A Multi-dimensional Framework for Characterizing Domain Specific Languages

Ø. Haugen[1,2], P. Mohagheghi[1]

[1] SINTEF ICT, P.O.Box 124 Blindern,N-0314 Oslo, Norway
[2] University of Oslo, Dept. of Informatics, P.O. Box 1080 Blindern, N-0316 Oslo, Norway
{oystein.haugen, parastoo.mohagheghi}@sintef.no

Abstract. The paper presents a questionnaire to assess Domain Specific Languages based on a multi-dimensional framework for characterizing languages. An issue is whether and how to distinguish between characteristics of domain-specific and general purpose languages. We discuss how to emphasize dimensions that are particularly important for domain-specific languages such as being formal, yet transparent as well as integrable with other languages. We consider hazards and potentials of the approach.

Keywords: DSL, language quality, language assessment.

1 Introduction

When applying Domain Specific Languages (DSLs) instead of General Purpose Languages (GPLs), the software design process will in fact be divided in two distinct activities, the design of the DSL and the usage of it.

The ultimate success criterion for a DSL is whether the usage of it produces good products. It is also possible to harvest empirics from the product design process. But is it possible to give a fruitful evaluation of the DSL only by assessing the language itself?

This paper is about assessing a language based on a general quality framework through the means of a structured questionnaire. The idea is that existing languages can be adequately characterized and that future languages can be guided in the right direction. The advantage over process empirics is that our questionnaire can be applied at a much earlier stage in the product development. The questionnaire was originally oriented towards GPL evaluation, especially for object-oriented languages, and relevance to the evaluation of DSLs is discussed.

Following this introduction we present the multi-dimensional quality framework in Section 2 on which the questionnaire presented in Section 3 is based. Section 4 discusses DSLs particular properties that go beyond those of languages in general. Section 5 presents related works, and Section 6 summarizes.

2 A Framework for Characterizing Languages

We characterize languages first by handling one language at one point in time. Then we add the complexity of having several languages combined, but still only at one point in time. Finally we consider language evolution over time.

The concepts in this Section originate in [1].

2.1 Characteristics of One Language at One Point in Time

We define three main dimensions of a language: *expressiveness*, *transparency* and *formalization*.

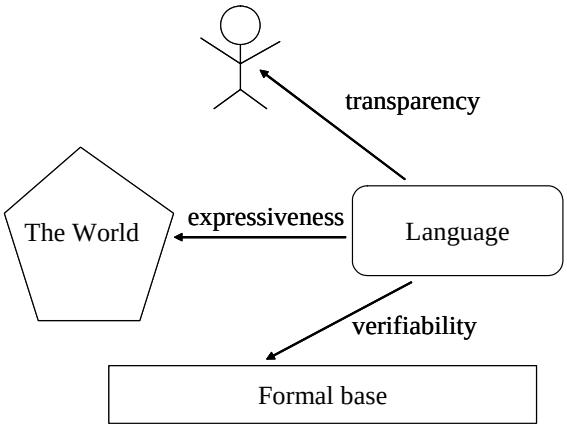

Fig. 1. Language dimensions

Expressiveness.
Expressiveness is not only about whether the language is Turing-compatible or not. We define a set of categories of expressiveness forming the acronym MAGIC. A given language construct may serve more than one purpose and fall into more than one of the categories.

We present our categories in the opposite order of the acronym MAGIC for explanatory reasons.

C – Concepts.
What are the basic concepts of the language? Are the basic concepts on a high abstraction level, or will a designer need several language concepts to define a domain level concept? While Java has concepts like “class”, “object” and “method”, we find in UML more expressive basic concepts like “state machine” and “interaction”.

I – Identity
Given an object, how does the language describe what this object is? In a nutshell this category is about compact modifications of identity to express similarities. Typical I-constructs are inheritance, overriding and overloading. While overriding may be bound at runtime, overloading can always be determined at compile time.

G – Generation
Generation is about dynamics. How are new entities created, and eventually destroyed? How are the relations between the generator and the entities it generates? We also categorize concurrency and parallel processes with their communication here.

A – Aggregation
Modern systems consist of smaller systems that again consist of even smaller systems. To get our arms around a system we often organize it in layers, and the layers may consist of sub-layers etc. A “seamless” language will apply the same language features on every nesting level and there will be no fixed number of layers.

