Analysis of a Metamodel to Estimate Complexity of Using a Domain-Specific Language

Jonathan Sprinkle
University of Arizona, ECE
1230 E Speedway Blvd, Bldg #104
Tucson, AZ 85721-0104
sprinkle@ECE.Arizona.Edu

ABSTRACT
This paper considers the complexity of building models by analyzing the structure of the metamodel that defines a domain-specific modeling language. An algorithm is presented that generates a state model to produce at least one instance of every model in the metamodel, and the complexity of that state model provides insight into the complexity of the modeling language.

Keywords
Metric, Metamodel, User Interface, Usability

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*; D.3.2 [**Languages**]: Specialized application languages, very high-level languages

1. INTRODUCTION
Domain-specific modeling presents a new specification interface to a modeler. Implicit to the success of DSM is the usability of the modeling environment. It is important to distinguish usability from expressivity, which prefers fewer features, in order to reduce specification time. While DSM generally prefers to raise the level of abstraction of the specification of a system, it does not aim to do so solely to optimize expressivity, since this inevitably produces "point solutions" that have high expressivity for a particular problem, but which do not lend themselves to reuse. The relative complexity of a domain-specific modeling environment, compared to using general modeling or programming for system specification and implementation, will determine how effectively a particular modeling language can be used by a domain expert who may not be a software expert. A useful metric the complexity of a modeling languages will likely depend on several measurements. As a goal of modeling is to create "correct by construction" models, a few candidate measurements immediately suggest themselves:

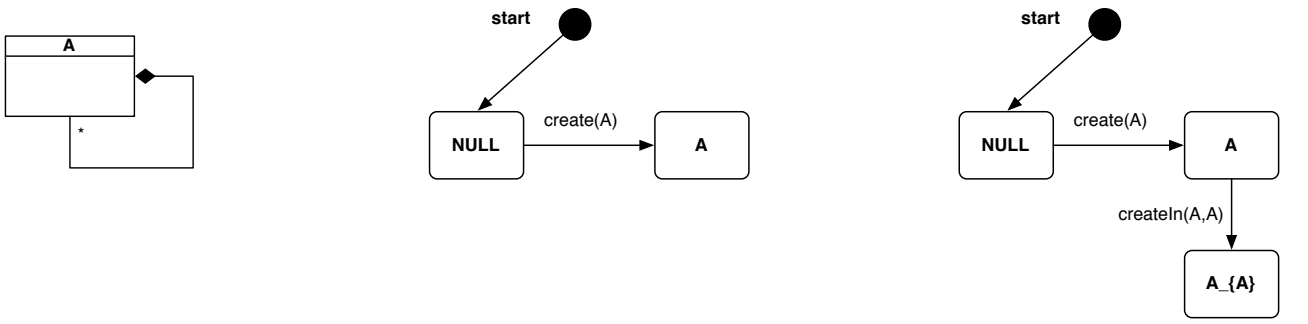
1. Given a metamodel, how hard is it to create an instance of every potential object in the metamodel?
2. Given a metamodel, what is the minimum set of elements required to create a *syntactically* valid model?
3. Given a modeling language (metamodel, plus semantic mapping) what is the minimum set of elements required to create a *semantically* valid model?
4. Given a modeling language, what is the complexity of creating a new syntactically *and* semantically valid model?

This paper focuses on (1) from this list, as this is a necessary and enabling step in order to tackle the remaining questions. A fundamental assumption of this paper is that an algorithmic metric for a domain-specific language is useful; the paper does not, however, claim that this is a sufficient (or even necessary) metric for a language. Complementary metrics for notation (including metrics for visual languages) should be considered alongside those in this paper; where this work is expected to have impact is in reducing the controllable complexity of a metamodel by revealing elements that increase complexity, so that the language designer may take some action. Work has been done in the characterization of *constructed* models [14], which is different from characterizations of a modeling language (though it may be related to the minimum complexity of any constructed model). The work in this paper is also different from generating instance models for the purpose of testing [4], although the technique suggested in this paper could be an alternative for the graph grammar approach to operationalizing a metamodel developed in that work.

This paper presents an algorithm to generate a state model that describes the construction of a model through generative analysis based on the metamodel. The generated model presents the minimum necessary steps in order to instantiate at least one model from each appropriate element of the metamodel. By analyzing the generated state model's structure, it is now possible to define several metrics that characterize the usability of the metamodel. Although this approach is neither necessary nor sufficient to generate a semantically meaningful model, it lays the foundation for such an examination of a language, which can lead to the development of methods to reduce the complexity of languages for certain domains.

