

Composing Domain-Specific Languages for Wide-scope Software Engineering Applications

Jacky Estublier, German Vega, Anca Daniela Ionita

LSR-IMAG, 220, rue de la Chimie BP5338041 Grenoble Cedex 9, France
{jacky.estublier, german.vega, Anca.Ionita }@imag.fr
<http://www-imag.fr>

Abstract. Domain-Specific Languages (DSL) offer many advantages over general languages, but their narrow scope makes them really effective only in very focused domains, for example Product Lines. The recent Model Driven Engineering (MDE) approach seeks to provide a technology to compose and combine models coming from different metamodels. Adapted to DSL, it means that it should be possible to compose “programs” written in different DSLs, which will enable the use of the DSL approach to build applications spanning different domains. The paper presents the Mélusine environment, where such a composition technology has been developed and experimented.

1 Introduction

Most domain engineering approaches emphasize domain modeling as an important mechanism for the development of product families. Domain-specific modeling addresses this issue by designing languages specifically tailored to directly represent the concepts of an application domain.

Domain-Specific Languages (DSL) [1] have several advantages over general-purpose languages:

- They raise the level of abstraction, by proposing constructs directly related to application domain concepts.
- They provide a notation (graphical or not) close to the practitioners’ natural way of thinking.
- They propose specialized tools (like optimizers, analyzers, editors) that embed much domain knowledge and thus provide better support for practitioners, which are not necessarily professional software engineers.
- They enable the partial automation of large parts of the development process, increasing productivity.

General-purpose languages propose generic, low-level concepts, so that developing an application requires lengthy and heavy programming, but they can be used for a very large range of applications, such that the development of high quality environments and tools becomes economically feasible.

In contrast, to be effective, a DSL must target a narrow and well-scoped domain; given the cost of the upfront domain analysis and of the development of the environment and tools, DSLs become economically viable only if many applications are to be

built inside the targeted domain. This compromise is the major limitation of DSL in practice. This limitation can be overcome in two ways :

- reduce the cost of developing tools,
- develop a large number of applications in the domain.

Programming languages address the first point, generating the tools from a formal DSL specification; Product Lines address the second point, emphasizing variations and features. In this paper, we present an alternative and complementary approach, based on the development of generic reusable domains which can be composed for developing wide-scope applications. We illustrate the approach by our environment - Mélusine - in which these solutions have been implemented and tested in real size industrial projects.

The paper is organized as follows: section 2 gives some background information and places our approach in the context of language and MDE technologies, section 3 presents our conceptual domain approach, section 4 is devoted to the subject of domain composition and evolution; the paper ends with related works and conclusions.

2 Languages and Models

Domain-Specific Languages (DSL) is a technology that takes its roots in two technological domains: programming languages and models. Their strength and weaknesses are briefly analyzed in this section, before taking a closer look at DSL.

2.1 The language and compiler technology

Programming languages heavily rely on a technology based on grammars. A grammar G is a finite set of production rules. A language $L(G)$ is defined by induction, as the set of sentences obtained by the reflexive and transitive closure of the derivation relationship, from an axiom. A sentence s is said to conform to the grammar G if there exists at least one sequence of derivations, from the axiom, that produces s ; s is said to pertain to $L(G)$.

In this sense, a grammar can be seen as the model of a programming language and the language as the set of all possible sentences (programs) conforming to that model (Fig.1).

The main lesson from the language domain is that grammars themselves can be seen as sentences in a (meta) language (e.g., EBNF - Extended Backus-Naur Form), defined by another (meta) grammar and compiled by another (meta) compiler. This meta-compiler can automatically generate, for a given grammar, the corresponding syntactic analyzer. For example, this is how Yacc and Lex work [2]. Yacc defines a metamodel for algebraic grammars; since it is formally defined, the algorithms can be proved to be correct and can also be optimized.

The success in languages can be measured by the fact that, on one side, compilers are now trusted and efficient and, on the other side, (simple) languages can be easily built, using meta-compilers. The major lessons are the following:

- Formal meta-grammars enable creating generators, making it easy to produce reliable compilers.
- Conformity is checked, based on the grammar.
- The semantic domain consists of logic and mathematics.

2.2. Modeling: the MDE approach

MDE, as depicted in Fig. 1, presents many similarities to languages, except that the meta-meta level is not a grammar definition language, like EBNF, but a model definition language, like MOF.

The first fundamental difference is that modeling focuses on the relationship between the model and the modeled system, while languages do not consider directly the relationship between a program and reality. In fact, many definitions of model refer to this relationship, a model is usually defined as “a simplification of a system built with an intended goal in mind. The model should be able to answer questions in place of the actual system” [3].

