A Declarative Approach to Cross-Domain Model Analysis

Dipl.-Ing. Ulrich Hartmann
Prof. Dr.-Ing. Petra von Both
Institute for Industrial Building Production, University of Karlsruhe, Germany

ABSTRACT: The analysis of digital models requires in-depth knowledge of the semantical, syntactical and technical specification of the model schema and its internal representation in the computer. Taking this huge overhead into account, the preparation steps for conducting an analysis onto a digital model can get more dominant than the analysis itself. Beyond that, people involved in providing the technical infrastructure and people conducting the analysis often do not share the same skill set. Productivity gains could be expected, if one side could concentrate on the conceptual-part of the analysis and the other part could focus into the implementation-part of digital model analysis. The concept shown here enables analysts to express elements and terms of an analysis in their own domain-specific language, while the underlying mapping between the conceptual and the technical view onto the system can be set up in a generalized and extensible fashion using state-of-the-art software concepts.

1 INTRODUCTION

1.1 *Motivation*

As stated above, getting data out of a digital model can soon get very complex. But complexity -if well documented- is not a first place issue. Access to structures and values of a model instance often requires internal knowledge about commonly used good practices on how to use data structures if alternatives exist. Although publicly available the model schema is sometimes not enough for working efficiently and unambiguously with model data.
Several commonly used model types (e.g. IFC) have their primary focus on product description, which is well suited for data exchange; while data analysis is often hindered by the way data is structured. For example IFC connects properties with entities through a plethora of at least four indirections before the requested value is finally in sight. While this loose coupling is a very elegant and flexible mechanism for assigning properties to elements, it is quite impractical for inquiries on the data set. This is independent of the data persistence technology used. Either relational or object-oriented modeling will have to resolve the deep indirection trees, making either table joins or equivalent OO mechanisms necessary. Depending on the underlying problem, this might either be solved by converting the logical model structure into a technical database structure in order to optimize data access (performance, complexity) or leads into complex query expressions. The price for the first solution is often conversion and redundancy management. Moreover, applications build on top of the database structures rather than the native model structures will find it harder to implement domain-specific logic, which by definition is using the native domain-specific concepts. As for the second solution, complex queries might be hard to maintain and error prone, scaling down application robustness.

1.2 *Vision*

The internal digital representation of a domain-specific model is the result of applying methods of computer science to the concepts of the respective problem domain. Although reflecting the understanding of the domain concepts, syntax and semantics the digital model carries the image of computer science, not of the original problem domain. An approach of using this very valuable computer model as an underlying foundation while expressing analytic interactions in plain domain-specific language would strengthen the ability of domain-experts to use domain-models, without having to be a software specialist at the same time.
E.g. instead of going along the following indirection path to find the *floor area* value of all *office room*s in an IFC model

```
IfcSpace→
  IfcRelDefinesByProperty→
    IfcSpaceProgram→
      IfcPropertySingleValue→
        IfcLabel:OccupancyType == “office”
        …
```

As an analyst you’d rather like to say: “total office room floor area” meaning the summed up floor area of all rooms of type ‘office’ on the respective floor in the respective building. Instead of expressing **how** to collect the values by using the syntax of the digital model, a declarative expression doesn’t concern about the ‘how to’. Instead it focuses on the ‘**what**’.

How can this be achieved? This article tries to lay out an approach, conceptually and technically, how the separation of concerns between domain experts and technical experts can be encapsulated, resulting into better usability and software and the re-use of analysis concepts and technical components alike. The following example should show the principle.

1.3 *Benefits*

Separating domain concepts from technical concepts not only encourages the utilization of digital models by domain experts. The capability of defining problem-centered namespaces could even support cross-domain collaboration of analysts of different domains. A view exposing familiar structures and concepts can be provided even if the analysis belongs to more than one domain. The mapping layer between expert view and technical view must be flexible enough to reflect changing requirements from the expert level. This also leads to more robust and reusable applications. In the end, complexity of digital models need not be challenging, as it comes along packed within suitable interfaces.