M – Meta
Our notion of meta is concerned with language constructs within one language that work on the description of the system while that very system is running. The most common meta-feature is that

of reflection where information of an entity's definition is made available to the program through predefined query operations. There are also languages (like LISP) that include constructs to execute objects that have been created by the very same program. This is generative programming within one language.

Transparency (or simplicity)

Transparency is about the relation between the language and the humans. How easy is the language to use and to understand? This question is almost impossible to answer without saying something about the persons that are supposed to use and/or to understand the language. If you know Java it is not so difficult to understand C#. This dimension is dependent upon more than the language itself, but we may give some general advice.

Distinct syntax
Similar syntax to define different constructs or dissimilar syntax to define similar concepts will be confusing. This may be compared with watching a soccer match where the two teams have very similar shirts, or conversely that all players have different shirts.

Structure similarity
It has been shown that if the model is structured similar to the way the running system is organized, this helps the understanding.

Locality and history independence
Modern systems are big. Therefore it should be possible to locate the significant pieces easily and these pieces should be well isolated. We call this local reasoning. Just as the reader should not need to read the whole system to understand a small piece, it should not be necessary to know the whole history of an object to grasp its future function. The concept of state machines may serve as an example of locality and history independence.

To understand what will happen in a state machine it is sufficient to know in which state the machine is and what signal it consumes. This determines the transition. It is not necessary to know the earlier transitions.

Educational prerequisite
If I try and read a Chinese newspaper I readily accept that I do not understand it. I know that I need to study Chinese to be able to read it.

The need for education should be clear and well documented. Sometimes knowledge of the domain is sufficient, while otherwise explicit training may be required.

Formalization (supportability)
Formalization is the relation between the language and a semantic base (semantic model). A reasonably precise semantics is the prerequisite for a number of desirable properties. Therefore one may believe that formalization is always a top priority, but experience shows that this is not the case. By formal definition we often mean a definition given in mathematical or logical terms. We also accept that a definition of a language given in an executable language such as a programming language represents formalization. Thus, code generation schemes may represent formalization.

We also have clear evidence that bringing an organization from modeling in a sketchy way to modeling in a precise way is a considerable mental step [2].

Simple analysis
Any artificial language should have a precise syntax and a reasonably precise set of well-formedness rules. The well-formedness rules are often described as constraints on the metamodel and may need special support to check.

Executability
Programming languages are always executable, while modeling languages have a tradition for not being executable per se. This probably stems from the sketchy tradition of the modeling languages. Modern DSLs are normally executable and their semantics is given through their code generators. Executability is defined as the possibility to generate code that can be interpreted as a program by a computer. However, models may also include information that is not used to generate the program, for example constructs needed for debugging or analyses. Not all DSLs are executable (e.g., static structure definition). The main advantage of a model being executable is that the coding phase has been eliminated meaning more effective design and less error prone implementation.

Model checking
Model checking means to explore the model executions systematically in order to find situations where certain exploration targets hold. Normally the model ought to be of a specific kind e.g. a set of communicating state machines. The language may restrict the model such that effective model checking is possible.

Automated metrics
Complexity metrics may be produced to give an indication to how complicated a specific piece of software is.

Refinement
Formalization opens up for supporting concepts like refinement. When a model evolves it is useful to be able to assert that the evolution is consistent in some way. That model evolution conforms to refinement is one way to define consistent evolution steps.

2.2 Characteristics of Several Languages Combined

We proceed to consider modeling with several languages; e.g., from different views or for different purposes.

Integrability
We consider the integration of languages that are used to model a system. When assessing one language it is interesting to explore how well it integrates with other languages to model one system.

Superlanguage
One approach to integration is to combine several languages into one superlanguage. The advantage of defining a superlanguage is that we are back to the situation with one language to be assessed. There are two disadvantages with this approach: a) the superlanguage easily becomes large, and b) it is easy to create *yet* another, even more sophisticated modelling language whose notions and constructs are (partially) not covered by the superlanguage [8].

Common Exchange Language
To exchange information between the sub-models we may define a common exchange format with its own formal semantics. The advantage of this approach is that the exchange language needs not define semantics for whatever goes on inside the sub-models, but only for the exchange.