2. BACKGROUND

2.1 Metamodeling
The metamodel is a declarative (as opposed to operational) form of the syntax of a modeling language. That is, while it will constrain a model to obey the language's syntax rules, it does not prescribe how models can be created. This paper considers metamodel concepts such as *containment*, *inheritance*, and *association*, although concepts such as set



(a) A simple metamodel.

(b) All models instantiated, but unused containment association.

(c) All metamodel elements (including containment association) are used.

Figure 1: Trivial example showing necessary steps to exercise the metamodel. (a) A simple metamodel, with only one type, and one containment association. (b) In a naive approach, the initial condition, followed by an event to create a model of kind **A**. (c) This state model creates instances of **A**, but also exercises the containment association that **A** can contain **A** objects.

containment and model references (pointers) are trivial extensions. Issues such as exercising *language composition* relationships that are found in a metamodel, and necessary changes to attribute values of a model or association, are left as future work. Also left for future work is the consideration of any OCL or semantic constraints on model structure.

2.2 State Modeling

State models are behavioral models originally designed to consider reactive systems [8]; transitions between states are attributed by an event, guard, and action. When an event that labels a transition takes place, if the guard evaluates to true then the transition fires and any actions are taken. When parallel paths are permitted, fork and join formalisms are used to express that once a fork is taken, both paths must be fully executed before the join can be fired. In this work, states in a model represent instantaneous snapshots of an instantiated model: transitions are triggered by user events, with actions that create models or associations. With this design, there is no need for guard conditions on these transitions. In order to decrease the combinatorial expansion of states, this paper heavily utilize forks and joins to indicate where the order of creation of objects is unimportant. It will be shown that this structure immediately lends itself to rudimentary metrics for language complexity.

3. APPROACH

Although the goal of this paper is not to actually instantiate instance models from a metamodel, it provides metrics to judge how difficult this is to do for a particular language. The approach treats the model instantiation/creation process abstractly, using a state model as an encoding of a symbolic transition system. This provides an objective measure for the steps necessary to create a certain model, as well as whether the *order* in which those steps are taken is important. Each element of a metamodeling language is considered next, in order to show how this element maps onto the generated state model.

3.1 Initial Conditions

The initial condition of the state model is always that there are no existing models. Every state in the state model will be attributed by the set of metamodel concepts that have been created at that time. The "final" state of the state model will contain at least one of every model.

3.2 Creation of Models

In order to create a model described in a metamodel, the system must transition based on an event that uses that model as a parameter. This is shown in Fig. 1b, which is a first step of generative instantiation of Fig. 1a.

3.3 Creation of Containment Associations

Although Fig. 1b instantiates all of the models contained in the metamodel shown in Fig. 1a, the containment association is not yet instantiated. In order to do this, it is necessary to create another object of kind **A** *inside* the existing model of kind **A**. The state model that details this set of actions is shown in Fig. 1c. Note that the final model created by this state model contains *two* models of kind **A**, not just one. Thus, it is not the case that a fully exercised modeling language will contain one and only one of each type, but rather at least one of each type.

3.4 Creation of Associations

Associations in a metamodel have rolenames associated with the source and destination of the association, and typed associations have a bona fide class for the association, capable of attribute values. This paper does not consider necessary changes to the attributes of a class, so the approach is the same for typed and untyped associations.

The development of the approach of creating associations requires a more complex metamodel shown in Fig. 2. In this metamodel, it is possible to create two kinds of associations between models of type **Z**, namely **W** associations (that are between exposed ports of a **T** model) and **Y** associations (that are between models of type **S** and **Z**, while encapsulated by models of type **T**).

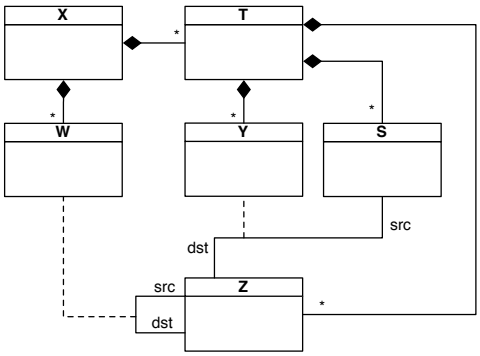


Figure 2: A more complex metamodel, exercising more of the capabilities of a metamodeling language.