2 EXAMPLE SCENARIO

The local administration of *X* city is planning a new bus line to enhance the public transportation system. The route should link business areas and living areas by providing an alternative option to travelling by car for reducing office-hour traffic significantly.

Data Sources
The communal cadastral system, often basically a GIS system, contains the city map with real estate and city road information. Each real estate entry has a reference to an electronic building document containing an IFC-based model of the building[1]. The technical specification is of no relevance for the concept.

Algorithms for Analysis
Different route alternatives have to be compared. The catchment area of the optimal route would collect most commuters in the morning and let them disembark as close as possible at their working location and vice versa in the evening. Although it's not possible to link everybody's living location with the individuals working location, it seems statistically sound enough to optimize the traffic line by finding the best coverage of the related embarking and disembarking areas. The floor area of all buildings in the covered region is summed up, separated into the different occupancies like type 'office', type 'private', etc. In this simple example we leave other considerations like route length, time of travelling, line switching and so on out of scope. The intention of this example is not to build up a sophisticated analysis model for urban traffic optimization. The focus is on domain-spanning analysis and the advantages of putting an abstraction layer between the software-centered and the domain-knowledge-centered view. We could apply nearly the same algorithms for placing block heat and power plants at optimal locations in the urban area and –even more important- separation of concerns into two separate physical layers would promote the re-use not only of concepts but also of components.

Basic Assumptions
- Coverage area is calculated assuming a maximum walking distance to the route. Beyond that distance it wouldn't be attractive to utilize the bus line.
- An estimation of the amount of people involved is given by the person ratio per square meter office floor and per square meter private home floor respectively.

Involving different Algorithms
As a first rough estimation, the coverage area could be calculated by applying a direct surface to surface connection between building and bus route. Buildings within the maximum distance belong to the coverage area.
In a more meaningful (but also more time consuming) calculation the exact walking distance between building location and bus route could be calculated by called a navigation service (e.g. Google maps).
Pursuing the concept of dynamic business logic involvement through the use components loaded at runtime, different algorithms for calculation could be consulted declaratively.

3 DECLARATIVE ANALYSIS ENVIRONMENT

3.1 *Design Principles and Requirements*

A systematic solution has to meet the following requirements.

- Separation of conceptual and technical level
- Extensibility on both levels
- Configurable domain-specific language support
- Independence of underlying domain model
- Independence of underlying data model
- Works with OO and relational paradigm alike
- Use of mainstream software technology

[1] admittedly quite an optimistic assumption by now, making it necessary to have an alternative instrument if this is not given

– More than one model type simultaneously (cross-domain)[2]

3.2 *System Architecture Concepts*

Separating the domain-specific language (DSL) layer from the data technology layer requires an abstraction layer between them. The language used in the DSL-Layer could be considered as an intermediate language that will be subject of further interpretation. The first step of building an interpreter is to define the language to interpret. This language will be defined in the EBNF (Extended Backus-Naur form[3]) notation, which is itself a language to describe other languages. This must be done by domain experts eventually supported by computer science experts. It includes the definition of operators, keywords and functions used in the domain-specific language, e.g.

Rooms→Occupancy(“office”)→FloorArea

or alternatively

FloorArea OFALL Room WITH occupancy “office”

This virtually exposes a room element with occupancy and a floor area attribute to the user. The selection of keywords and operators is entirely up to the domain expert. In the first example above the ‘→’ is used as dereferencing operator which would evaluate internally to a query for room elements with their occupancy attribute set to “office”. The second example uses a different syntax expressing identical semantics. Independent of the DSL design, after interpretation both expressions would return the same list of values of the floor area of each room. In this simple example the DSL consists of the operators OFALL and WITH, the keywords FloorArea, Room and occupancy and the parameter “office”. Obviously the originator of the DSL is absolutely free in choosing the names for operators and keywords.
DSLs have to be interpreted by an interpreter[4]. An interpreter is a program that is able to dynamically execute commands written in a defined language, in this case the DSL. E.g. a mathematical expression evaluator should enable dynamic input and calculation of formulas, whereas a model analysis interpreter should be able to evaluate terms, characterizing concepts and properties of the given model domain.
It must be able to maintain the returned sets of objects for further processing. A DSL has to be unambiguous, but it is by intention not Turing-complete in the sense of general purpose languages (GPL).
The creation of an interpreter can be largely supported by tools. As known, lexical analysis is the process of converting a sequence of characters into a sequence of tokens. Programs performing lexical analysis are called lexical analyzers or lexers. Tokens have a meaning given by a grammar (data type, keywords, functions, etc.). A parser can then create an abstract syntax tree (AST), representing the hierarchy among the tokens. The AST aligns with the syntax defined for the DSL.
In order to separate the evaluation of the syntax tree from the model logic itself, we take advantage of the *visitor pattern* (Gamma et al. 1995).
Visitors can be implemented separately in a modular way, providing a means of adding domain-specific evaluation logic (semantics) by configuration,

