustrate the application of our approach in a production of untypical document instances and an automated refinement of the DSL for specification of documents and their templates.**

Keywords: DSM, document engineering, templates, model refinement, M2T/T2M transformations, incremental rendering, model evaluation

1. Introduction

As the disciplines of Domain-Specific Modeling (DSM) and Domain-Specific Languages (DSL) constantly grow and evolve [1], new possibilities for their incorporation into the document engineering as a complex application domain constantly appear. Modern DSM tools can be successfully combined with target interpreters (i.e. document renderers) using model-to-text (M2T) and text-to-model (T2M) transformations. In this way, they become a powerful environment for model execution, as well as debugging of document types and instances during the rendering. However, the following requirements appearing in practice of document modeling prevent a wider application of the existing DSM tools in the document modeling process:

- A need to handle potentially large number of document instances and document templates;
- A need for a precise specification of topological relationships between layout elements. This kind of relationships is predominant in document modeling, requiring special care;
- A need to provide end-users with a textual modeling interface together with visually oriented modeling tools, due to a strong practical requirement to make the modeling process as fast as possible; and
- A need for numerous and efficient refinements of document templates, as well as a modeling DSL itself, with a strong requirement to keep the compatibility with already produced instances.

On the other hand, the expected benefits of a deployment of DSLs in the document modeling process are: (i) simplifying the process by providing problem domain concepts at the appropriate level of abstraction; (ii) a use of metadata and their embedding in every document instance; (iii) a possibility to transform abstract models into different target languages for rendering under different interpreters; and (iv) a possibility of simultaneous testing of document templates and documenting test results in a readable PDF or HTML format.

In the paper we present our approach to document modeling and rendering process, named DVDoc Apporach. It is a result of the research and application of DSM in document engineering. Practical benefits of DVDoc Approach are illustrated through a case of document production including a large number of untypical document instances. Besides, the modeling process is provided by a simple textual modeling interface.

Apart from a presentation of DVDoc Approach, we also discuss in the paper pros and contras of our DSL with respect to CVL – a language for the description of model variations [35], from the point of their usability in document modeling. Besides, we present different modeling interfaces necessary to provide easier description of variations and give proposals for a concrete syntax based on abstract models. Finally, we propose a construction of a domain specific

[1] A part of the research presented in this paper was supported by Ministry of Education and Science of Republic of Serbia, Grant III-44010, Title: Intelligent Systems for Software Product Development and Business Support based on Models.

query language that unifies queries over meta-model, model, and generated code.

Beside Introduction and Conclusion, this paper has seven sections. In Section 2 we consider shortcomings of current technology in modeling documents and templates. In Section 3 we give an overview of related work and specifically the CVL language for description of model variations. In Section 4 we introduce the notion of an untypical instance, which requires model variations and automated refinement of models and the modeling DSL. In Section 5 we explain the concept of a modifier. It provides specification of untypical instances at different abstraction levels. The refinement of the model, meta-model and target interpreter is described in Section 6. In Section 7 we present the use of our query language DVQL [19] aimed at performing document analyses and thus enriching the modeling tool interface. In Section 8 we present in short a declarative language for transforming model templates into text specifications that conform to any previously defined syntax.

2. State of the Art and DSM Approach

A deployment of modern software engineering achievements in formal specification and production of documents in various formats is expected to provide new solutions to several important problems in this application domain in the future. As the most important, we consider the following: (i) discrepancy between relatively simple document layout and its complex specification; (ii) discrepancy between expected and achieved layout quality at different consol devices and with different interpreters; (iii) very large resource consumption in document production, especially in real systems with a large number of untypical instances; and (iv) limited usability of existing ISO and IEC recommendations for meta-data for simultaneous tracking layout and document lifecycle. In academic research communities it is widely accepted opinion that XML languages, and particularly XML Schema language, are fully appropriate to provide a high quality description of document types, structures and contents [2, 7, 9, 11, 12, 17]. During the last several years, considerable efforts are invested in the development of XSL-FO language for layout specification [10, 17]. Following practical experiences, however, we may say that the current state of the rendering technology based on XSL-FO could not always satisfy strong requirements concerning the response time, layout quality, as well as the required level of simplicity and integration of the document production process [8]. The formal specification of documents by means of XSL-FO language is a task often more complex than classical programming in a general purpose programming language (GPL). Therefore, it needs to be simplified so as to raise the level of its practical applicability. For this reason, in different application domains, various DSLs and the appropriate generators are developed. Alternatively, GPLs with the appropriate libraries are also used [3, 14].

Numerous shortcomings of the existing tools for specification of document layouts are even more dominant than shortcomings of DSM tools [24]. Those are: (i) despite that layout modeling tools provide code generation, there is a weak or even no synchronization between created models and generated code; (ii) a lack of modularity; (iii) a complex user interface for creating models and model variations; (iv) impossibility of any modeling language adaptation; and (v) impossibility of defining various viewpoints on created models. Besides, model testing is appeared to be even more complex due to the low response time of the rendering process, as well as the noticed incompleteness of the XSL-FO renderers, as target interpreters [10].

Our DVDoc Approach with the appropriate tools and languages, implemented and integrated in DVDoc Framework [15], is based on the DSM architecture [1, 33]. DVDoc Framework comprises the tool for template description (DVDocEditor) and document renderer (DVDocRender) with the corresponding infrastructure, repository (DVDocRep), the tool for administration (DVDocAdmin), query language (DVQL) and several client applications for document specification, rendering and querying. In the scope of DVDoc Approach, we developed a graphical tool for meta-modeling and modeling, as well as the appropriate textual DSL, named DVDocLang. Videos demonstrating its use can be found at [27, 28, 29]. DVDocEditor and DVDocLang provide modeling static and dynamic characteristics of documents [6]. We have also developed a DVDocRepLang language and the appropriate code generator [20, 26]. They are used to transform abstract specifications into the concrete ones that are rendered by DVDocRender. DVDocLang has been designed to efficiently provide: (i) modeling of untypical instances that are frequently occur in practice; (ii) the use of business process software models so as to control the document production process [16, 27, 29]; and (iii) the automated refinement of templates by deploying the existing language concepts or by introducing the language extensions.

Many existing tools for document production provide specifications of just topological relationships between content units (CUs) and thus significantly reduce the expressivity of models being created. These tools do not support transformation of abstract models to target models. Impossibility of the model transformations into arbitrary target language significantly hinders the model validation. However, model validation is a very important task in practice, in the process of creating document layouts. Besides, a lack of knowledge about the nature of relationships between CUs, as well as the roles of CUs in these relationships significantly reduce possibilities of a practical application of documents in modeling static and dynamic characteristics of a system being considered. Our goal is to provide a way to resolve the aforementioned problems by the DVDoc Approach. It is done by introducing a framework with a flexible specification language that provides modeling document templates and instances, as well as further language customizations.

In this paper the notion of "untypical document instance production" is used to denote a document production process that provides an effective control of all document variations in the model validation and execution activities.

Document models may be seen, at the same time, as a kind of models of business activities in a system being observed. From this perspective, a document is a composition of CUs used for reporting about each state change or just a progress in business activities. To meet this requirement, a document is considered as a formal structure of well-formed CUs, presented in an electronic format and suitable for printing.

In DVDoc Approach, we deploy DSM and incremental document rendering techniques to overcome shortcomings of the existing technologies. The following features have already been provided: (i) a User-Driven Modeling (UDM) of document types based on modeling document instances; (ii) specifying variations of untypical document instances at different levels of abstraction; (iii) automated refinement of document type specifications by performing analyses of already created instances; (iv) incremental and full document rendering; and (v) an analysis of the equality of various layouts generated by applying different templates or template variations on the same document instance data.

Our experience from various industry projects reports that nowadays one of the essential problems preventing a wider application of DSM tools in document engineering is a lack of user-friendly oriented meta-modeling and modeling interfaces. Meta-modeling interfaces provide design of meta-models, while modeling interfaces provide designers a possibility to use all the meta-model concepts and create their models in an efficient way. In our opinion, it is crucial to provide designers the interfaces of the following types:

- Graphical representations and operations over meta-modeling concepts [30,31];
- Operations for maintaining repositories of models [4,32];
- Action reports embedded into M2T and T2M transformations and code generators [30,31];
- Domain-specific query languages providing query and manipulation operations over models and meta-models; and
- Textual representations of DSLs, meta-models, and models.

There are lot of issues related to creation or improving meta-modeling and modeling interfaces that may be a matter of an extensive research in document engineering domain. In this paper, we address two of them. One concerns providing designers a textual representation of DSLs, meta-models, and models. The second is about deploying different modeling interfaces in automated refinement of meta-models, models, transformations and target interpreters. In Section 6 we consider possible solutions to these issues through a common model of automated refinement.

For the construction, fast implementation and testing some of our DSLs, we have used MetaEdit+ Workbench. As an implementation of domain-specific query languages, we have developed DVQL [19]. It is a query and command language aimed at browsing the logical and implementation properties of document models under development.