In creating associations, the approach is to first instantiate the necessary models in order to have (at least) one of each model. At this time, it is possible to examine any new objects that the associations must create. In the generated state model, it is clear how the concurrent creation of objects results in a fork and join inside the state model, as shown in Fig. 3.

3.5 Inheritance Relations

Inheritance is used to reduce the complexity of a class diagram through generalization. If the goal of this paper were to generate models that test every potential association and containment association for a metamodel, then this would result in a combinatorial explosion of model creation (see [4]). However, the objective is to find the minimum steps to create one or more of each type. For abstract types, this is clearly not necessary, so if there are n subtypes of a model M, then the algorithm ensures that at least n of those models will be instantiated. Similarly, if associations may exist between superclass elements, it is necessary to create only one instance of that association.

4. RESULTS

Given the above algorithm with introductory examples, it is now possible to consider the results when applied to several metamodels that are gathered from the distribution of the Generic Modeling Environment [9]. Future work will import other well-established metamodels into GME, in order to calculate their metrics.

4.1 Nomenclature

Trajectories of the state models are expressed in a compact form, to provide room for several examples. In order to clarify this approach, the compact for for Fig. 3 is shown; recall, this is the progression of *states*, not the progression of events to create these states.

$$\emptyset, X, X_T, \{X_{T_S}, X_{T_Z}\}, X_{T_{SZ}}, X_{T_{SYZ}}, X_{T_{SYZW}} \quad (1)$$

For brevity, this is hierarchically refactored, where parentheses indicate containment, to achieve:

$$\emptyset, X, (T, (\{S, Z\}, SZ, SYZ), W) \quad (2)$$

Recall that the nomenclature $\{S, Z\}$ indicates a fork, that either state S or Z will occur first, and the other must occur next, before continuing on to the next state (join), which will include (SZ) in the list of owned models of the container. For brevity, the notation omits the final mention that all these objects will be contained, thus

$$\emptyset, M, (\{N, P\}, NP) = \emptyset, M, (\{N, P\}). \quad (3)$$

This permits a more succinct expression of the state model trajectory, without needing to specify the end result of a fork/join combination.

4.2 HFSM (Hierarchical Finite State Machine) metamodel

The HFSM metamodel is shown in Fig. 4a. This metamodel describes a language to express hierarchical finite state machines, as well as a series of events that will execute that finite state machine. This metamodel is interesting in that there is an implicit relationship between the text fields of **Transition** associations, and the names of **Events** models. This relationship complicates created models, but in the current approach attribute settings are not considered as an action that the user must perform in this minimal creation scheme. The trajectory of the state model generated from Fig. 4a is,

$$\begin{aligned} &\emptyset, \\ &\{State, (State, Transition), \\ &InputSequence, (Events, Sequence)\} \end{aligned} \quad (4)$$

Note that there is a fork in creating either a **State** or an **InputSequence**, but that creating (e.g.) a **State** inside the existing state *must* occur prior to creating the **Transition**, because that transition requires a **State** model to participate in the source and destination of that association.

4.3 SF (Signal Flow) metamodel

The SF metamodel is shown in Fig. 4b. The trajectory of the state model that is generated from this metamodel is,

$$\begin{aligned} &\emptyset, Folder, \\ &(\{Folder, \\ &Compound, \\ &(\{Compound, InputParam, Param, OutputParam, \\ &InputSignal, OutputSignal, \\ &\{DataflowConn, ParameterConn\}\}, \\ &Primitive\}). \end{aligned} \quad (5)$$

Note that the metamodel includes a containment association stating that the **Compound** type can contain the abstract class **Processing** (its superclass). It is arbitrary whether the **Compound** contains a **Compound** or a **Primitive**.

Readers familiar with the SF metamodel will note that the **DataflowConn** and **ParameterConn** associations are intended to utilize *module interconnection*, that is, the association belongs to the grandparent of the models that play a role in either end of the association. This is not currently accounted for in this work, but is planned for future implementation.