It is important to notice that the *Model_of* relationship is also fundamental to DSL. Indeed, some of the alleged benefits of DSL stem from the fact that there is a close, intentionally, direct link between the program and the modeled reality in the domain.

Interestingly enough, current work in MDE has shifted its emphasis from *Model_of* to *Conform_to* [4] and recognizes that a metamodel is “a model that defines the language for expressing a model” [5].

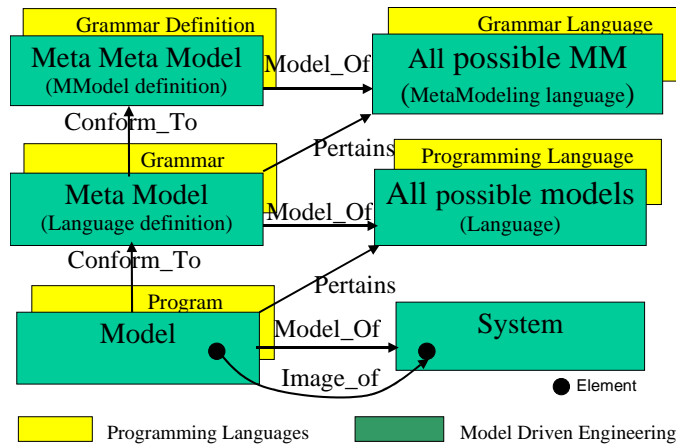


Fig. 1. MDE fundamental relationships

In contrast with programming languages, MDE makes the assumption that a single model can have different views and that the target system is described by many different models, possibly in different metamodels. Therefore, instead of considering a single source and target language, MDE considers that many source models can produce one or more new models.

2.3 Domain-Specific Languages (DSL)

DSL can thus be seen from a programming language or from an MDE perspective. The fundamental difference lies in the relationship between a model (a program) and the modeled system (the meaning of the program).

2.3.1 Model Meaning and Program semantics

Consider the following Java method (seen as a model):

```
int m(int c, int d) {return c*10 + d;} . (1)
```

Its metamodel is the definition of the Java language [6]. Java syntax is defined through a grammar; formal semantics can be defined through denotations toward some mathematical semantic domain. Method m is a (valid) Java program, therefore it satisfies *Conform_to* for the Java specification and, as such, is perfectly defined; however, m gives no information about its “real world” meaning. A virtual machine (the JVM – Java Virtual Machine) recognizes the Java concepts, but ignores what the program means. In this example, a possible *meaning* for m could be that it computes the speed of an object, occurring during c time, with d initial speed. *Interpretation* of 10 is earth acceleration (9.81 ms^2); *interpretation* of c is a time in seconds, etc. This simple example shows that formal semantics (i.e., models and metamodels, per se) do not give any information on the meaning of the modeled system.

The major difference between a program and a model is that a model makes sense only if an interpretation is available (relationship *Image_Of*, Fig. 1 and Fig. 2), while a program's formal semantics do not provide any information about what it *means*.

2.3.2 Metamodel and Domain Semantics

An important characteristic of DSL is that some primitive constructs of the language have an embedded *interpretation* in the target domain. A DSL for our simple example could include a primitive construct for the concept of *speed* with its corresponding operators. A DSL can be seen as a language where some concepts have a predefined interpretation, i.e., these concepts are intended to be used as *Image_Of* (see Fig. 2) their corresponding entities in the target domain.

Following the programming language approach, from the metamodel, a parser is developed, which identifies the language *elements*. In a DSL, these elements constitute the range of an *interpretation* relationship; the elements in the application domain constitute its destination. Therefore, in a DSL, the metamodel makes explicit the model elements for which a relevant interpretation relationship should be established toward the domain elements.

To a large extent, a DSL metamodel is a model of the application domain.

In semiotics and linguistics, semantics is defined as “*the study that relates signs to things in the world*” [7]. From that point of view, the interpretation relationship can be considered the semantics of a DSL. We will call it the *domain semantics*. In summary, we can identify the following important characteristics of a DSL:

- The metamodel is a model of the domain.
- Semantics can be defined with respect to (1) a mathematical domain (formal semantics) and (2) the application domain (domain semantics).