MetaModel Transformations
In this approach, the metamodels are specified, as well as the transformational relationships between them in a suitable formal language [8]. The advantage is the possibility to combine and compare languages. The metamodels are used to define a root metamodel. The relationships will then be transformed as being specialization (introducing more specialized concepts), restriction (adding constraints) or degeneration (for example removing concepts) of this metamodel.

Traceability
When we describe systems where the sub-systems are made in different languages, it is practical to keep close control over associations between entities in one sub-model with entities in another. One object in the real system is represented by separate objects in the different sub-models.

Examples are requirement objects that turn into design objects or database records represented both in the tele-protocol and the SQL database schema.

2.3 Characteristics of Language Evolution over Time

The languages are more stable than the models that are instantiated from them, but languages still evolve. Just as models must be updated to cater for new requirements, so must languages.

Maintainability
How is the language maintained? Is there a group of people or institutions that take that responsibility? How will changes be propagated in the community? There are organizational as well as technical issues related to maintenance, and we are not only talking about bug-fixes. We must consider improvements and versioning as well.

Extendability
Adding new features to a language is not the same as language extendability. When a language is extended, another language appears, but the original still remains. Hopefully there is a relation between the original and the extended similar to what we have in class inheritance. When the original language is maintained the extended language should be maintained automatically together with the original.

An example of language extension mechanisms is the profiling mechanism in UML which takes UML 2 as starting point and the profiles may extend classes of the UML metamodel to create a profile-specific metamodel.

Scalability
Scalability is the ability to work well with small models as well as big and gigantic models. This is an ability that becomes increasingly important as time passes. A language normally evolves from being small with few concepts that work very well on toy examples. If the examples are convincing enough, the language becomes popular and is being used for more purposes and for bigger systems. This always results in establishing the need for more structuring constructs. If these extensions are successful the language is used for even bigger systems and for an even wider range of systems.

3 The Questionnaire

The following schema is a questionnaire that can be used to assess a given language. Each question represents a dimension and the min and max characteristics indicates a range of values. The assessor will give a value between 0 (min) and 10 (max) and up to 3 extra points for features that go beyond what the max value characteristic covers. The evaluators should prior to the individual

assessments define weights to every question representing the relative importance of the dimension in the domain in question.

Question	Min	Max
One Language at One time		
Expressiveness		
Concepts		
1. Expressing structure	Flat language, no referencing, no hierarchies	Hierarchies, nesting, referencing
2. Units of behavior	One main program only	Many different units of behavior with value parameters
3. Templates and type parameters	No higher order concepts	Templates and type parameters
4. Communication	No communication primitives	Different kinds of comm. primitives for synchronous as well as asynchronous.
Identity		
5. Inheritance	Inheritance cannot be expressed	Multiple inheritance
6. Overriding	No overriding	Overriding of all pattern concepts (classes, methods, etc.) bound at runtime
7. Overloading	No overloading other than for normal arithmetic	User defined overloading of all methods / operations
8. Identity modifiers	No other	Other identity modifiers exists
Generation		
9. Entity / behavior creation	No dynamic creation	Dynamic creation of methods, objects etc.
10. Entity destruction	No destruction	Automatic destruction (garbage collection)
11. Concurrency	Sequential process only	Fully independent concurrent processes
12. Arbitration	No arbitration	Concepts of priority as well as synchronization exist
Aggregation		
13. Namespace (nesting)	One namespace	Multiple, nested namespaces
14. Levels of system	One level	Unlimited number of levels. Each level is uniform with the former.
Meta-concepts		
15. Reflection	No reflection	Fully equipped with reflective mechanisms
16. Interpretation of objects	Objects cannot be seen as programs	Objects may be interpreted as executable
Transparency		
17. Syntax	Counterintuitive syntax	Different things are described differently. Similar things similarly. Intuitive symbols.
18. Descriptions vs. runtime	Running system bears no resemblance to model	Running system has a structure similar to that of the description
19. Locality	Reasoning always need the	Understanding can be achieved for