4.4 Rudimentary Complexity Measures

4.4.1 Complexity of Exercise

This metric is the number of instantiation *elements* that *must* be done in order (O for *order*), minus the number of

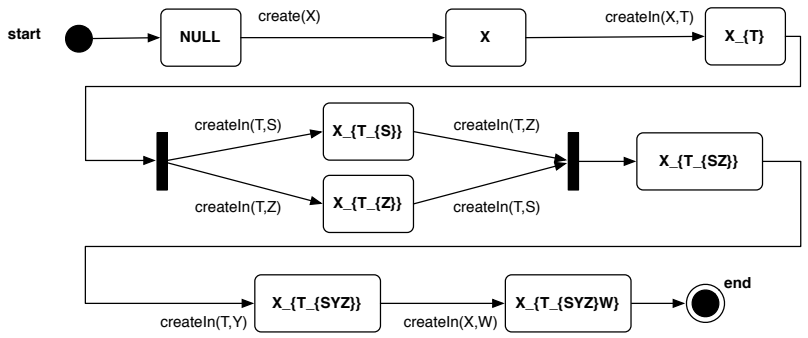


Figure 3: State model that instantiates minimal set of models to create necessary associations (from metamodel in Fig. 2).

instantiation groups that *must* be completed, but in any order (C, for *choice*). The higher this number, the more complex the modeling language is *if the goal is to create at least one of each model*: complexity of exercise, C_e.

$$C_e = O - C \tag{6}$$

In the case of hierarchy, all created objects inside the hierarchy count as one instantiation element, when ordered after their parent, whether or not those contained objects are required to be created in a particular order. For HFSN, $O = 8$ and is made up of the following elements:

- **State**, (**State**, **Transition**) (2)
- **State**, **Transition** (2)
- **InputSequence**, (**Events**, **Sequence**) (2)
- **Events**, **Sequence** (2)

The value for C_e is reduced by the choices available to the modeler, $C = 2$, itemized as:

- **State** (1)
- **InputSequence** (1)

This results in a $C_e = O - C = 6$, for the HFSM metamodel. For SF, C_e depends on the following elements, and $O = 6$

- **Folder**, (...) (2)
- **Folder**, **Compound**, (...), **Primitive** (4)

According to the calculation, $C = 2$:

- **Compound**, **InputParam**, **Param**, **OutputParam**, **InputSignal**, **OutputSignal** (1)
- **DataflowConn**, **ParameterConn** (1)

Thus $C_e = 4$ for the SF metamodel. Comparing this to the seminal work of McCabe's cyclomatic complexity [11], an analog of the number of independent paths through a program is the formulation of the ordered and unordered decisions required. Restated in terms of the metamodel, the metric calculates the number of ordered decision points (which correspond to a linear progression through McCabe's graph), and the number of unordered decisions (which correspond to loops in a software graph).

Intuitively, the previous examples roughly obey the metrics assigned to them: the HFSM metamodel depicted in Fig. 4a seems less complex than SF (Fig. 4b) when looking at the metamodels, but the HFSM metamodel requires more ordered actions when creating the model elements, so scores slightly higher when estimating its complexity. Although the SF metamodel has more elements than HFSM, many SF elements are taking part in generalization/specialization relationships, so the larger metamodel size is not indicative of increased complexity.

4.4.2 *Expansion of Cardinality*

Another metric of the complexity of a metamodel is whether in order to create at least one of each element, it is necessary to create more than one of some elements. In this metric, E_c is defined as the number of elements of the metamodel whose final cardinality is greater than one, when counted over the entire metamodel.