The domain semantics identifies the model entities that are *Image_of* entities in the system and defines their behavior. These entities and their relationships constitute the structural part of the model. The interpretation relationship allows interpreting this structure as a description of the target system structure at a given point in time. *The structural part of a model is a model of a state of the system.*

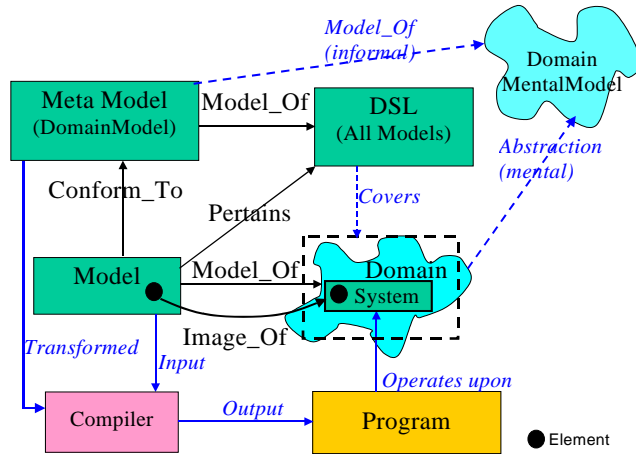


Fig. 2. The DSL Approach

The formal semantics serve essentially to specify the behavior, i.e., the operations on the model entities that change the structure of the model. The interpretation relationship allows interpreting this model change as a system change. *The behavioral part of the model is a model of the system dynamics*, i.e., it describes how the system (is supposed to) evolve.

2.4 The composition and evolution problems

The application domain evolves under market and technology pressures and, therefore, the domain model should evolve accordingly. Unfortunately, most tools are based on the domain model and changes have dramatic consequences: rewrite the compiler, editors, translators, programs (models) and so on. In practice, the cost of such changes is so high that they are not performed. It is not easy to extend the compiler, nor the other associated tools, even for simple changes.

A similar scenario arises when we try to compose different domains. While the composition can be clearly specified at the domain model level, it is not easy to specify the corresponding modifications at the compiler level. Indeed, the difficulties are not at the meta and model levels; the problems arise at the implementation level, because there is another level of abstraction. This problem is well known and extensible (DSL) languages and/or composition of existing DSLs [9] have been proposed. These technologies have not been successfully deployed yet.

3 The Mélusine approach

Our approach to domain composition and evolution follows the underlying trend in DSL and MDE: perform as many activities as possible at the level of the domain model, not at the implementation level. Both DSL and MDE propose to perform not only design, but also a part of the implementation in the problem area, since problem concepts are directly available in the programming (modeling) language. Our proposition pushes this idea a step further: not only the language, but also the run-time architecture is based on the conceptual model of the domain and consequently, domain composition can be performed at an abstract level, using high-level domain concepts.

Mélusine emphasizes two new requirements: (1) Reuse existing components and tools and (2) Support different types of evolution.

3.1 Conceptual domains

As in DSL, Mélusine relies on a metamodel, which is a domain model, and assumes that there is an interpreter for that metamodel. A model is seen as a “program” interpreted by this interpreter; the dynamic part of the model specifies the behavior of entities and the structural part defines the state of the system, see section 2.3.2.

An important characteristic of most DSLs is that the metamodel encapsulates most (if not all) the behavior of the entities in the domain. A survey of DSLs [10] has shown that most DSLs do not provide constructs for user-defined abstractions: only 15% of the surveyed languages provide user defined types and roughly one third provide user-defined functions. This is interesting because in many DSLs, when developing a model, there is no need (and no way) to specify the behavior of the system; this behavior is implicit in the constructs of the language. Most of the time the model represents only the structural part of the application and simply parameterizes the predefined behavior.

The fact a model is purely structural has important consequences:

1. It is relatively easy to fully generate, from the metamodel, a model editor and a model does not need any programming.
2. The system behavior (how it evolves) is mainly defined by the behavior of the predefined domain concepts, implemented by the domain interpreter.
3. Domains can be composed by composing their interpreters, without modifying the existing models.

3.2 Domain-Specific Virtual Machines

A straightforward implementation of a domain interpreter is to transform each metamodel concept into a class and the concept behavior into methods of these classes (plus some technical classes, not discussed here). In this case, the structural part of the model is simply transformed into instances of these classes and considered as the initial state of the interpreter. As this transformation is a bijection, there is an isomorphism between the model and the program state and, therefore, the program state is also a model of the target system. Since execution is based on the domain model, the program state evolves in accordance with the behavior of the associated domain enti-

ties (see Fig. 3). This implementation is not only straightforward - and supported by most UML environments - but also has two important properties:

1. At any time, the state of the program is a model of the target system.
2. The interpreter is a domain virtual machine.