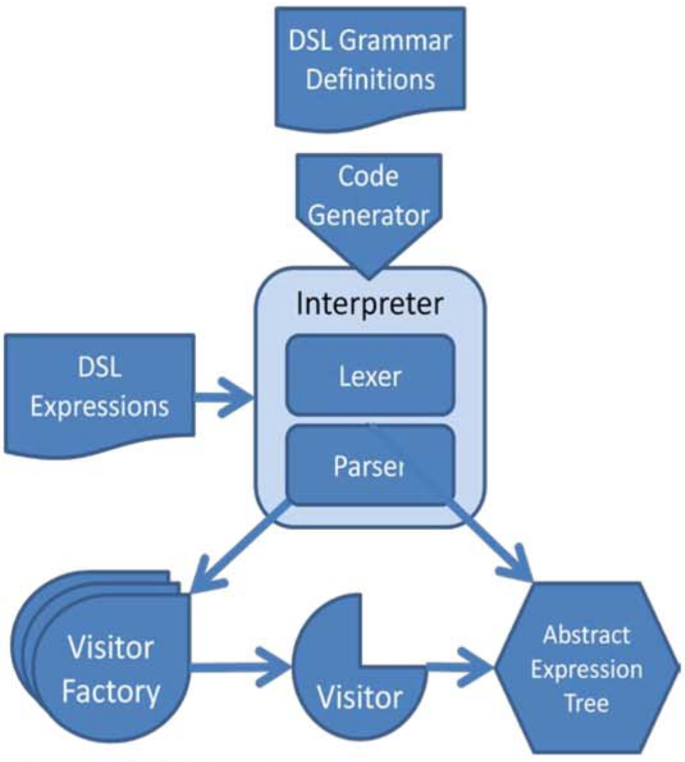

Figure 1. DSL Interpreter

according to analytic requirements. Several tools exist for the semi-automatic creation of lexers and parsers. They create program code from EBNF notations. A closer description of how to create an interpreter is beyond the scope of this paper.

[2] Avoiding the term ‚multimodeling here which is discussed extensively in [Hess09] and others (see References section)
[3] ISO Standard 14977
[4] or transferred into machine by a compiler

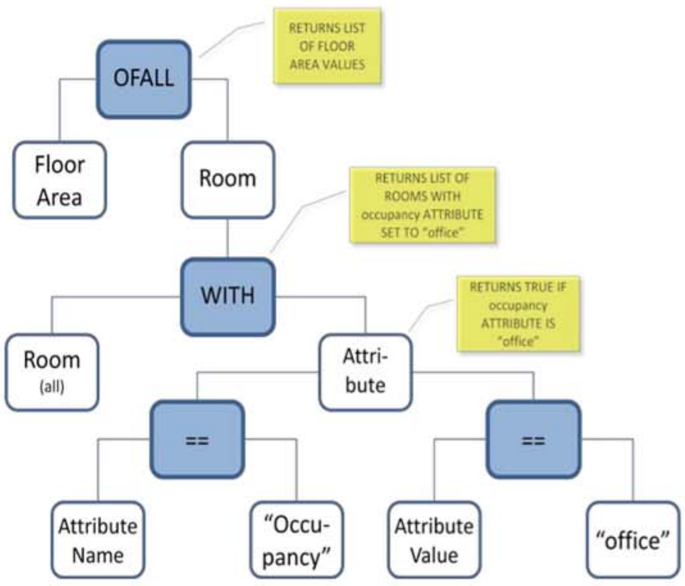
RETURNS LIST OF FLOOR AREA VALUES
OFALL
Floor Area
Room
RETURNS LIST OF ROOMS WITH occupancy ATTRIBUTE SET TO "office"
WITH
RETURNS TRUE IF occupancy ATTRIBUTE IS "office"
Room (all)
Attri-bute
==
==
Attribute Name
"Occu-pancy"
Attribute Value
"office"