3. Related Work

At present, we are not aware of any research related to DSM and meta-models of document templates with specific concepts covering description of untypical instances. For this reason, we refer to a Common Variability Language (CVL) [35]. It is a generic language for modeling variability in models in any DSL that is based on Meta Object Facility (MOF). Although conceptually quite comprehensive, CVL still does not offer an adequate support for handling untypical document instances. The following practical constraints limit the use of CVL for this purpose:

- The language for pattern, i.e. fragment specification has to provide the differentiation between patterns of content, structure, functionality and layout, because all of these categories are relevant in the specification of template variations;
- Specifying CVL fragments and their referencing needs to be more intuitive and thus easier for use by average users;
- Specification of a large number of variations at the level of a model significantly diminishes model understanding and usability.
- Specification of variations needs to be provided as a UDM activity, due to a requirement of systematic gathering and classification of variations;
- A specification of variations needs to be provided not only at the level of models, but also at the level of submodels (specific CUs), as well as a target language to which the models are transformed. In practice, it is motivated by the requirement that the effects of a variation may be scoped to different abstraction levels. For example, variation effects may be scoped to: instances of a particular document type; CUs; subtypes of CUs; variations specified recursively on a particular type, etc.
- For the purpose of document refinement, it is important to provide specification of operations over variations and their analyses through the template specification activity. For this purpose users need to be provided with a declarative domain-specific query language which is applicable over the modeling DSL.

Patterns play an important role in solving the problem of specifying variations. There are a lot of references covering application of patterns in DSM. In [34] and [36] the authors address understanding pattern construction and classification. In [34], the process of creating UML profiles for particular domains is presented. In [36] it is discussed the role of patterns in constructing valid DSLs.

In DVDoc Approach, a document is considered as a domain-specific five-dimension entity [9], whose dimensions are: content, structure, layout, behaviour and meta-data. For this reason, DSL patterns for document templates may include more than one of these dimensions. Additionally, there is a need to provide various kinds of referencing patterns to specify the scope of pattern effects, as well as the time of pattern interpretation. Patterns may be referenced at: (i) design time of meta-model; (ii) design time of models, i.e. templates; or (iii) interpretation (rendering) time of document instances. To the best of our knowledge, present references do not address the issues related to contextualization of multidimensional patterns in an adequate way.

In contrast to general purpose languages for specifying model variations, in DVDoc Approach we provide the DVDocRepLang language [20]. It is used for specification of patterns at the level of a code generator. Allowed variations of every model are specified using a generator, i.e. report, which produces a set of pattern definitions as its output, for a model given as its input. A check of the variations of some model instance is performed over a set of defined patterns. That allows for more freedom in work with instances that are not fully specified using the modeling DSL only. It is because patterns can express the kind of semantics that is not always expressed explicitly by means of modeling DSL only. That "additional" semantics is also considered during automated DSL refinement. During the refinement process, the meta-model is extended and the level of pattern generator commonality is raised.

Another topic of related research is about multi-level modeling. The most recent advances in this field are related to the concept of deep instantiation. In [21, 37], a concept of deep instantiation supported by DeepJava is presented. We address the similar concept in this paper in Sub-section 5.4, as well as in [27, 31].

4. Single Document Production with Untypical Instances

Since we observe a document as a structure of well-formed CUs in electronic format that is suitable for printing, a document instance may be recognized as untypical with respect to the environment or a producer. The aspect of an environment refers to the quality of software framework for document production. The aspect of a producer refers to the experience and capability of users to produce a document in a selected environment. The commonality of both aspects is that the document instance is not completely untypical, but often just one CU is seen as untypical. When untypical CU documents a particular business activity, a tie-up in production of such a document has the same effect as a tie-up in performing the business activity. Despite that the activity provides an increment of the document content, a software environment is not capable of its verification on the basis of the document specification itself. On the other hand side, in an ideal case, there is a request that only well structured documents are to be used in the verification of business activities. It is a case in various application domains that are based on document-centric systems dealing with transaction documents. As a rule, in such systems document layout is as important as its content. A typical example is the document production system in Directory Publishing, where the main products are advertisements (ads) that are to be published.

Fig.1 depicts a process of single document production based on DSM. The first, a meta-modeller creates a DSL. Then, by means of a modeling tool and the DSL being created, modellers specify documents, i.e. both document types and document instances. To raise the productivity, a modeller may easy produce documents by referencing already created document templates and CU layout styles, and then just entering document instance data. A validation of document models is done in the document rendering process. Instances that cannot be produced in this way are considered as untypical ones. The environment aspects influencing the appearance of untypical instances refer to the modeling framework, which is depicted in Fig. 1 by means of wide arrows. There are three typical environmental causes of having untypical instances: (i) a lack of the appropriate modeling concepts embedded into the DSL; (ii) imprecise M2T transformations from abstract to concrete document templates; and (iii) insufficient powerfulness of the target interpreter, i.e. document renderer.

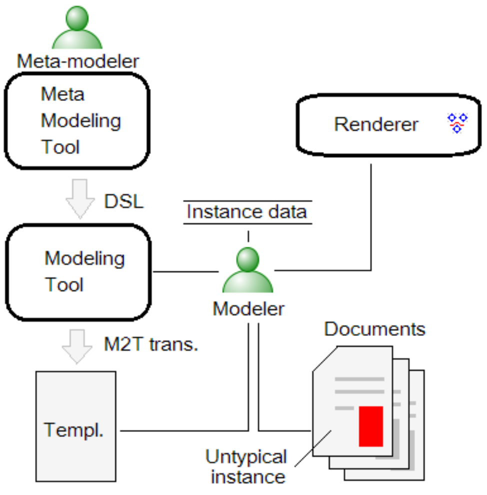

Fig. 1: A document production based on DSM

The document production handling a large number of untypical document instances is inefficient and expensive, particularly if general purpose graphical or text tools are used. The appearance of an untypical instance diverts production activities from the expected workflow and decreases the level of their automation. The main characteristics of untypical instances with respect to the impact on the document production process are:

- During production, untypical document instances cannot be abstracted as any known type or template, or cannot be described using existing DSL concepts, or cannot be rendered;

- A document producer is capable of perceiving untypical document instances by using his or her own experience, personal creativity, and the level of knowledge the of modeling framework and language;
- A significant increase of a number of untypical instances recognized in practice may be caused by some of the following factors: (i) the modeling language is not semantically rich enough for a particular domain; (ii) the template definitions and template variations are incomplete; and (iii) the users have inappropriate knowledge of the modeling language or they feel difficulties in its deploying through the existing modeling framework;
- Untypical instances are also the ones for which M2T transformation cannot produce the expected outcome or for which a spatial distribution of CUs is inappropriate; and
- A document having an untypical instance is most often specifiable. It can be rendered up to the beginning of a certain CU, as well as from its end.

Single document modeling is characterized by two important facts. The first one is that for users it is far easier to construct documents using domain-specific concepts of the appropriate level of abstraction. For those reasons, a use of graphical DSLs is completely justified. The second fact is that users spend most of their time for entering instance data and frequent switching from textual to graphical interface, which significantly slows the production process. Therefore, a synchronization of graphical and textual model representations, as well as a synchronization of model and code is a still open issue in DSM [24], and specifically in document production process. The current state of our research related to issues reported in [24] may be found in [30, 31]. Here we just outline the main topics of these research activities:

- A synchronization between models and generated code;
- A use of action reports as M2T and T2M transformations and interfaces to the generated code, interpreter environment, and models;
- A use of generic functions and a query language as modeling interfaces;
- An intensive construction and application of submodels and transactions;
- Testing the target interpreter and transformations; and
- Automated documenting of testing process.

The rest of the text about untypical instances is related to the single production of advertisements and associated documents in Directory Publishing. Such a single production covers an incremental, user-driven modeling of documents. It includes validation of temporary document states, submodels and state-to-state increments through document rendering. To address the aforementioned issues in such kind of document production we propose improving the meta-modeling and modeling interfaces by providing: (i) specifications of document variations and synchronization of abstract and concrete models at the level of code generator; (ii) a systematic refinement of the DSL by means of an analysis of already produced instances; and (iii) a domain-specific query and command language for manipulation over meta-models, models, and generated code. In the remaining sections we present all of the three improvements, while a particular attention with more details has been paid to the first two.

5. A Modifier as a Descriptor of Document Variations

A DVDoc modifier of documents and their templates, or a modifier for short, is a DVDocLang concept used to specify variations of document instances from the template or variations of document templates. By DVDoc modifiers a designer may specify variations:

- At the level of all meta-model concepts: graph, object, relation, role, port and property;
- For all layout types, such as: texts, tables, lists and figures; and
- At the level of the target language concepts used in generated code.

In our approach, we provide embedding meta-data into the document instances, as well as the inheritance from logical and implementation concepts, such as template, pattern, or concrete PDF or HTML document. Thanking to that, as well as to modifiers, the practical problem of continuous increase of a number of new and logically unrelated templates may be solved. In practice, if users recognize the existence of untypical instances they may resolve it without modifiers only by introducing new templates. However, these new templates are formally different, despite that may be very similar to some of the previously created ones. In this way, an explosion of a number of templates may occur. On the other hand side, by applying modifiers, instead of introducing a new template, a user may introduce a variation of the existing one. In this way, it may be guaranteed that the same template may be successfully used to generate various document instances, including those ones that were previously recognized as untypical. The application of modifiers also allows for:

- A simplification of concrete textual syntax of the modeling language;
- Avoiding problems in document production caused by using incompatible languages and formats; and
- Systematic refinement of DSL meta-models.