Question	Min	Max
	whole model	small units at the time
20. History dependence	Execution can be understood only if the whole history is known	Situation can be understood with a minimum of knowledge of earlier execution.
21. Educational prerequisite	Unclear what knowledge is needed	Well defined educational requirements. Otherwise intuitive.
22. Efficiency	Not important	It should be time-efficient to create or update a model
Formalization		
23. Static analysis	Not even syntax analysis possible	Syntax rules are well-defined and so are the well-formedness rules
24. Executability	Not executable	Executable
25. Dynamic analysis	No analysis of execution possible	Model checking possible with effective means
26. Metrics	Metrics impossible	Metrics can be compiled to show complexity measures
27. Formal semantics	No mathematic formalisms	Logical foundation that can support analysis of refinement
28. Completeness	Language only to be used for sketching	Language to give a complete representation of the system
Multiple Languages		
29. Integrability	No features for integration with other languages	Superlanguage, common exchange format or metamodel relationships
30. Traceability	No inter-language references	Inter-language references are primitives of the language
Languages over time		
31. Standardization	No central standardizing body	Standards body in place with rules and principles
32. Propagation of changes	New versions of the language do not occur	New versions are automatically propagated to all registered platforms
33. Tools	No language-specific tools	Language-specific tools are available within a few months of new language release
34. Compatibility	Compatibility is not an issue	Backward compatibility is ensured or automatic migration is made available
35. Deprecation	Deleted concepts is not an issue	Deprecated concepts are handled explicitly
36. Extension	Language is not extendable	Language has well defined means to extend itself as language
37. Restriction	Language restriction is not an issue	Language restriction can be expressed
38. Scalability	Only applicable for small models	Syntax has means to keep overview when model increases
39. Degradability	Execution degrades sharply and discontinuously	Execution degradation is well predictable and linear

4 What is Specific for Domain Specific Modeling languages?

One issue is how to distinguish DSLs from GPLs. Can the particulars of DSLs be captured with more weight on some dimensions?

DSLs are typically small, highly focused languages used to model clearly identifiable systems. An important criterion is domain-appropriateness: A DSL must be powerful enough to capture the major domain concepts and should match the mental representation of the domain. We try and cover this in the questionnaire by the questions on transparency (#17-22).

DSLs are typically used for prediction or simulation, as well as code generation and execution. Thus the language should be formal and accurate, which we cover by the questions on formalization (#23-28).

There may be several DSLs when developing a complex system. Thus there is a need for integration between these languages. Round-trip engineering where changes in the generated code are reverse-engineered back to the model is dependent upon traceability. These aspects are covered by questions #29 and 30.

Any DSL with a diagrammatical syntax should have proper layout, allowing easy distinction between concepts while similar concepts should have similar layout. It should be possible to hide details and expose them on demand. Question #17 covers this.

Finally creating a DSL should be cost-effective or costs should not exceed the advantages of defining a DSL rather than using a GPL. This is not easy to evaluate merely from the language itself, but the bottom line is whether the DSL has increased productivity for developers, or has reduced maintenance costs by improving quality of software and providing models that are easier to understand, correct and evolve.

5 Related Work

Krogstie describe a quality framework for evaluating modeling languages in general in [3], which is based on earlier work described for example in [4]. The framework covers concepts such as *domain appropriateness* (powerful enough to describe the domain), *participant language knowledge appropriateness* (correspond as much as possible to the way that individuals perceive reality and based on experiences with languages used for the relevant types of modeling), *knowledge externalizability appropriateness* (there are no statements in the explicit knowledge of the participant that cannot be expressed in the language), *comprehensibility appropriateness* (e.g., each language's phenomenon should be easily distinguishable and the number of phenomena should be reasonable) and *technical actor interpretation appropriateness* (the language lend itself to automatic reasoning. This requires formality (i.e., both formal syntax and semantics).

Grossman et al. have evaluated UML using a set of criteria [5] such as *having right data* (necessary constructs and their semantics. [6] adds *completeness*, which is capturing all concepts), and *accuracy of concepts*. All the criteria are also relevant for DSLs, while we can add some other criteria from other literature that is more relevant for DSLs from [6]:

- *Inherence*: the concepts should be straight to the point and focus on essential aspects only,
- *Consistency*: the concepts must not conflict with each other in representation of (abstraction from) aspects of the real world,
- *Clarity*; i.e. a designer must be able to comprehend the concepts and rules, as well as be able to apply them in models without spending too much time and effort (subjective!).