Figure 2. AST instance of sample expression

Linking Syntax to Semantic

The main reason for using DSLs is to hide technical complexity and let the domain expert use his own language. The interpreter can traverse and "understand" analytic expressions given in the domain-specific language. The semantics is implemented inside the interpreter. Encountering a node in the AST, the interpreter knows with function to call and which set of data to provide as parameters. Important is, that there is a well-defined relation between naming conventions used in the AST and in the visitors. This is being assured by using the same EBNF notation for the lexer and the implementation of the visitor.
In the expression "a + b" everybody has an immediate idea about the meaning of the plus-sign. On a closer look, this certainty turns out to be unfounded. It depends on the data type of 'a' and 'b' which operations to be performed. E.g. a matrix operation "a + b" differs deeply from a simple addition of integer values. Some programming languages offer a mechanism for over-defining existing operators (e.g. operator overloading in C and C++), for re-using operators within different contexts. In our example, a visitor object (implementing the visitor pattern) will be developed for a specific context. This context is given by the problem domain represented by the underlying model schema (e.g. IFC). Therefore the DSL-term 'Rooms WITH occupancy "office" ' will evaluate always to the same syntax tree, but the visitor implementation evaluating the tree will be depending on the actual model type. E.g. a CityGML model will have another notion of building locations as an IFC model, therefore the algorithm calculating the distance between the building location and a given route (e.g. polyline) queries for different entities in the model and resolves links due to the characteristics of the specific model type.
In our case the DSL-Interpreter represents the language of a specific analysis; the visitor implementation encapsulates the access methods to the domain model. One DSL-Interpreter might therefore work together with different visitors, each representing the semantics of a concrete DSL.

Dynamic method binding

Two problems are still waiting for a solution here:

- How to dynamically associate domain-specific language syntax (AST) with the domain-model-specific semantics (visitor object)?
- How to add a visitor object dynamically at runtime?

Another design pattern comes to the rescue: the Abstract Factory Pattern (Gamma et al. 1995). In our case it will be a factory for visitor objects.