A higher level of automation of the document production process is also achieved by applying modifiers onto already created document instances, stored in a repository or a document management system. Besides, modifiers are applied to specify variations of model driven client applications or applications automatically generated from models, as it is presented in [18, 23, 29, 31].

To illustrate a practical motivation for a consideration of document variations, in Fig. 2 we present a base document layout with its two variants. It is an example of the document layout of the "Offer" document type. A base layout is presented in the left hand side of Fig. 2, while two variations,

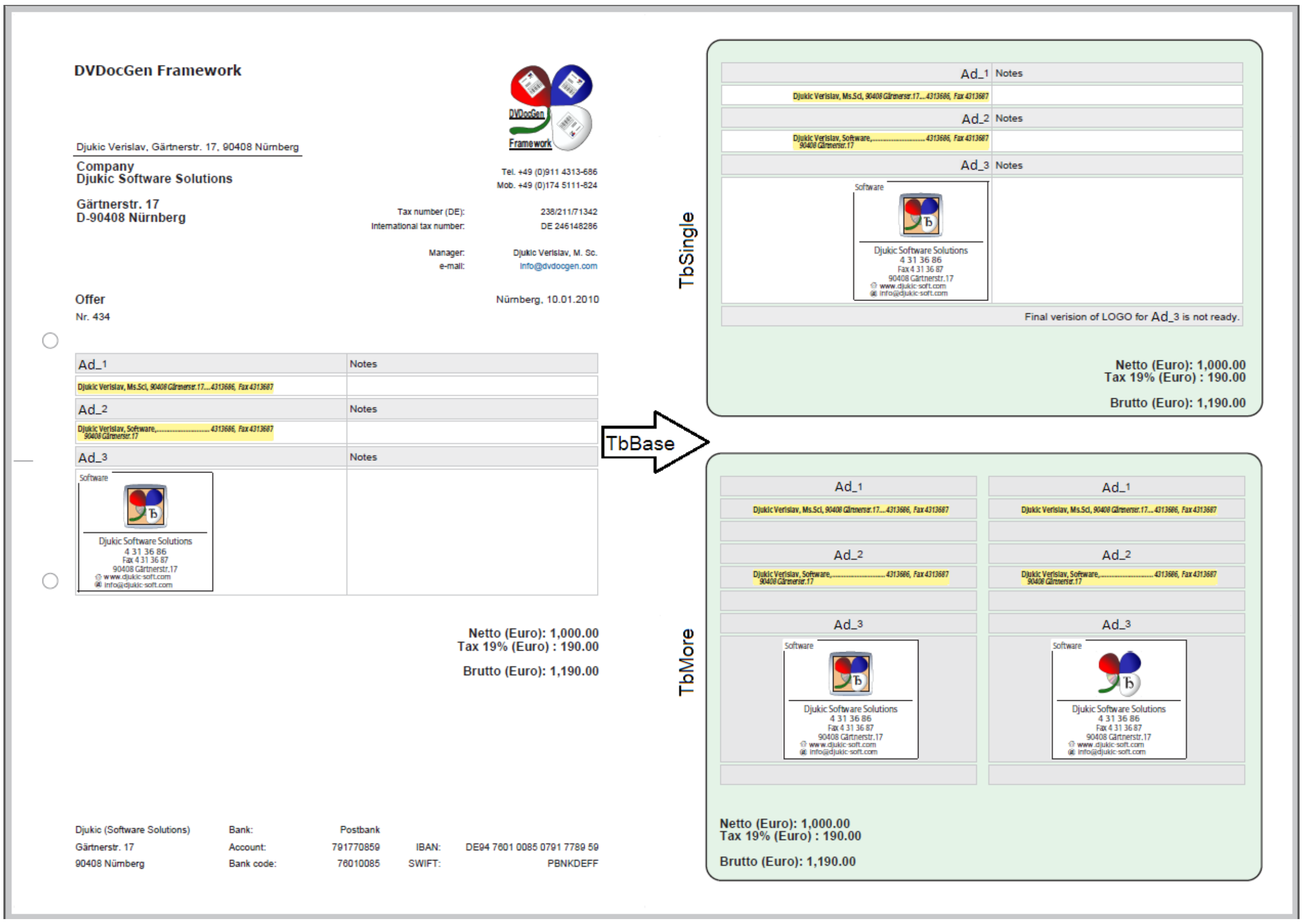

Fig. 2: Document layout variants

one of a table CU and the other of a text CU, are presented in the right hand up and down sides.

In the base definition, the TbBase table with ads has two columns, within which the content is aligned to the left. The text on the right hand down side under the table is the ad price. The first variation of the TbBase table, marked as TbSingle, is presented in the right hand up side. The content of the first column is right aligned, while the content of the second one is left aligned. There is also one text line more with the "Final version of LOGO for AD_3 is not ready" remark at the end of the table. The text with the price has the same position as in the base layout. The second variation of the TbBase table and the text is market with TbMore. Those are two tables with one centered column only. Under each picture, an empty cell for notes is placed. In the base layout, the same cell is placed in the second column. For TbMore, the following rule applies: The number of columns to be shown is equal to the number of alternatives for ads with logos, but at least one and the most three are allowed. The price is under the table, left justified, spread in three lines. The document layout from Fig. 2 is used in the examples given in the following text.

The document from the Fig. 2 is a typical example for which a single document production using the general purpose tools such as Winword or CorelDraw, would be slow. In these tools, variations of content units cannot be specified in a formal way. DVDocEditor [28], as a DSM tool for modeling of documents and templates, provides a creation of semantic relationships between CUs, description of variations and their automated classifications. Each variation is to be validated by generating a PDF document. The transformation language is similar to the MERL. It is simple and flexible enough to be applied to any DSL [20, 26].

For the purpose of illustration of various approaches in template modeling, as well as a relationship between abstract and concrete syntax, MetaEdit+ Workbench [4] has been used. In contrast to DVDocEditor, it is much more suitable for a fast construction and testing of graphical DSLs and their simple transformations to the concrete syntax. The target language used for transformation of abstract models is DVDocLang. More details about DVDocLang a reader may find in [6, 8, 13], where we have presented its syntax, a comparative analysis to XSL-FO with a large number of examples, as well as the rendering response times on the target interpreter.

In the following text, we illustrate three modeling approaches how to create DSL scripts in the concrete syntax that are based on the subtyping concept. The concept of subtyping denotes a relationship between two templates, where a template object has a role of the subtype of another template object. It is specified by means of a set of new properties, or by applying default new values of the existing ones. The properties may be of the different document dimensions: layout, content, structure or dynamic characteristic properties.

We present examples of a template subtyping with and without the application of modifiers, as well as a subtyping via modifiers only, without object subtyping. In general, a language designer decides about the scope of modifiers being modeled, when they are applied from document instances, by selecting the appropriate language expressions. In our DSL, various language expressions provide default scoping of modifiers (i) to a current instance of a CU; (ii) to an instance and the CU type within the current document; (iii) globally, i.e. to the whole document instance; and (iv) to all subsequent instances that will be produced latter on. In DVDocLang, a named modifier is of the type (i), i.e. it is scoped to a concrete instance of a CU, for which it is applied. Unnamed modifiers are of the type (ii), i.e. they are scoped to an instance and the type of CU within the document instance. In our example, these are the tables of the TbBase type. Global modifiers are of the type (iii), while proactive modifiers are of the type (iv).

For practical reasons, we decided not to create specific language constructs just to define the scope of a modifier explicitly. Instead, common language constructs are used with their default scoping of modifiers. Such an approach makes the design process faster and more efficient.

5.1. Subtyping via properties or unnamed modifiers

By Fig. 3 we illustrate a usage of the template subtype concept on a document presented in Fig. 2. A model from Fig. 3 is just one of many possible abstract models that may be used to generate various concrete template specifications from the same abstract model, by means of the same code generator, and by applying different DSL concepts.

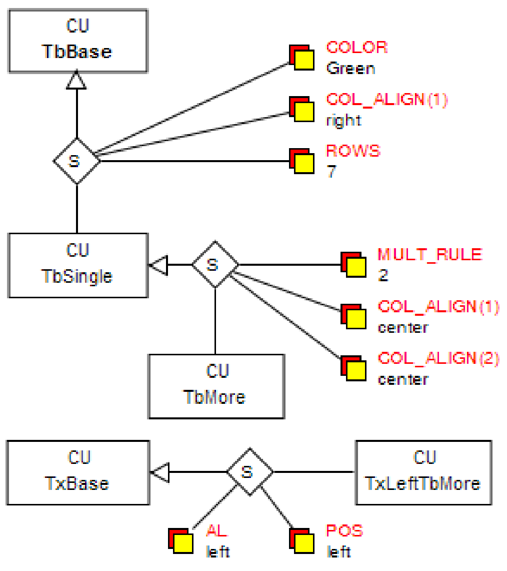

Fig. 3: An abstract model with subtyping via properties

Two types of CUs are given in Fig. 3: tables and texts. CUs are represented by rectangles. Properties are denoted with small double-rectangles. Templates are related to properties using the subtype relation (S). The following template subtype objects are created: TbSingle, TbMore and TxLefTbMore. The following subtype relations are established: S(TbBase, TbSingle), S(TbSingle, TbMore) and S(TxBase, TxLeftTbMore). Values of the properties are redefined as: `COLOR`, `COL_ALIGN(1)`, `COL_ALIGN(2)`, `MULT_RULE`, `ROWS`, `AL` and `POS`.