The former property is fundamental for DSL composition (see section 4). The latter is important, since it gives a way to solve reuse and evolution issues. Indeed, since the interpreter implementation is based on the domain semantics, its behavior is defined only in terms of changes of the instances that are images of domain elements. *It is a formal execution.* The execution, in this case, does not rely on lower level libraries or languages, as is usually the case in DSL technology (Fig. 2), nor on a transformation toward lower level “platform dependent” models, as in MDE. A very important property of this approach is that *the interpreter is independent from actual components, tools and platforms.*

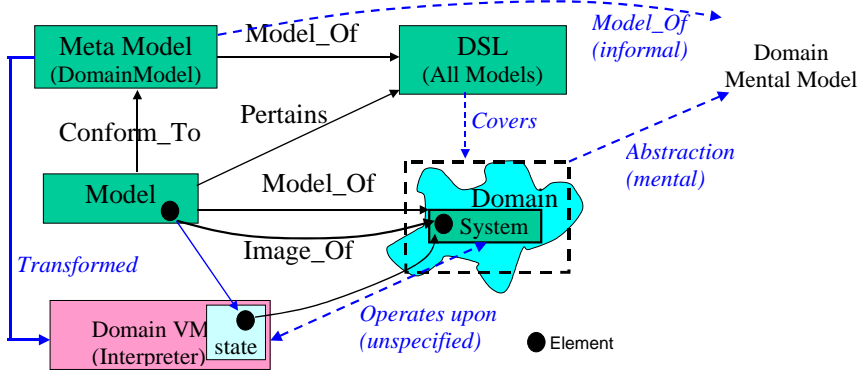


Fig. 3. Domain modeling and Mélusine

3.3 Virtual Machine implementation

Formal execution means the execution only changes the state of abstract entities (Java objects) but, most often, such a change “means” that its “image” in the system (either software or physical) must change its state accordingly. Conversely, if the state of the system changes, the model should be updated accordingly. In other words, formal interpretation is not sufficient, abstract actions should be mapped to actual software components, devices and so on.

In order to reuse existing software artifacts, Mélusine supports a bottom-up approach, defining the concepts of role and tool [11]. A role is an abstract interface for a class of tools. A tool is any piece of software, (a COTS, a legacy application, a component, a library, a physical device and so on), local or distant, that can play a role (directly or through a wrapper).

The *Mapping* expresses the relationship between a state change in a model entity (a Java object) and the correspondent change for its image in the “real” system (a tool executing an action, a device activations) and so on. Conversely, the mapping changes the model entity state to keep them synchronized with changes in the “real” system.

Our requirement is to keep the interpreter independent from mapping. For this purpose, mapping is performed in the Mélusine environment by transparently translating it into aspects, in the AOP (Aspect Oriented Programming) sense. This is easy to do, because the formal interpretation directly changes the state of the model entities; it is enough to capture the methods that change the (Java) model entities and to call the corresponding mapping. Our AOP machine [12] inserts byte code in the interpreter, to execute the aspect in accordance with the mapping specification (Fig. 4).

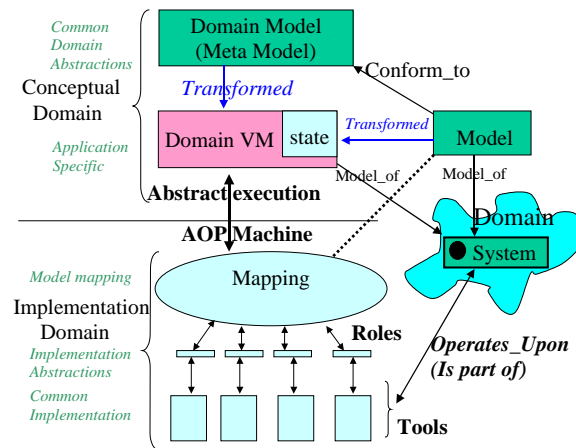


Fig. 4. Conceptual domain and implementation

In our solution:

- the interpreter and the models are independent from any specific implementation,
- the mapping is defined at a high level, between the model and roles (abstract tools),
- actual tools implement abstract services (roles) and can be changed at any time.

Our domain-layered architecture emphasizes the reuse of tools, models and interpreters and enables each actor (analyst, designer, implementer, administrator) to work with tools and concepts at its level of abstraction.

Reusing a domain model implies being able to combine it with other domains in order to cope with wider scope applications (see section 4 about domain composition) and to adapt it to specific requirements (see section 5 about domain evolution).