Paige et al. also identify some principles in the design of modeling languages in general, where the most important one is *simplicity*; i.e. no unnecessary complexity, including being small. Some other principles are *uniqueness or orthogonality* of features, c*onsistency* (language features cooperate to meet language design goals), *reversibility* (changes in one stage can be reflected back to earlier stages), *scalability* (large and small systems can be modeled) and *space economy* (concise models are produced) [7]. Some of the principles may be of more importance for DSLs such as

space economy. The FRISCO project report of 1998 by Falkenberg et al. discusses three properties of modeling languages important for defining meta-model transformations (relations of languages to each other); i.e. *expressiveness* (to what degree a given modelling language is capable of denoting the models of any number and kinds of application domains), *arbitrariness* (the degrees of freedom one has when modelling one and the same application domain, and *suitability* (to what degree a given modelling language is generally applicable, or specifically tailored for the particular task of modelling a specific kind of application domain).

For modeling languages, selecting language goals also depends on the purpose of modeling, such as communication, code generation, simulation, or performance analysis.

What distinguishes our approach from the above mentioned approaches is that we aim to operationalize the criteria through our questionnaire so that a concrete and quantitative assessment is possible.

6 Conclusion and Future Work

We have presented a questionnaire for assessing DSLs based on a multi-dimensional quality framework such that the assessment can be done only by assessing the language itself.

Most of our dimensions are based on experience and on state-of-the-art theory, and some of the dimensions are directly intuitive. Still there is a need for empirics of applying the assessment method on real and semi-real examples.

An alternative proposed for evaluating languages is collecting some metrics from the language's metamodel such as the number of concepts and relations as a measure of a language's complexity [9]. On the other, a more complex metamodel may actually facilitate the task of modeling and cannot per se be used as an indication of a language's suitability for a specific task. Comparing such metrics needs proper baseline data and involves human judgment.

Our method cumulates assessment points from each dimension. This will mean that languages with many features will get a high score. The risk is that simplicity will disappear in all the language constructs that add to the score. The evaluators may compensate for this by carefully emphasizing the transparency dimension in the weight definition.

Another problematic issue is that the dimensions are probably not orthogonal and again this must be compensated by the weight definition.

Finally we have no way to assert that our set of dimensions is complete or even that the dimensions span the full "quality space". We believe that our future exploration of the method by applying it to a number of existing and emerging languages will make shortcomings visible. The questionnaire will be used in the design of DSLs in a European research project to collect language requirements.

Acknowledgments. This paper has been sponsored by EU FP6 project MODELPLEX (MODELling solution for comPLEX software systems) (Contract no. 034081).

References

1. Haugen, Ø.: *Practitioners' Verification of SDL Systems*. Institute for Informatics. Oslo, University of Oslo: Dr. Scient. Thesis. 290 p (1997)
2. Haugen, Ø., R. Bræk, et al.. The SISU project. In *SDL '93 Using Objects*. Proceedings of the Sixth SDL Forum, Darmstadt, Germany, North Holland (1993)
3. Krogstie, J.: Evaluating UML Using a Generic Quality Framework. Chapter in *UML and the Unified Process*, Idea Group Publishing, 1-22 (2003)
4. Lindland, O.I., Sindre, G., Sølvberg, A.: Understanding Quality in Conceptual Modeling. *IEEE Software*, Vol. 11, No. 2, 42-49 (1994)

5. Grossman, M., Aronson, J.E., McCarthy, R.V.: Does UML Make the Grade? Insights from the Software Development Community. *Information and Software Technology* 47, 383-397 (2005)
6. Teewu, W.B., van den Berg, H.: On the Quality of Conceptual Models. In *Proc. ER'97 Workshop on Behavioral Models and Design Transformations: Issues and Opportunities in Conceptual Modeling* (1997)
7. Paige, R.F., Ostroff, J.S., Brooke, P.J.: Principles for Modeling Language Design. *Information and Software Technology* 42, 665-675 (2000)
8. Falkenberg, E.D., Hesse, W., Lindgreen, P., Nilsson, B.E., Han Oei, J.L., Rolland, C., Stamper, R.K., Van Assche, F.J.M., Verrijn Stuart, A.A., Voss, K.: *A Framework of Information Systems Concepts*. The FRISCO Report (Web edition), ISBN 3-901882-01-4 (1998)
9. Rossi, M., Brinkkemper, S.: Complexity Metrics for System Development Methods and Techniques", *Information Systems* 21(2), 209-227 (1996)