By examples 1-10, given in the following text, we present selected modeling techniques based both on graphical and textual interface.

While in Fig. 3 we illustrate modeling of subtypes directly via properties, in Example 1 we illustrate how subtypes are derived by predefining values of a set of simple properties or by introduction of the subtype specific properties. According to this, the simplest concrete syntax is of the form `<CUid>value`, where `CUid` is a CU type identifier and `value` is a property value. Such textual representation of document models given in a concrete syntax is named in our approach as "DSL script" or "logical script". In our approach, DSL scripts are given in the DVDocLang language.

Example 1: Here we present in the concrete syntax a DSL script representing the document instance data. The script conforms to the document "Offer" from Fig. 2. It illustrates modeling of subtypes directly using properties.

```
The Base Variant:
<TbBase>AD_1;;Notes ...
<TxBase>Price...
The TbSingle variant:
<TbSingle>AD_1;;Notes...
Final verision of LOGO for AD_3 is not ready
<TxBase>Price...
The TbMore variant:
<TbMore>AD_1;;Notes...
<TxLeftTbMore>Price...
```

Texts of the form AD_1;;... specify the contents of tables from Fig. 2, where ;; is a cell separator and each new line is a row separator. □

The abstract model from Fig. 3 is transformed into the DSL script of a template using MERL generator [4]. The generator is presented in Example 2, while the generated template is given in Example 3.

Example 2: A template generator providing subtyping via properties is given:

```
Report 'Subtyping by properties'
'[' id ']' newline
'<TEMPLATE>=sybtype_of:BaseTemplate' newline
foreach .ContentType
{
do ~SubtypeOf ~BaseTypeFor.ContentType
  {'<' :Name;1 '>=INHER_FROM:' :Name; ','
```

```
    }
    do ~SubtypeOf>BaseOrSubtype
    {
      dowhile ~RedefPropsForSub.LayoutProp
      {
        :PropertyName ':' :PropertyValue  ','
      } newline
    }
  }
endreport
```

Example 3: A generated template with subtypes, specified in DVDocLang is of the form:

```
[Subtyping by properties]
<TEMPLATE>=sybtype_of:BaseTemplate
<TbSingle>=INHER_FROM:TbBase,ROWS:7,
COLOR:Green,COL_ALIGN(1):right
<TbMore>=INHER_FROM:TbSingle,MULT_RULE:2,
COL_ALIGN(1):center,COL_ALIGN(2):center
<TxLeftTbMore>=INHER_FROM:TxBase,AL:left,
POS:left
```

To generate a complete document specification, a DSL script from Example 1 is to be combined with this template specification. □

In this, as well as in the following approaches to the specification of abstract templates, we may use a concrete syntax of the form: `<CUid.p1: v1, p2: v2, pn: vn>value`, where `p1:v1, p2:v2, pn:vn` is a list of simple properties with their values. We call it "unnamed modification" and such simple properties "unnamed modifiers". In Example 4 we give such a DSL script for the TBSingle variant of the template. The table identifier is always the same (TbBase), while the variations are given by means of a property list. If there is a significant raise of a number of unnamed modifiers in DSL scripts, then a better classification and more precise definition of templates is needed.

Example 4: A DSL script for TbSingle with simple properties is given as:

```
<TbBase.COLOR:Green,COL_ALIGN(1):right,
ROWS:7>AD_1;;Notes...
Final verision of LOGO for AD_3 is not ready
<TxBase.AL:left,POS:left>Price...
```

It is semantically equivalent to the TbSingle variant from Example 1. The difference is that TbSingle does not have to be explicitly derived as a subtype using transformations, while properties have to be given at the base type TBase. □

A common characteristic of Examples 1, 2 and 3 is that the variations are defined in the model, as well as in the code generator. Since a semantic is formally specified at the level of model relations and transformation formulas, we have a possibility to provide not only direct transformations from models to DSL scripts, but also the inverse transformations from DSL scripts to models, without loss of information.

5.2. Subtyping via named modifiers

To further extend our DSL with neccessary domain specific concepts for templates modeling, we introduce a new meta-type, called the "named modifier". It is used for grouping and identifying sets of properties that are often used.

As it may be noticed in In Fig. 4, a named modifier is depicted with the symbol <Mod:>. Properties are grouped into non disjunctive subsets using the equivalence relationship, denoted with ⇔. Each subset has a short name pointing to the meaning of the modification being modeled in a formal way.

In the model represented in Fig. 4, modifications are grouped into subsets whose names are "gColor", "tRNote", "twoTables" and "priceLeft". For example, "tRNote" is a short name derived as an acronym of "**T**able **r**ight and **note** row"; "priceLeft" is for specifying a position and alignment of the text showing a price; etc. These abbreviations are used as names of modifications.

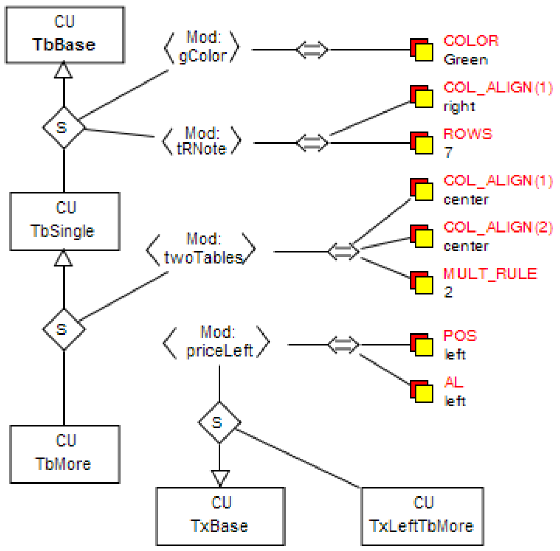

Fig. 4: An abstract model with subtyping via named modifiers

Example 5: Here we present a DSL script for TbBase, TbSignle and TbMore tables and TxLeftTbMore text with named modifiers. The script comprises three variants of the template table definitions.

Template table TbSingle is not predefined using simple properties. Instead, it is specified by means of the allowed named modifiers from the DSL script over the TBase table.

```
The Base variant:
<TbBase>AD_1;;Notes...
<TxBase>Price...
The TbSingle variant:
<TbBase.gColor.trNote>AD_1;;Notes...
Final verision of LOGO for AD_3 is not ready
<TxBase>Price...
The TbMore variant:
<TbSingle.twoTables>AD_1;;Notes...
<TxBase.priceLeft>Price... □
```

A definition of a modifier can also be generated and then saved as a value of the global attribute "modifies", as it is shown in Example 6.

Example 6: Modifiers and subtype definitions:

```
[Subtyping by named modifiers]
<TEMPLATE>=modifiers:"('gColor'=COLOR:Green;
'tRNote'=ROWS:7,COL_ALIGN(1):right;'twoTable
s'=MULT_RULE:2,COL_ALIGN(1):center,COL_ALIGN
(2):center;'priceLeft'=AL:left,POS:left;)"
<TbSingle>=INHER_FROM:TbBase,
ALLOWED_MODIF:(gColor;tRNote)
<TbMore>=INHER_FROM:TbSingle,
ALLOWED_MODIF:(twoTables)
<TxLeftTbMore>=INHER_FROM:TxBase,
ALLOWED_MODIF:(priceLeft)
```

By means of INHER_FROM property, a reference to a base template is defined. A base template is the superordinated template, from which subtypes are derived. In our example, it corresponds to the table on the left hand side of Fig. 2. In contrast to Example 3, there is no here specifications of simple properties describing characteristics of subtypes. Instead, variations of CU types are allowed by means of ALLOWED_MODIF clause. □

An advantage of this approach is a possibility to specify allowed variations explicitly. These variations are domain specific and often related to particular CUs and layout types. In this way, we provide a possibility to extend the document specification language with domain specific phrases. Well defined named modifiers may direct developers how to improve the modeling framework with new concepts and operations. Target interpreters may also be customized by introducing operations that are consequences of the modifiers existence. An example is rendering of specific table types. Suppose that we have some modifiers describing table variations. During the rendering, they need to be mapped to a set of primitive table drawing operations. However, if the modifiers are used frequently, it is more convenient to enhance the target interpreter by new functions directly supporting these modifiers, instead of using sets of primitive operations.

5.3. Modeling without object subtyping

Aforementioned subtyping approaches and already presented abstract models of templates are not aimed at any significant reduction of a number of templates being used. On the contrary, by Example 7 and Fig. 5 we present another approach that results in the smallest number of templates being used. Subtypes are not given explicitly. Instead, each modifier or a combination of modifiers is allowed to change the base template. In this way, a massive creation of new templates is avoided.

Example 7: A DSL script specifying instance data without template subtypes is given:

```
<TbBase.gColor.trNote.twoTables>AD_1;;Notes
...
<TxBase.priceLeft>Price...
```

From the DSL script, only the TbBase table template is referenced. Variations are referenced using named modifiers gColor and trNote. The same holds for the TxBase text.