4 Domain composition

Our approach to domain composition is built on the insight that composition is easily expressed at the conceptual level (see section 2.4), and that most of the reuse benefits can be achieved if one can use the existing domains and their models without modifying them.

Domain composition (section 4.1) consists of defining concepts and relationships that are valid for all the applications in the new composite domain. The new behavior is implemented in the composition virtual machine (section 4.2) by synchronizing its

execution with the corresponding sub-domain interpreters. Then the domain models can be easily composed (section 4.3).

4.1 Domain Model Composition

The composition is initially defined at the conceptual level, by identifying relationships that must be established between existing domains and potentially new concepts and behavior, specific to the composition (see the upper part of Fig. 5).

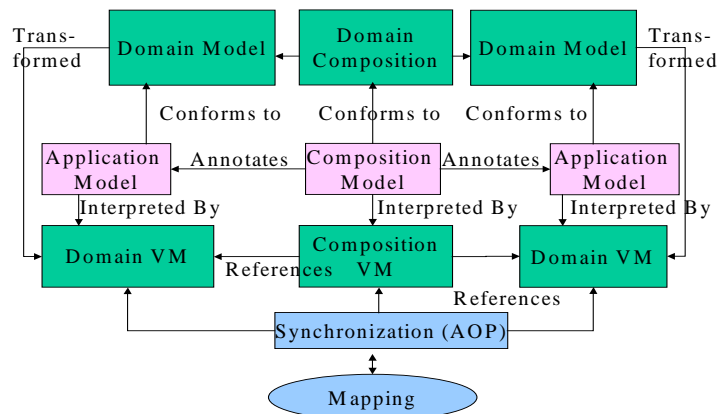


Fig. 5. Conceptual Domain Composition

Because domain models are designed independently, they often contain similar concepts, defined in different ways, since each domain corresponds to a specific concern and outlines the characteristics relevant for this concern only. Two types of relationships can then be established between concepts present in two different domain models: associations (in the UML sense) and correspondences, relating similar or overlapping concepts [13]. For example, Fig. 6 shows some of the new relationships defined for the composition of the Process and Resource domains. The *association* Project/Resource models the resources assigned to the project, while *Activity/Human* indicates the person in charge of an activity; they are usually class associations that capture some emerging behavior of the composition. The relationships *Process/Project* and *Task/Activity* are *correspondences* between overlapping concepts in different domains, in the sense that they can be considered as different aspects of a single unified concept in the composed domain. The example illustrates an important property: domain composition may involve more than two domains. The human resources assigned to an activity must be selected from the available resources of the project; this is a constraint that covers the three composed domains.

A crucial point to highlight is that conceptual composition defines the metamodel of the composed domain. This new metamodel comprises the concepts and associations existing in the sub-domains, the added relationships and, eventually, new emerging concepts. As for any other domain, an interpreter must be implemented for this

new metamodel and appropriate models must be developed for the new composed domain (section 4.3).

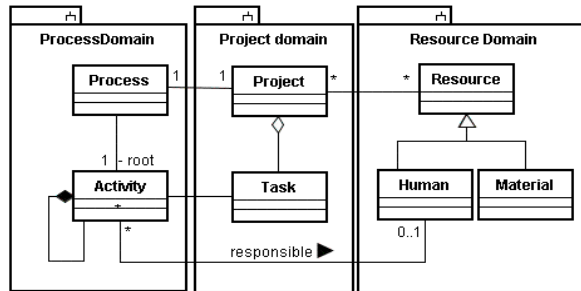


Fig. 6. Conceptual Composition

4.2 Composing Virtual Machine

To foster reuse of existing domains, the interpreter of the composed domain must be implemented by composing the sub-domain virtual machines, as schematically presented in the lower part of Fig. 5.

It turns out that it is necessary to implement the new relationships and behavior without changing the existing interpreters. The intention is to synchronize the execution of several independent virtual machines (sub-domain interpreters). Note that the new virtual machine is allowed to explicitly reference existing classes and associations in the sub-domain interpreters, but not the other way around.

To implement this synchronization, AOP technology is used again, defining aspects that capture the methods representing significant events in the sub-domain interpreters and calling the appropriate methods in the composition domain interpreter, that implements the behavior of the new classes and relationships. This approach may look low level, but this is not the case because, as pointed out in section 3.2, the state of the virtual machine is a model of the target system and the captured events are meaningful in the new conceptual model.