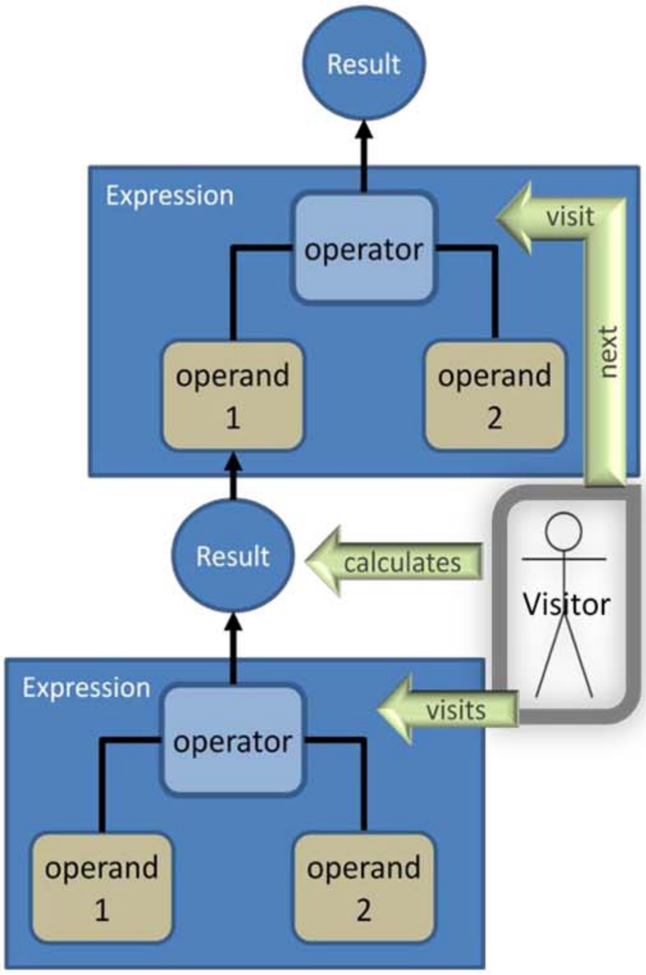

Figure 3. Visitor traversing AST

All we have to know about the visitor object to be loaded is the base class from which it has been derived (*VisitorBase*). The concrete type is (and will remain) unknown to the application invoking it.
According to (Gamma et al. 1995) a factory hides the concrete type of an object to be created against the application. The application can create and use the object without knowing its type. Our goal here is, to be able to add functionality by configuration. Therefore, the factory object itself needs a dynamic

mechanism of being able to instantiate new DSL-visitor class objects. For that, the factory object needs to know where to find the code (e.g. assembly), the type name of the new visitor class, and the internal name of the visitor class. The internal name is for keeping and internal list of available visitor classes.

All this information can be kept in an external configuration file being loaded at runtime by the factory object. The factory object can then dynamically load all referenced assemblies and instantiate the right visitor objects on demand (see C# code snippet below).

```
public static object CreateFactory(string virtualName)
{
   string path = "../../VisitorFactory.config";
   XmlDocument doc = new XmlDocument();
   doc.Load(path);
   XmlNode factoriesNode=
             doc.SelectSingleNode("//Factories");
   foreach (XmlNode factoryNode in
                                   factoriesNode.ChildNodes)
   {
      if (virtualName.Equals(
               factoryNode.Attributes["virtualName"].Value))
      {
         string assemblyAndTypeName =
                        factoryNode.Attributes["type"].Value;
         return CreateInstance(assemblyAndTypeName);
      }
   }
```

The internal list will also have an entry for the domain assignment of the respective visitor. This enables the interpreter to invoke the correct factory responsible for the creation of the visitor object of the domain requested.

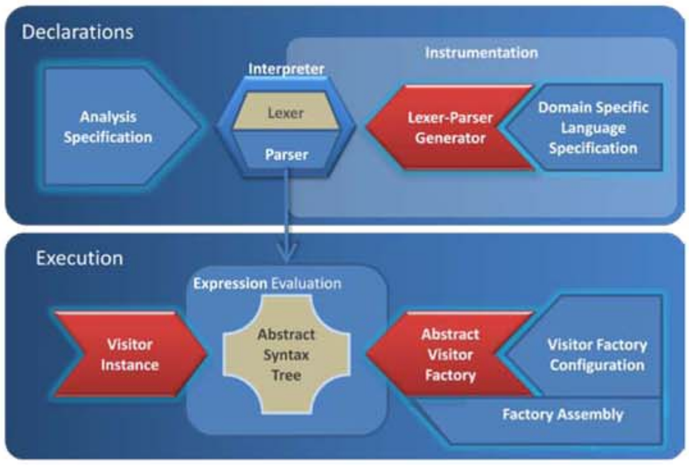

Figure 4. Analysis declarations, instrumentation and execution

For convenient processing the format of the configuration file is XML (see sample below)