A template specification created from the abstract model from Fig. 5 is slightly different from the one represented in Example 6. It is of the form:

```
[Without object subtyping]
...
<TEMPLATE>=modifiers:"('gColor'=COLOR:Green;'
tRNote'=ROWS:7,COL_ALIGN(1):right;'twoTables'
=MULT_RULE:2,COL_ALIGN(1):center,COL_ALIGN(2)
:center;'priceLeft'=AL:left,POS:left;)"
<TbBase>=ALLOWED_MODIF:(gColor;tRNote;twoTabl
es)
<TxLeftTbMore>=ALLOWED_MODIF:(priceLeft) □
```

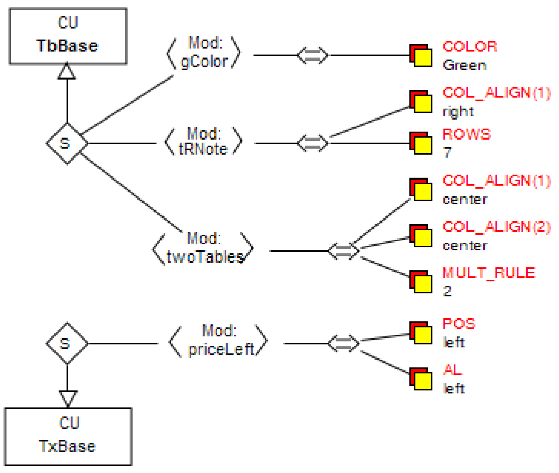

Fig. 5: An abstract model without object subtyping

By combining modifiers, any layout variant of TbBase, TbSingle or TbMore CU from Fig. 2 may be specified. From the viewpoint of end users participating in the simple document production, such DSL scripts are the most convenient, since users are not overwhelmed by a large number of templates. Besides, a number of named modifiers necessary in practice is reduced usually to only a few ones, whose names are not hard to remember.

All of the aforementioned cases of modifications, modifiers and untypical instances are the ones for which both templates and document instances can be modeled using just simple properties. However, for practical reasons, we introduced new domain-specific concepts at a higher level of abstraction.

There are also cases for which document instances cannot be modeled by means of the existing language concepts only. In such cases, property values are to be controlled by constraints. An example is a rule for the TbMore table from Fig. 2, by means of a number of table columns and a table cell distribution may vary, depending on the nature of input data. To provide a specification of such constraints, the most existing modeling languages are complemented with a domain specific, or a general purpose constraint language. For these purposes, however, in DVDocLang the MULT_RULE property is used with a value set to a function call, where the function implements the constraint.

By Example 8 we illustrate how to specify a constraint over a CU type. An alternative way for this is based on a DSL script. It is illustrated in Example 9.

Example 8: A constraint named LSLogoCount is specified over the TbBase CU type by the following script:

```
[LSLogoCount]
if LS.LOGO.Count<1
  <TbBase>=MULT_RULE:1
elseif LS.LOGO.Count>3
  <TbBase>=MULT_RULE:3
elseif
  <TbBase>=MULT_RULE:[func:LS.LOGO.Count]
endif
```

When `[func:LS.LOGO.Count]` stands for a property value in the application of LSLogoCount, the real value is calculated during the document rendering process, by invoking the function `[func:LS.LOGO.Count]`. □

Example 9: By means of `<TbBase>=MULT_RULE`, we predefine a layout of the TbBase CU type and create a content of a document instance. A DSL script for the TbMore table is given:

```
<TbMore>=MULT_RULE:2
<TbMore>AD_1;;Notes...
```

MULT_RULE:2 denotes that from one table, two new tables with the same layout are created, while the content is filled in the specified order. □

Apart from constants and functions, values of properties may also be complex expressions and comments.

Complex expressions may be defined as property values by using embedded functions in DSL scripts. However, the real values of these expressions are calculated during the rendering process. Therefore, we introduce a concept of the implementation script. The implementation script is generated from a DSL script through the rendering process. Despite that it contains all implementation details about the document instance, such as physical details about sizes and positions of pictures and fonts, it may be still independent of the selected output format. However, it helps to calculate real property values from previously defined complex expressions. More about implementation scripts is given in Section 6.

In Example 8, we deploy a complex function as a property value. It is a function LS.LOGO.Count. By such kind of expressions a user may influence both logical and implementation document features, at the same time.

If the interpreter cannot resolve a property value or some layout rule, it considers it just as comment.

If a user is not capable of precisely defining constraints, then she or he may use some less restricting alternatives introduced also by means of modifiers. Such special kind of modifications is called "informal". The next example illustrates a use of informal modifications.

Example 10: Here we present three variants of informal modifiers for non-resolvable modifications. The first is a comment given as the modifier name of a CU; the second is a free comment; and the third is a reference to a picture:

```
<TbBase."Put in 3 columns">AD_1;;Notes
```
or

```
/* Fit tables in three columns */
```
or

```
<LOAD>http://Examples/PctExample.png □
```

With respect to the level of formality, there is a crucial difference between Examples 9 and 10. While the modifications in Example 9 are used in a precise and formal way (no matter that all layout parameters are not defined yet), the modifications in Example 10 are completely informal and can be used just as remarks. However, in the process of model refinements for templates and document instances, deploying informal modifiers may also be very useful. Namely, if a user has knowledge about the business activity creating a CU, informal modifiers may serve as source information how to refine the DSL so as to provide an automation of the activity.

5.4. Modifiers and multi-level modeling

Approaches in constructing a concrete syntax of the DSL for the description of model variations are related to the multi-level modeling approach []. In this regard, two important characteristics of DVDocLang are: (1) it is a language whose expressiveness is not limited to two levels of abstractions only; and (2) by means of modifiers, it provides specifying model variations at the same abstraction level, as well as at the different levels of abstraction.

In our approach, modifiers are globally defined at the level of a graph meta-concept, using the TEMPLATE command, as it is presented in Example 11.

Example 11: A DSL script that illustrates modifiers and deep instantiation is given:

```
(1)  <TEMPLATE>=modifiers:m1(defm1),m2(defm2),…
(2)  <C1>=p1:vp1;p2:vp2;…
(3)  <C1>val_ic1
(4)  <C1.p5:vp5>val_ic1
(5)  <C2>=INHER_FROM:C1;…;p3:vp3;…
(6)  <C2>=INHER_FROM:C1(2);…;p3:vp3;…;NO:p1
(7)  <C2>=INHER_FROM:PathToFile.pdf;…;p3:vp3;…
(8)  <C2>=INHER_FROM:repID;…;p3:vp3;p4:vp4,…
(9)  <C2.ID:InstFromPDF>val_ic2
(10) <C3>=ALLOWED_MODIF:m1,m2;…;p5:vp5;…;NO:p3
(11) <C3>val_ic3
```