To illustrate this claim, the sample of program presented below shows the synchronization between Process and Resource domains (this is the real, complete code, extracted from an operational document management application). The new behavior to be implemented is assigning a human to be responsible for a particular activity. The aspect `assignActivity` captures, in the process virtual machine, the signals representing that an activity has become ready; the aspect calls the composite virtual machine (class `ActivityAssignmentManager`) that itself calls the resource virtual machine to display a list of humans playing the associated role and changes the responsible.

```

import apel.motor.model.*;
aspect assignActivity(int newState) of Activity {
  when newState == Activity.READY;
  body( JAVA ) {
    activityAssignmentMaager.assignActivity(instance);
  }
}
  
```

```

    }
}
public class ActivityAssignmentManager {
    public ActivityAssignmentManager () {
        resource = Domains.getRoot("resourceEngine");
    }
    public Vector getPotentialHumans(String roleName) {
        Role theRole = resource.getRole(roleName);
        return resource.getHumanIds(theRole);
    }
    public void assignActivity (Activity activity) {
        String user = showAssignDialog(activity.getName(),
            getPotentialHumans(activity.getRole()));
        return activity.setResponsible(user);
    }
}
}

```

The classes in the composition virtual machine are similar to other domains, they capture abstract concepts of the composition and, as shown in Fig. 4, may require a mapping to lower level software, components, devices and so on, just as for other domains, see section 3.3.

Composing virtual machines is not necessarily easy, but not too complex either, because each interpreter is a direct implementation of the corresponding domain concept and therefore, the composition is performed at the conceptual level of the composite domain. This is much easier than trying to change the existing interpreters or to implement a new one. In our experience, a typical composition interpreter is very tiny; for instance, in the document management application, the composition interpreter is about 15% (in LOC) of all composed domain virtual machines.

4.3 Model Composition

The composite domain has its own metamodel, meaning that we may need to develop new models conforming to this metamodel. These models can refer to existing sub-domains models and can make the links between them explicit. For example, for the document management composite domain presented in section 4.2, the data circulating in the data flows defined in a process model should be associated with actual product definitions in the product data management domain; more specifically, the entity called *doc* in the process model is the document *specifProjectX* in the Product domain. This information is captured by the composite model and is interpreted by the composition virtual machine (see the code of class `ActivityAssignmentManager` in the previous section).

There is an important point to highlight: the existing models have not changed at all, but a new model was defined that relates the existing sub-domain models. The experience shows that this is very easy to do and allows models to be reused. Since the domain itself evolves, its model has to incorporate the novelties introduced in the real domain.

4.4 Domain evolution

Some variations in the domain can be expressed without having to change the domain model. A *feature* captures *optional domain behavior*; they are implemented in the same way as mappings, by capturing domain behavior and adding/substituting it with the actual feature behavior. In this way, the domain model, the interpreter and models are totally unchanged. For example, the Process domain has *trace*, *mail_notification* and *persistent* features. They can be selected or not for each application in the domain.

Extensions are new concepts, added to the domain and linked by associations with the domain model concepts. These concepts, with their semantics, behavior and implementation, make sense only in connection with the domain model.

This is in contrast with domain composition, where each domain is independent and makes sense by itself. Nevertheless, the technology we use to define and implement extensions is the same as for domain composition [14].

5 Related work

The idea of extending or specializing a language by modifying its interpreter has been actively studied in the context of Meta Object Protocols [15] and reflective programming languages. In this context, the formal domain interpreter can be seen as the meta level, the actual tools and components as the base level and the mapping as the causal link between the two. An important difference in our approach is that the meta level is directly related to the domain (domain semantics) and not (only) to the formal semantics. Another difference is that we develop the two levels separately, in order to be able to evolve them independently, and we use AOP to keep them synchronized.

More generally, our approach is based on the idea that a domain presents two interfaces: when used in the development of a particular application, only the model editor is visible, the domain model and its interpreter are hidden. When composed or extended, the conceptual model is exposed like a white box. In this regard, our work can be related to the idea of open implementations [16][17] and more particularly to the ideas of open design programming languages [18]. The domain-specific virtual machine can also be related to the idea of a UML virtual machine [19]. The difference is that, instead of implementing a low-level UML action language, we implement the behavior of the high-level domain concepts.

Composing modeling languages by composing their corresponding metamodels is also considered in [20], but the approach is limited to the generation of the model editor and does not consider the domain behavior and run-time environment. Fritsch and Renz [21] present a meta-level architecture for the development of product lines based on several related DSLs, similarly Barry et Al. [22] present an example of composition of Process and Product Data Management software, by composing metamodels. Although very similar to our approach, both are limited to a particular domain. We have generalized the approach and have applied it systematically in very different domains.