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
   <Factories>
      <Factory virtualName="OccupancyAnalysis" type=
            "Factories. OccupancyFactory, AssemblyName1"
                        domaineName=”IFC”
                        subDomaine=”3.0”/>
      <Factory virtualName=" BusLineAnalysis " type=
         "Factories. BusLineFactory2, AssemblyName2"
                        domaineName=”CityGML”
                        subDomaine=”1.0”/>
   </Factories>
</configuration>
```

At runtime the parser can now invoke the right visitor object factory when encountering the fully qualified function name (e.g. domainName + ’.’ + functionName) in the AST. The factory will create the visitor object which will be used to evaluate the related node in the AST.

3.3 *Technologies and Tools*

ANTLR, ANother Tool for Language Recognition, is a language tool that provides a framework for constructing recognizers, interpreters, compilers and translators from grammatical descriptions containing actions in a variety of target languages. Generated code for lexers, parsers and the like can be used inside applications. ANTLR uses EBNF notations as input for code generation. It provides excellent support for tree construction, tree walking, translation, error recovery, and error reporting.

"Oslo" is the code name for the Microsoft's next generation application development platform. "Oslo" leverages domain-specific models, languages and tools. It promotes the idea of model-driven methodology and provides a toolset for the creation of DSLs. Two modeling languages MGrammar and MSchema and the data format MGraph have been developed. The extensible editor Intellipad supports the development of schemas and grammars.

The programming language C#[5] has a built-in support for conducting queries with the same formal query language on all data sources[6]. According to (Akehurst et al. 2008) ‘the recent offering from Microsoft of the “Orcas” version of Visual Studio with C# 3.0[7] and the LINQ library provides functionality almost identical to that of Object Constraint Language (OCL[8])’. Other than its ‘platform independence’ the major advantages of OCL over traditional Object Oriented programming languages has been the declarative nature of the language, its powerful navigation facility via the iteration operations, and the availability of tuples as a first class concept. LINQ / OCL is a powerful mechanism for the speci-

5 Standard ECMA-334 and ISO/IEC 23270
6 LINQ – Language INtegrated Query
7 C#4.0 release candidate available at time of this publication
8 OMG Standard

fication of an object set in a declarative way. Translation between declarative expressions of a DSL and the respective evaluation expression in the semantic layer can therefore be more in sync, avoiding the need for changing the paradigm.

Reflection-oriented programming[9] extends the object oriented paradigm by the ability of self-examination, self-modification, and self-replication. However, the emphasis of the reflection-oriented paradigm is dynamic program modification, which can be determined and executed at runtime. Reflection is used in the visitor factory for type validation of visitors loaded dynamically and for inference of data types by the interpreter.

4 APPLICATION

Now, as the puzzle begins taking shape, we can proceed with our bus line example on the fully equipped analysis environment. The interpreter is now able to run the following script:

modelKA =

CityGML.LOAD_MODEL "Karlsruhe"

In the interpreter the visitor for the CityGML domain will be instantiated, the method associated with the keyword LOAD_MODEL will be called with the parameter "Karlsruhe", the respective model will be instantiated. Finally a model instance will be assigned to the variable modelKA, the data type of the variable will be inferred for being of type CityGML. This, by intention is hidden from the domain user, who can fully concentrate on the his/her domain-centric view. Next the expression

route = LOAD_ROUTE "BUS_LINE_ALT_1"

will be processed. This will load the geometric description of the bus route and stores it in the variable 'route'.

In the next step we create objects for the subsequent analysis step. It could have been done completely inside a visitor object e.g. by using a joined LINQ query inside the visitor, but it is shown here explicitly for clarity. This will explicitly show a query on the data sources of the two domains involved. Using a loop over all city buildings we get the building location from the CityGML domain and the floor areas (office and private) from the IFC domain. Within the loop IFC models are being instantiated and queried for the respective floor area values.