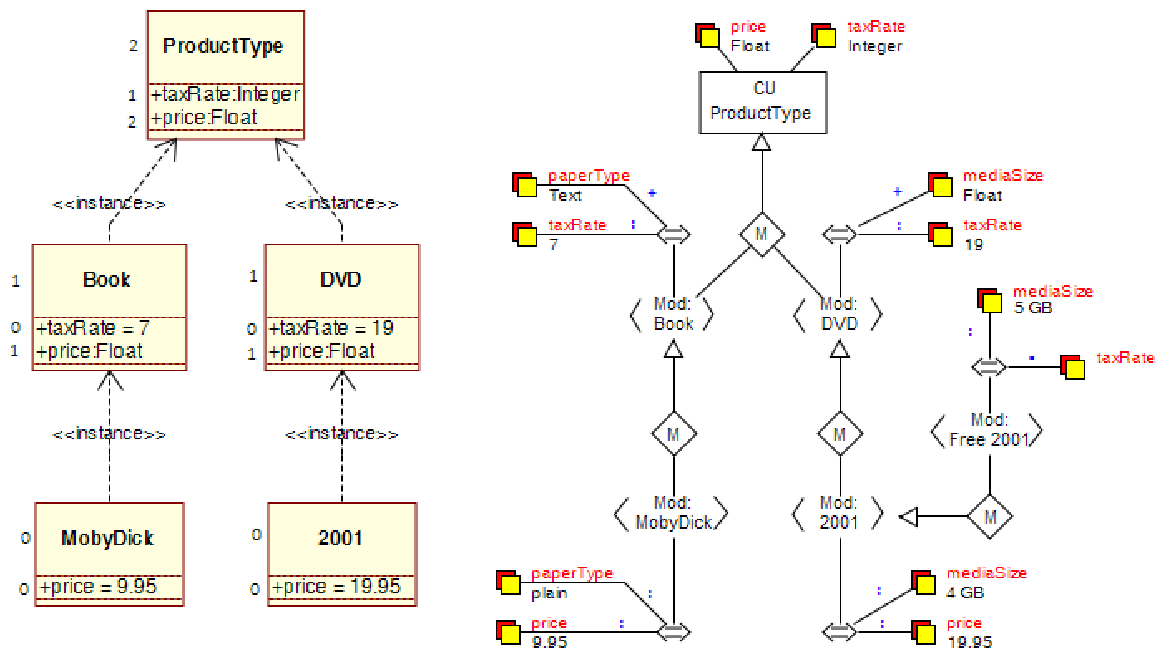

Fig. 6: UML deep instantiation vs. DVDocLang modifiers

A template command is given in Line (1). It is of the form: <TEMPLATE>= modifiers: m1(defm1), m2(defm2),..., where m1, m2,... are names of modifiers, while defm1, defm2,... are their definitions. By means of command <C1>=... in Line (2), a new language concept, i.e. type is defined in an arbitrary state of the document, even if the rendering has already started. To construct a new type, a list of pairs of the form *property*: *default value*: (p1: vp1, p2: vp2,...), needs to be defined. An instantiation of the object type is shown in lines (3), (4), (9), and (11). Object instances are uniquely identified by their types and positions within a sequence of commands. Alternatively, they may be identified by using an alternative key of the form <CU.ID:altKey> value, as it is presented in Line (9). Object values may be simple or composite. Object instances may be further extended by defining additional properties, as it is given in Line (4). A designer may apply inheritance from: (i) a type, as it is presented in Line (5); (ii) an instance, as it is presented in Line (6); and (iii) a document in PDF or HTML format using the file or identifier of the binary version in the repository, as it is presented in Lines (7) and (8). Inheritance allows for adding a new set of properties. At the same time, some of the properties may be declared as inapplicable. Examples are NO:p1 and NO:p3, given in Lines (6) and (10). For each type declaration, no matter whether it is a base or a subtype, a designer may restrict the application of modifiers, i.e. document variations, by defining a list of allowed modifiers, by means of ALLOWED_MODIF property. An example is given in Line (10). In the same way, further propagation of properties to subtypes may also be restricted. □

DVDocLang provides the concept of deep instantiation by means of its incremental specification. The incremental specification is fully supported by the dynamic construction of types, as it is presented in Lines (2), (5-8) and (10) of Example 11. In contrast to the concept of deep instantiation supported by DeepJava [21, 37], DVDocLang even does not impose the specification of a level to which attributes can be propagated. Herewith, it is allowed to apply refined DSL models dynamically, onto document instances whose lifecycle has already begun, where some CUs are already rendered.

In the following text we illustrate an application of multi-level modeling principles in DVDocLang that are based on the use of modifiers. We give a presentation of multi-level modeling through a comparison of the approaches based on an UML extension for deep instantiations and DVDocLang. To do that, in Fig. 6 we present to models. The first one, placed in the left hand side of Fig. 6, is a slightly modified example taken from [37]. It is an example of deep instantiation based on a UML extension. The numbers given in the left hand side of each class, next to each class name and each of its attributes, represent potencies. The notion of potency is used to specify the level to which the instantiation is allowed.

To illustrate our approach to the same model, we recall Example 7 and Figure 5, in which modifiers were applied to model untypical instances with a minimal number of templates. There, as well as in other previously presented abstract models, we used subtype relations with the S symbol put inside diamonds. In the abstract model given in the right hand side of Fig. 6, a new language concept, named *modification* is used. A modification is represented in the diagram placed at the right-hand side of Fig. 6 by diamonds with the M symbol inside. Such kind of relationship combines an inheritance with instantiation. Semantics of a modification is expressed at the level of the subtype or instance of an object, following the roles of properties in the modifiers. Modifiers are represented in the diagram by the ⇔ symbol. The roles may be: *instance*, denoted with the (:) symbol; *new attribute*, denoted with the (+) symbol; or *inapplicable property*, denoted with the (-) symbol. The first two roles are intuitively clear as they are available with the same meaning in the majority of contemporary modeling languages. The inapplicable property role is introduced as a counterpart of the potency concept from the UML extension. However, it does not require predefining the allowed depth of the instantiation and does not restrict any attribute to be applicable again to some subtype or instance.

A 1:1 mapping of a model to the resulting DSL script is established by the M2T transformation from Example 2. By this, we have a possibility to create an unambiguous reverse transformation of the code to the model.

As an example, for the DVD segment of the model from Fig. 6, a variant of the resulting DSL script may be:

```
(1) <ProductType>=price:Float;taxRate:Integer
(2) <DVD.FROM:ProductType;mediaSize:Float>taxRate:19
(3) <2001.FROM:DVD>price:19.95; mediaSize:4 GB
(4) <Free 2001.FROM:2001; -taxRate>mediaSize:5 GB
(5) <DVD.mediaSize>price:19.95; mediaSize:3 GB
(6) <2001>price:19.95; .mediaSize:3 GB
```

The first, a base type `ProductType` is declared by Line (1). Then, a `DVD` class/instance modifier is defined in Line (2). It introduces a new attribute `mediaSize`. From `DVD`, a new 2001 modifier is derived by Line (3). The values of the `price` and `mediaSize` are also given. `Free 2001` is a modifier given in Line (4). It removes the `taxRate` attribute, since the DVD is free of charge because of 1GB of advertisements, added to its main content. In Lines (5) and (6), two instance constructors of the 2001 type are given.

A selection which of available approaches (illustrated by the previous examples and Figures 3, 4, 5, and 6) is to be applied in a given scenario is not just a matter of free designer's decision. In practice, it depends on the level of current knowledge about the document types and specifics of the document production process. In an initial stage of this process, template models are usually created by use of simple properties. Latter, as the process gets more matured, template models evolve by including named modifiers, reducing the number of necessary templates and specifying relationships between modifiers and business activities. A possibility to formally specify relationships between CUs, templates and business activities by means of some DSL allow for a complete and clearly understandable specification of the dynamic characteristics of a system being modeled [18].

In the document production process, we may expect that modeling DSLs also evolve as their template models. There is also an emergent requirement to provide various forms of the concrete syntax of the same DSL made for different categories of users. In Section 6 we discuss in more details the process of refinement of a modeling DSL and template models.

6. A Refinement of Templates and DSLs

In the document production management, as well as in the model-driven software development, one of the common issues is a feedback that enables refinement of template models from document instances and DSLs from template models. In DVDoc Approach, we provide automated refinement of templates and DSLs based on the following assumptions: (i) a DSM tool is used to provide the document instance production, where each instance (produced either in PDF or HTML format) contains meta-data including specifications of both meta-model and model; (ii) documents are modeled using the DVDocLang coupled with the appropriate application providing variants of DSL scripts. All of this makes the document production process faster; and (iii) semantics of modifiers, particularly the named ones, can easily be redefined. It makes possible to keep a compatibility with previous models, while DSL is changed.

A notion of DSL script or Logical script (LS) is used here to denote a document or template model as a textual DSL specification. It is also a semantic equivalent of a Platform Independent Model (PIM) specification created by means of a graphical DSL. Therefore, in our approach, PIMs are also specified by DSL scripts. They are transformed into Platform Specific Models (PSMs) by means of DVDocRender in two ways. The first one results in an implementation script (IS) for an idealized interpreter or renderer denoted as ISI, while the second one results in an implementation script for a concrete interpreter, denoted as ISC. The main reason why a user identifies some instance as untypical is in inability of its modeling at the level of a LS. However, other reasons may also be related to inappropriate transformations of the LS to an ISI or the ISI to an ISC.

Here we present a template refinement within a context of document production process. Template refinement is a process of transforming one model to some other, which allows for simpler and more precise specification of a larger class of instances. The refinement of templates as document type models is often combined with the refinement of: (i) the modeling language; (ii) existing document instances; (iii) code generators (reports); (iv) target interpreter; or (v) modeling applications.

A generic refinement model is outlined in Fig. 7. In single document production users (User1, …, UserN) produce LSs, denoted as <LS>. They do this by using different client applications, such as DVDocPainter, Structured-text Editor or even a simple text editor. The applications are template driven. Template models are stored at the Template Server. By this, a time needed to obtain templates and their instances so as to merge them with document instance specific data is significantly reduced [13]. The target interpreter DVDocRender merges a script specifying a document instance with a template instance. This process is called script merging. After that, DVDocRender generates documents. This process is called document rendering. By this, DVDocRender produces as its outputs: figures, PDF or HTML documents, implementation scripts in both variants – ISIs and ISCs. Under specific circumstances, it is also possible to generate applications. An IS comprises all implementation properties of a document, as well as the relationships to the corresponding LS, and further to its original model expressed by means of the graphical DSL.