The problem of metamodel evolution and adaptation and its impact on the corresponding interpreters has been stated in [23] and an approach for metamodel evolution based on a transformation language is presented in [24]. These approaches are based on the idea of refactoring the metamodel and automating the impact on the existing models and interpreters. Our approach is based on the idea of modularly defining the metamodel and reusing the existing models and interpreters. The two approaches complement each other very well.

Another solution other than DSL would be to use a general modeling language, such as UML 2.0 [5], that offers support for:

- *Evolution*, with the possibility to introduce variability (through templates, power-types for creating metamodels and semantic variation points like model annotations) and extensions (with inheritance, stereotypes, constraints and tagged values);
- *Reuse*, with patterns, stereotyped packages like model libraries or frameworks and the facility to merge packages (models), by introducing a generalization for classifiers with the same name;
- *Domain-specific concepts*, with profiles, that allow for the definition of stereotypes grouping property extensions.

UML was not adopted, because defining a DSL is simpler than defining a profile. A profile, even if extending only a metamodel subset, requires conformity with the huge UML semantics and checking this conformity is not entirely supported by existing tools. From the point of view of code generation, model transformation often needs supplementary marking models [25] and restriction to UML subsets. Executable UML [26] goes forward, by creating an UML profile and adding actions for a detailed definition of the behavior, such as to be executed. Models for different subject matters are woven together by an executable UML compiler that, unlike Mélusine, keeps all the burden of general languages. Apart from making the composition at the meta level, between small DSLs, the flexibility added by our approach also comes from its layered architecture, which separates the models from their implementation tools and allows domain extensibility.

A possible solution for directly manipulating the domain concepts is expected to be given by future tools, for example, based on MOF [27], allowing users to define entirely new languages via metamodels. In the meantime, the Eclipse Modeling Framework (EMF) seems to be closer to our needs, proposing Ecore meta-metamodel, similar to MOF and expressing models as XML schemas, UML class diagrams or annotated Java [28]. EMF provides all the facilities and extensibility of Eclipse and also offers a number of tools to support automatic editor generation and round-trip engineering, while still leaving the user the possibility to write code that remains outside models.

6 Conclusion

Our work is based on a simple idea: DSL is a good engineering approach, but it is limited by its narrow scope; so, composing DSLs would permit implementing wide-scope applications, while retaining the strong points of DSLs. Unfortunately, this simple idea is far from trivial if one seeks a solution answering the question: how is it

possible to *compose DSLs*, but still *reuse* existing components and tools and support different types of *evolution*.

We have spent a number of years answering this question, implementing (re-implementing) solutions, and validating them in real scale industrial applications. The lesson we have learned is that no single technology or technical approach alone can solve these issues. Indeed, our approach puts together ideas coming from DSL, MDE, programming languages, AOP and component technologies.

AOP, as well as components, are implementation techniques, not engineering approaches. Nevertheless, AOP is our corner stone implementation solution, because it allows both reuse and evolution.

Our approach is typical of MDE, but, in contrast with the main stream, our technology allows for the composition of independent models and metamodels by defining relationships among their concepts. This composition technology is a practical and high-level way to compose DSLs, defining a new, extended DSL, that can be further composed itself. Furthermore, as this approach uses formal interpretation and AOP techniques, it is possible to reuse the existing domains (interpreters, models and so on) without changing them. Finally, the introduction of features and extensions at the conceptual level and the explicit mappings and roles at the implementation level, provide large evolution capabilities.

Composing DSLs in the general case is very difficult, but it becomes a practical and promising software engineering approach when supported by a methodology and a specific environment, like Mélusine, providing high-level modeling, generation, evolution and reuse.

Acknowledgements: The work of Anca Daniela Ionita is supported by a Marie Curie Intra-European Fellowship, within the 6th European Community Framework Programme.