ForEach (cityBuilding in modelKA)

Note that 'cityBuilding' is from the CityGML domain (inferred by the interpreter)

ifcBuilding = IFC.LOAD_MODEL cityBuilding

Instantiates the IFC building model referenced in the city model

officeFloor += ifcBuilding.OfficeFloor

Uses IFC domain visitor to dereference *OfficeFloor,* consecutive values summed up in the variable officeFloor

privateFloor += ifcBuilding.PrivateFloor

Uses an IFC domain visitor to dereference *PrivateFloor,* consecutive values summed up in the variable *privateFloor*

location = cityBuilding.Location

Uses CityGML domain visitor to dereference *Location* in the city model

buildingList.Add(cityBuilding.ID, location,

officeFloor, privateFloor)

Stores the attributes relevant for analysis in a list, one entry for each building

EndForEach

profile = EMBARKATION_PROFILE(buildingList)

Calculates embarkation profile using default visitor

STORE_PROFILE ("ALTERNATIVE_1", profile)

Stores profile

SHOW_PROFILE(profile)

Visualizes profile

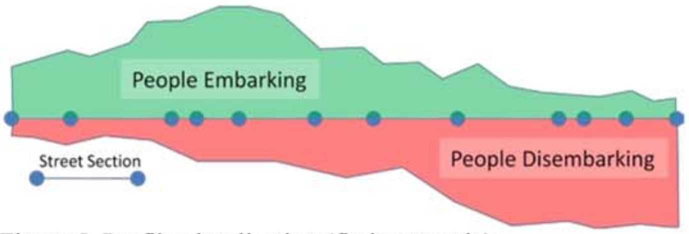

Figure 5. Profile visualization (fictive sample)

Needless to say, that alternative routes could easily be compared running the same script with different route data.
Refining results by exchanging the rough estimation algorithm for coverage area calculation could even be realized 'under to hood' by changing the visitor assignments in the xml configuration file, then running the scripts with critical data again.

5 CONCLUSION

Today much of the model logic is buried inside some developers' guideline handbook or just in the heads of a few nerds. The value of a model is rising with the capability to interact with domain experts in an efficient and less time consuming fashion.

[9] Paradigm driven by reflection is called *reflective programming*

Encapsulation of technical complexity by using domain-specific languages as an abstracting layer offers efficient handling of model interaction.

The degree of complexity exposed at DSL level can easily be scaled, due to the specific granularity needed for an analysis. Reusable components can implement model semantics and model access patters. They can serve as foundation for the semantic level as basis for putting different DSL-Layers on top.

Through this interface many applications could prosper from the ability of not just storing and exchanging data through digital models, but also getting specific pre-processed data back from the model on demand of the domain. As shown in this paper, complex domain-spanning analysis can be handled declaratively at domain user level by using and / or providing the right abstractions and toolset. Standard technology has evolved greatly to support domain-related requirements.

REFERENCES

[ANTLR] ANother Tool for Language Recognition, framework for constructing interpreters, compilers, etc. from grammatical descriptions. http://www.antlr.org/

Denton, T. & Jones, E. & Srinivasan, S. &Owens, K. &Buskens, R. 2008. NAOMI-An Experimental Platform for Multi-modeling. In Proceedings of MODELS, pages 143–157, Toulouse, France, October 2008.

Gamma, E. & Helm; Johnson, R. E. 1995. Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley

Akehurst, D.H. & Howells, W.G. & Scheidgen, M. & Mcdonald-Maier, K. D. 2007. Proceedings of the Workshop Ocl4All: Modeling Systems with OCL at MoDELS 2007, C# 3.0 makes OCL redundant!

Hessellund, A. & Lochmann, H. 2008. An Integrated View on Modeling with Multiple Domain-Specific Languages. IASTED submission

Hessellund, A. 2009. Domain-Specific Multimodeling, Ph.D. Dissertation (preprint), Supervisors: Peter Sestoft and Kasper Østerbye

Mernik, M. & Heering, J. & Sloane. A. M. 2003 When and how to develop domain-specific languages. Submitted to ACM Computing Surveys

Mellor, S. & Kendall, S. & Uhl, A. & Weise, D. 2004. MDA Distilled – Principles of the Model-Driven Architecture. Addison-Wesley

[OMG_MDA] Object Management Group: Model Driven Architecture. www.omg.org/mda

[oslo] Microsoft model-driven development platform prerelease. http://msdn.microsoft.com/oslo

Stahl, T. & Völter, M. 2007. Modellgetriebene Softwareentwicklung. Techniken, Engineering, Management. 2. Auflage, dpunkt,

Starke, G.: Effektive Software Architekturen-Ein praktischer Leitfaden. Hanser, 2008

White, J.; Hill, J. H.; Tambe, S.; Gokhale, A., Schmidt, D. C.; Gray, J. 2009. Improving Domain-specific Language Reuse through Software Product-line Configuration Techniques. Submitted to IEEE Software Special Issue: Domain-Specific Languages and Modeling, Jul/Aug 2009. (http://www.dre.vanderbilt.edu/~jules/white-dslreuse.pdf)