A relationship between a LS and the generated IS is similar to the relationship between an abstract model and the generated code. In order to provide inverse transformations from concrete to abstract models (T2M), we provide in the direct M2T transformation that all IS elements will keep a relationship to their origin in LS. Therefore, we provide bidirectional and unambiguous transformations between LSs and ISs. They are performed by means of the same algorithm. A feedback from the generated code to the model is provided by:

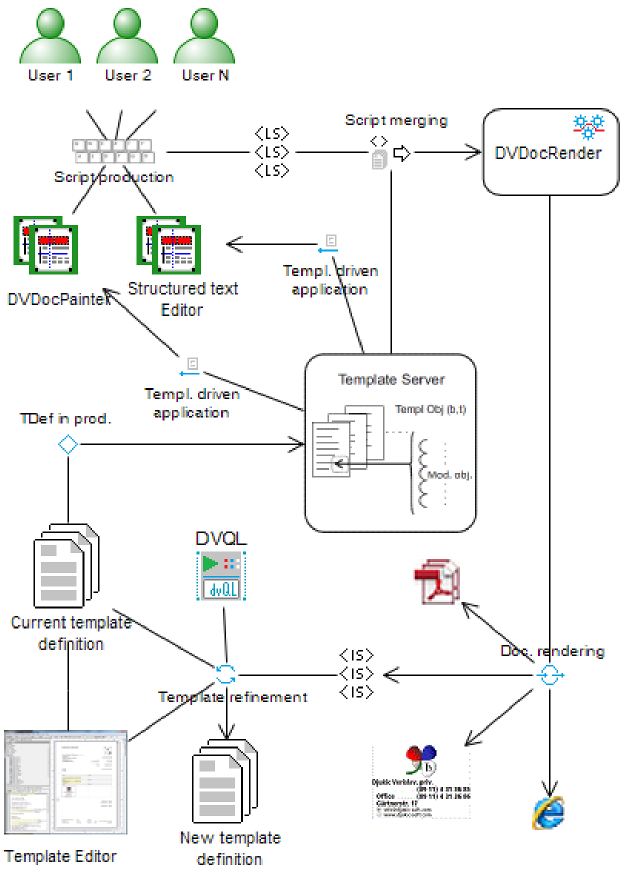

Fig. 7: A generic refinement model

- Action reports [31], i.e. transformations that are similar to MERL reports;
- Introducing the concept of a state in a language for M2T transformations [20, 25, 26]. States are used for synchronization of target interpreter activities, operations on models, and model execution steps; and
- Using a domain-specific query language which provides executing queries over logical and implementation document properties existing in models and generated code.

In DVDoc Approach, the DVQL [19] is used to perform analyses of ISs and LSs. Such analyses are aimed at identifying modifications, their classification by document types and users being applied them, as well as their breaking into atomic properties for the purpose of their re-grouping.

In general, an analysis of a LS may be specified by the two functions. The first one tackles templates, while the second one the LSs of already produced documents, as it is illustrated by the following example:

```
temDef' = f1(IS,LS,modifiers,temDef,dsl) and
LS'     = f2(IS,LS,modifiers,temDef,dsl).
```

Our intention is not to explain here the specifications of the functions `f1` and `f2` in detail, since they are too complex. In the example, `f1` and `f2` are used to retrieve temDef' and LS' scripts, merge them and, by repeated rendering, generate the implementation script that preserves the document layout from any changes. At the same time, they also provide a better use of existing DSL concepts. In this way, temDef' and LS' are simplified and expressed by means of more precisely defined domain-specific phrases. In this form, they are used for the refinement of template models, mostly through the approach reaching the minimal number of templates, as it is illustrated by the model in Fig. 5. Finally, the new template is declared as the current one and stored at the template server.

For the purpose of faster script merging, i.e. merging of document instances with template instances, the template server buffers both template and named modifier instances in the working memory. By our practical experience and performed measurements, it reduces the time needed for rendering up to 30% [13]. In this way, the buffering provides faster access to those document types and their variations expressed by named modifiers that are predominantly produced in a given period of time.

7. DVQL and Document Analyses

In this section we outline our document specific query language DVQL. It is a declarative modeling language providing queries and manipulation over meta-models, models and instances, i.e. the appropriate textual representations of types, modifications and document instances. By this, it is a language scoped not only with PIMs, but also with PSMs. Detailed specification of DVQL with the appropriate examples is given in [19].

By incorporating transformation formulas into the DVQL definition, an easier construction of test cases for model checking would be provided. A construction of transformations, i.e. generators becomes just a programming activity by the use of DVQL. For the purposes of our approach, however, the use of DVQL as a language embedded into MERL or DVDocRepLang is much more important, because it enables full automation of refinement process, as depicted in Fig. 7.

All documents rendered by DVDocRender are given in penta-format [5, 7, 9], by means of content, structure, layout, behavior and meta-data are defined. DVQL commands are most often executed over larger sets of LSs, which we observe as one script and call it a batch script. Operations executed by DVQL may change specifications given as LSs, ISIs, ISCs, PDFs or HTMLs. Apart from this purpose, DVQL is used to specify constraints or implement complex user services. DVQL consists of the following four units:

- Common Command Language (DVQL-C);
- Document Template Language (DVQL-T);
- Logical Script Language (DVQL-L); and
- Document Language (DVQL-D).

The syntax of DVQL is created so as to mimic the SQL syntax. Besides, it enables queries over both relational and hierarchical structures.

We also introduced the following special data types into DVQL: TemplDef – a string representing a template definition; BookDef – a list of strings representing a definition of a book, i.e. set of templates; InpScr – a string representing a LS; BatchScr – a list of strings representing a collection of LSs; ImpScr – a string or an XML structure representing an IS; and DocScr – a list of strings or an XML structure representing a collection of ISs.

As most of the other programming languages, DVQL has also standard operators and embedded functions. It enables referencing to all object types that are the instances of DVDocLang concepts, such as: LS, Book, Template, CU type definition, command, command structure, IS, item, global attribute, element attribute (i.e. property), modifier, logical page, implementation page, document state, etc. DVQL provides variety of specific operators. Some of them are division, union, intersection and difference of scripts, books and document collections. In the following example, we illustrate the use of DVQL on a LS.

Example 12: Here we present a LS that specifies the header of the "Offer" document type from Figures 2 and 8. The LS contains named modifiers for colors and fonts: `s4`, `f8`, `s1`, `s17`, `s4`, `s13`, as well as unnamed modifiers, i.e. simple properties, for the element positions and text alignments: `POS` and `AL`:

```
<TI.s4.f8.POS:XY(20;15)>DVDocGen Framework
<PZ.s1.POS:XY(20;41)>Djukic Verislav,
Gärtnerstr. 17, D-90408 Nürnberg
<AD.s17.POS:XY(20;47)>Company
Djukic Software Solutions

Gärtnerstr. 17
D-90408 Nürnberg
<SX.s4.AL:left,POS:XY(20;91)>Offer
<TX.AL:right,POS:XY(193;91)>Nürnberg, ...
<TX.s13.AL:left,POS:XY(20;98)>Nr. 434
```

The following query presents all element modifiers (EM) that are the first with their position in the list (M.1), for all types of CUs, from all LSs that belong to the current batch script (BSCR):

```
SELECT BSCR.<*>.EM.<*>.<*.*.M.1>
```

The elements of the LS given above that are referred by this query are bolded. A short form of the result obtained by the query execution is:

```
s4.s1.s17.s4 'AL:right,POS:XY(193;91)'.s13
```

while the basic form of one n-tuple is:

```
('em','B','T','1','B','T','1','1','<TI>','1
','s4','DVDocGen Framework')  □
```

Each element of the query result set is an n-tuple that contains: an identifier of the CU type, the name of the book, the name of the template, the ordinal number of the LS, a reference to the other book and template related by the CU, the page number, the ordinal number of the element in LS, the ordinal number of the element of the same type, the modifier and the value. These n-tuples, no matter whether their origin is a LS or IS, can be transformed into LS or IS without loss of information. These bidirectional transformations are used for the refinement of templates and already existing document instances.

A frequent use of the same modifiers, as it was the case in Example 12 with the named modifiers for colors and fonts: `s4`, `f8`, `s1`, `s17`, `s4`, `s13`, leads to a solution in which the default values should be assigned to the logical fonts for the appropriate CUs, in the base template definition. The same holds for unnamed modifications for the position and justification, expressed by `POS` and `AL`. However, if the modifiers are not to be applied to all document instances, then a subtype should be introduced by using some of the approaches presented in Section 5. By this, such document instance is produced by means of a semantically equivalent script of the form:

```
<TI[.mod1]>DVDocGen Framework
<PZ[.mod2]>Djukic Verislav, Gärtnerstr. 17,
D-90408 Nürnberg
...
```

Semantics of modifications is specified through a model by means of the DSM tool. For the first line of the LS from Example 12 the equivalence: `mod1` ⇔ `s4.f8.POS:XY(20;15)` holds. Such semantic relationships together with meta-data embedded into each document instance, provide the refinement of previously generated document instance without loss of information. A further analysis of the LS from Example 12 may lead to the conclusion that in some cases `mod1` is to be divided into `mod1_1` and `mod1_2`.

One of such cases is when `mod1_2` is applicable on `<PZ[.mod2]>`.

8. A Tool for Creating Abstract Template Models

In this section we outline just main characteristics of our DSM application in document engineering, for creating formal template specifications.

To adequately support a template modeling activity, we have developed a specialized graphical editor for drawing templates. It is named DVDocEditor. A screenshot of its main window is presented in Fig. 8. To support generating code from the abstract templates, we have developed DVDocRepLang [28, 20] and coupled it with DVDoc Integrated Development Environment (DVDocIDE).