References

1. D. S. Wile., Supporting the DSL Spectrum, Journal of Computing and Information Technology, CIT 9, 2001 (4) 263-287
2. Levine, John R., Tony Mason and Doug Brown [1992]. [Lex & Yacc](#). O'Reilly & Associates, Inc. Sebastopol, California
3. Bézivin, J., Gerbé O., "Towards a Precise Definition of the OMG/MDA Framework", ASE'01, Novembre 2001
4. Favre J.M., "Towards a Basic Theory to Model Model Driven Engineering", 3rd Workshop in Software Model Engineering, WiSME 2004, <http://www-adele.imag.fr/~jmfavre>
5. OMG, "UML 2.0 Superstructure Specification", August 2003
6. Gosling J., Joy B., & Steele G., The Java Language Specification, Addison Wesley, 1997
7. J.F. Sowa, Ontology, Metadata, and Semiotics, in B. Ganter & G. W. Mineau, eds., *Conceptual Structures: Logical, Linguistic, and Computational Issues*, Lecture Notes in AI #1867, Springer-Verlag, Berlin, 2000, pp. 55-81
8. R.A. Falbo, G. Guizzardi, K.C. Duarte, An ontological approach to domain engineering, *Proc. of the 14th Int. Conf. on Software Eng. and Knowledge Eng.*, Ischia, Italy, 2002, ISBN:1-58113-556-4, pp. 351 – 358

9. R. Prieto-Diaz, Domain Analysis: An Introduction, *Software Engineering Notes*, Vol. 15, No. 2, April 1990
10. S. Thibault, "Langages Dédiés : Conception, Implémentation et Application", Ph.D. Thesis Université de Rennes1, 1998
11. T. Le-Anh, J. Villalobos, J. Estublier. Multi-level Composition for Software Federations. In Proceedings of the 6th European Joint Conferences on Theory and Practice of Software (ETAPS 2003) Workshop on Software Composition, April 2003
12. F. Duclos, J. Estublier, R. Sanlaville "Separation of Concerns and The Extended Object Machine." Submitted to Journal Advise. <http://www-adele.imag.fr/Les.Publications/BD/ADVICE2004Est.html>
13. J. Estublier, A.D. Ionita, Extending UML for Model Composition, *Australian Software Engineering Conference*, 29 March – 1 April, Brisbane, Australia
14. J. Estublier, J. Villalobos, T. Le-Ahn, S. Sanlaville, G. Vega. An Approach and Framework for Extensible Process Support System. . In Proceedings of the 9th European Workshop on Software Process Technology (EWSPT 2003), September 2003
15. G. Kiczales, J. des Rivières, D. Bobrow. The Art of the Metaobject Protocol. MIT Press, Cambridge Massachusetts, 5th Printing 1999
16. G. Kiczales. Beyond the black box: Open Implementation, *IEEE Software*, Vol. 13 Issue 1, January 1996
17. C. Maeda, A. Lee, G. Murphy, G. Kizales. Open Implementation Analysis and Design, *ACM SIGSOFT Software Engineering Notes*, Vol. 22 Issue 3, May 1997
18. P. Steyaert. Open Design of Object Oriented Languages. PhD thesis, Vrije Universiteit Brussel, 1994.
19. D. Riehle, S. Fraleigh, D. Bucka-Lassen, N. Omorogbe. The Architecture of a UML virtual machine, Proceedings of the 16th ACM SIGPLAN Conference on Object oriented programming, systems, languages, and applications OOPSLA 2001, Tampa Bay, USA, October 2001
20. G. Karsai, M. Maroti, A. Ledeczi, J. Gray, J. Sztipanovits. Composition and Cloning in Modeling and Meta-Modeling, *IEEE Transactions on Control System Technology*, Vol. 12 No. 2, March 2004
21. C. Fritsch, B. Renz. Four Mechanisms for Adaptable Systems A Meta-level Approach to Building a Software Product Line. Proceedings of the 3rd International Software Product Lines Conference, SPLC 2004, Boston, USA, August 2004
22. A. Barry, N. Baker, J.-M. Le Goff, R. McClatchey, J.-P. Vialle. Meta-Data based design of Workflow Systems. Proceedings of Workshop on Meta-data and Active Object Model pattern mining, OOPSLA 1998, Vancouver, Canada. October 1998
23. J. Zhang, J. Gray. A generative approach to model interpreter evolution. Proceedings of Workshop on Domain Specific Modeling, OOPSLA 2004, Vancouver, Canada. October 2004
24. J. Sprinkle, G. Karsai. A Domain-Specific Visual Language For Domain Model Evolution. *Journal of Visual Languages and Computing*, vol. 15, no. 2, April 2004.
25. S. Mellor, K. Scott, A. Uhl, D. Weise. MDA Distilled: Principles of Model-driven Architecture, Addison-Wesley, 2004
26. S. Mellor, M. Balcer. Executable UML: A Foundation for Model Driven Architecture. Addison-Wesley, 2002
27. OMG, "Meta Object Facility (MOF) 2.0 Core Specification", October 2003
28. F. Budinsky, D. Steingerg, E. Merks, R. Ellersick, T. Grose, "Eclipse Modeling Framework", Addison Wesley, 200