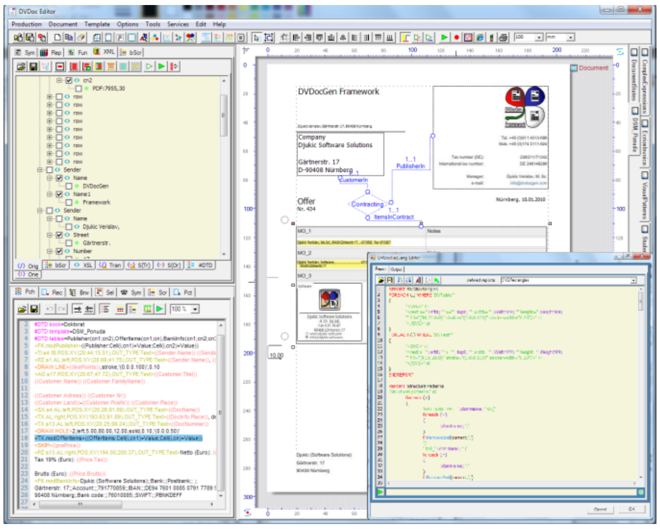

Fig. 8: DVDocIDE with DVDocEditor and code generators

General-purpose editors of the kind are mostly oriented to defining document layout styles. However, DVDocEditor allows for specification of CU semantics. Furthermore, it provides the concepts of a role, port, relationship and property. It also provides more than fifteen groups of layout controls that are used to specify text layouts, tables, figures, bar-codes, etc. Apart from explicitly defined properties, DVDocEditor provide an access to all properties supported by the .NET framework controls. In this way, we provide generating graphical components of applications as in [22].

By DVDocEditor, CUs may be related to particular activities in a document lifecycle. This allows for basic modeling of dynamic system characteristics and document lifecycle [18].

The following advances of DVDocEditor are the consequences of providing a code generator and a language for the code generation into the DVDocEditor template editor:

- A possibility to define various patterns for the document structure validation;
- A possibility of generating Web or Windows applications;
- A simple definition of new domain-specific phrases, i.e. modifiers;
- A generation of parts of template specifications that are not supported by graphical editor;
- A management of template versions at a higher level of abstraction;
- A possibility to create various and significantly different templates from one model diagram; and
- A systematic template refinement.

A full formal specification of DVDocRepLang and examples of defined generators may be found in [20].

9. Conclusion

An application of DSM in the Document Engineering, which includes the development of DSLs and tools for modeling documents and their templates, may significantly contribute to resolving the problem of producing untypical document instances. The practical benefits of such approach are wide and involve deploying of a consistent development methodology, languages and tools for document modeling, and particularly for specification of model variations. The main practical advantages of the application of our DVDoc Approach in the document production process are the following:

- By using DSL script as a domain-specific instead of a general purpose specification, we raise the efficiency of the creation and refinement of template definitions and, consequently the document production process in general;
- At the level of a code generator, we provide simple transformations of document templates as abstract specifications into arbitrary target specifications. Transformations into XSL styles are just one of the examples; and
- Semantic based template editors are much more powerful modeling tools, than the editors aimed at defining layout styles or topological relationships only.

Our further research work is aimed towards automated generation of semantic template editors for different application domains. One of the research directions is enabling incremental specifications and incremental generators to be used in modeling dynamic characteristics of a system by means of executable software models. By this, a better support to modeling and execution of complex business activities is provided. Also, we plan to improve DVDoc Approach to the level of a full application of multi-level modeling principles, so as to overcome shortcomings of current environments, based mostly on XSL-FO language.

10. Acknowledgement

The authors would like to kindly thank Juha-Pekka Tolvanen from the University of Jyväskylä for his valuable support and proof reading.

11. References

[1] Steven Kelly, Juha-Pekka Tolvanen, "Domain-Specific Modeling: Enabling Full Code Generation", ISBN: 978-0-470-03666-2, March 2008, Wiley-IEEE Computer Society Press.

[2] Robert J. Glushko, Tim Mc Grath, "Document Engineering", MIT Press 2008.

[3] PdfLib for creating PDFs, http://www.pdflib.com/
AbcPdf library for creating PDFs, http://www.websupergoo.com

[4] MetaEdit+ Workbench, "Workbench User's Guide" http://www.metacase.com/support/45/manuals/mwb/Mw.html

[5] Di Iorio, A. "Pattern-based Segmentation of Digital Documents: Model and Implementation, Ph.D. Thesis", UBLCS-2007-05, Department of Computer Science, University of Bologna, 2007.

[6] Verislav Djukic, "DVDocLang Language Reference", www.dvdocgen.com/Framework/DVDocLang.pdf

[7] Antonina Dattolo, Angelo Di Iorio, Silvia Duca, Antonio A. Feliziani, Fabio Vitali, "Structural patterns for descriptive documents", Proceedings of the 7th international conference on Web engineering, Italy, Lecture Notes In Computer Science, 2007

[8] Ivan Lukovic, Verislav Djukic, "DVDocLang vs. XSL-FO", www.dvdocgen.com/Framework/DVDocLang_XSL-FO.pdf

[9] Angelo Di Iorio, Luca Furini, Fabio Vitali, "Higher-level Layout through Topological Abstraction", ACM DocEng 2008

[10] Apache Software Foundation: "FOP", http://xmlgraphics.apache.org/fop/0.95/index.html

[11] Microsoft Extensible Application Markup Language (XAML) http://xml.coverpages.org/ms-xaml.html

[12] User Interface Markup Language (UIML) http://www.uiml.org/

[13] Verislav Djukic, "DVDoc Renderer Benchmak", http://www.dvdocgen.com/Framework/DVDocRenderBench.pdf

[14] Kosar T., Oliveira N., Mernik M., Pereira M. J. V., Črepinšek M., Cruz D., Henriques P. R., "Comparing General-Purpose and Domain-Specific Languages: An Empirical Study", Computer Science and Information Systems (ComSIS), ISSN: 1820-0214, Vol. 7, No. 2, May 2010, pp 247-264.

[15] Verislav Djukic, "DVDocGen Framework, application interface", http://www.dvdocgen.com/Framework/DVDocFramework.pdf

[16] OMG Model Driven Architecture, http://www.omg.org/mda/

[17] Exstensible Stylesheet Language, Formatting Objects (XSL-FO), Reference Manual, http://www.w3.org/TR/xsl/.

[18] Verislav Djukić, Ivan Luković, Aleksandar Popović, "Domain-Specific Modeling in Document Engineering", Proceedings of the Federated Conference on Computer Science and Information Systems, Poland, 2011

[19] Ivan Lukovic, Verislav Djukic, "DVQL Language Specification", www.dvdocgen.com/Framework/DVQL.pdf
www.dvdocgen.com/Framework/DVQLDemo.wmv, video

[20] Verislav Djukić, Aleksandar Popović, "DVDocRepLang grammar specification", www.dvdocgen.com/Framework/DVDocRepLang.pdf

[21] Colin Atkinson, Thomas Kühne, "The Essence of Multilevel Metamodeling", Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, Springer-Verlag London

[22] Tony Clark, Andy Evans, Stuart Kent, "Aspect-oriented Metamodelling", The Computer Journal, 46 (5), 2003, pp 566-577.

[23] Olivier Beaudoux, Arnaud Blouin, Jean-Marc Jézéquel, "Using Model Driven Engineering technologies for building authoring applications", ACM Symposium on Document Engineering 2010: 279-282

[24] Ber hard Rumpe, Robert France, "On the relationship between modeling and programming languages", Editorial for the SoSyM Issue 2012/01: Part 1.

[25] Benjamin Klatt, "A Closer Look at the model2text Transformation Language", http://wiki.eclipse.org/Model2Text_using_ Xpand_and_QVT_for_Query

[26] Verislav Djukić, DVDocRepLang demo, video http://www.dvdocgen.com/Framework/ModelTransformation.wmv

[27] Verislav Djukić, DVDocFlowLang demo , video http://www.dvdocgen.com/Framework/DVDocFlow.wmv

[28] Verislav Djukić, Using DVDocIDE , video http://www.dvdocgen.com/Framework/UsingDVDocIDE.wmv

[29] Verislav Djukić, Using MetaEdit+ from DVDocIDE , video http://www.dvdocgen.com/Framework/DVDocIDEMetaEditCtrl.wmv

[30] Verislav Djukić, Marko Bošković, Aleksandar Popović, Ivan Luković "Using Domain-Specific Modeling for Integration of Heterogeneous Business Activities", Internal Report, 2012, http://www.dvdocgen.com/Framework/IntegrateHeterBA.pdf

[31] Verislav Djukić, Marko Bošković, Aleksandar Popović, Ivan Luković "Using Action Reports for Testing Meta-models, Models, Generators and Target Interpreter in Domain-Specific Modeling", Internal Report, http://www.dvdocgen.com/Framework/ActionReports.pdf

[32] Tolvanen, J-P., Kelly, S., "Integrating Models with Domain-Specific Modeling Languages". Procs of 10th Workshop on DSM, Reno, Nevada, USA, Helsinki Business School, 2010.

[33] Atzmon Hen-Tov, David H. Lorenz, Assaf Pinhasi, Lior Schachter: "ModelTalk: When Everything Is a Domain-Specific Language". IEEE Software 26(4): 39-46 (2009)

[34] Dae-Kyoo Kim, Robert B. France, Sudipto Ghosh: "A UML-based language for specifying domain-specific patterns". J. Vis. Lang. Comput. 15(3-4): 265-289 (2004)

[35] Common Variability Language (CVL), CVL 1.2 User Guide, http://www.omgwiki.org/variability/doku.php

[36] Christian Schaefer, Thomas Kuhn, Mario Trapp, "A Pattern-based Approach to DSL Development", SPLASH '11, Workshops on DSM'11, Proceeding.

[37] Thomas Kühne, Daniel Schreiber, "Can Programming be Liberated from the Two-Level Style? Multi-Level Programming with DeepJava", ACM SIGPLAN IOOPSLA'07.