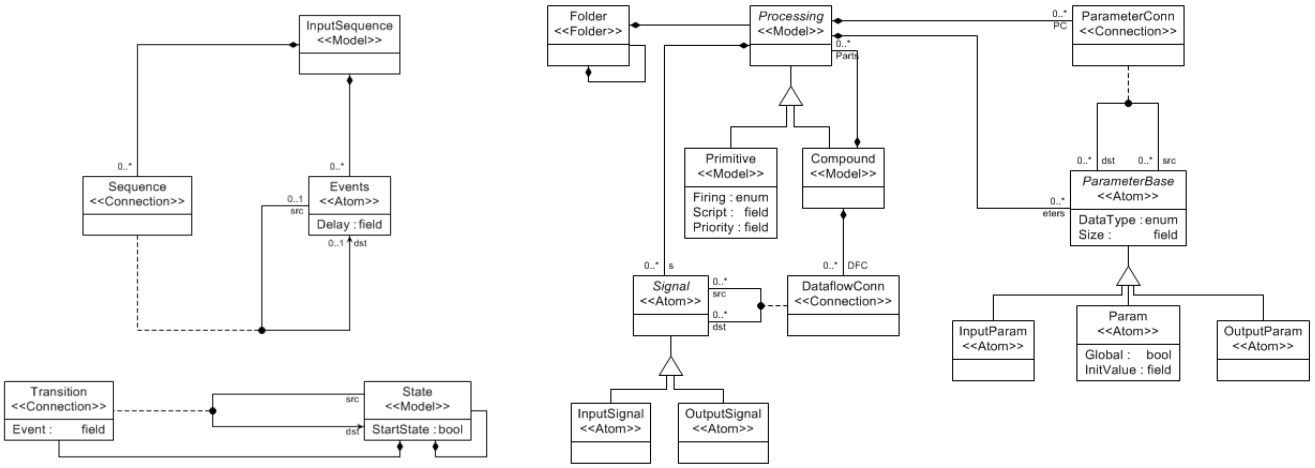
For the HFSM case, the $E_c = 1$, as **State** must be created twice. For the case of SF, the $E_c = 2$ (both **Folder** and **Compound** must be duplicated). An initial investigation of E_c indicates that the closer to 0, the more expressive the language is, but this bears further investigation.

5. RELATED WORK

It is important to distinguish the presented effort from the foundational and inspirational work upon which it is built.The work in this paper depends on the ability to create from a metamodel the necessary actions to create a model. Although this goal is discussed in detail in [4], that work addresses the construction of large, complex models, and uses graph grammars as the operational technique to create these models. Some analog to that approach could also be used to determine the minimum actions to create a single instance of each object, and is interesting as future research.

5.1 Software Metrics

McCabe's seminal paper on the cyclomatic complexity metric [11] measures complexity through the basis set of execu-



(a) The HFSM metamodel.

(b) The SF metamodel.

Figure 4: (a) The hierarchical finite state modeling metamodel. (b) The signal flow metamodel.

tion paths for a procedural program. Halstead recognized that issues of physical stability and dynamics could be relevant to software lifecycles [7]. Other software measures are implied based on properties such as coupling and cohesion. In practice, many overall metrics are a combined sum of heuristically weighted measures. In many of these these state of the practice metrics, weights are selected based on accepted or expert ordering of codebases, or based historically measured complexity (e.g., cost/time/performance overruns). Briand, Poels, and other [2, 17] present a software-independent set of mathematical definitions for the properties of size, length, complexity, cohesion, and coupling, in order to objectively compare various software metrics.

5.2 OO and Modeling Metrics

Pittman and Miller [16] adapt McCabe's cyclomatic complexity metric for a specific visual language: LabView's virtual instrument (VI) specification environment. In Belle et al. [1] several variants of modeling languages for a particular domain (enterprise modeling) are compared based on visualization of a common domain-specific feature: entity fanout. In that work, each modeling approach displayed a unique signature.

In many cases, the metrics applicable to a model are not consistent with the same metrics for software, since the important artifacts for analysis may be the generated artifacts (code, other models, etc.). Guerra et al. [6] discuss an approach to permit users to annotate the analysis of modeling artifacts in different semantic domains. As that work points out, such approaches must consider consistency throughout a toolchain in order for such analysis to be valid. The work by Martens et al. in [10] provides a reminder that the important metrics of a design may not be metrics of the language, or models built, but of the output artifacts (such as reliability, power, cost, etc.). In that work, these metrics are considered during automatic rewriting of the model to address costs measured in terms of domain-specific language elements.

The specific treatment of refactoring UML class diagrams, based on complexity measures of constructed models, is presented in the work of Ruhroth et al. [18], where a cycle of measure, diagnose, refactor, is suggested to reduce complexity metrics through structured refactoring, while provably maintaining the semantics of constructed models.

5.3 Metamodel-Based Methods

Eessaar [3] uses the metamodels of two similar languages to apply separately specified metrics (defined on models of the first language) to models built using the second language. This approach presents a way to reuse metrics that may have been peer-reviewed or critiqued in the literature, when the new models are in a similar, but not identical, syntactic and semantic context. McQuillan and Power [12] point out that the definition of model metrics is fundamentally a metamodeling activity; that when defined at the metamodel level, metric specification comes at a low cost; and that such an approach makes the development of new metrics trivial. Along with Eessaar [3], these authors mention that reuse of existing metrics is key to the validity of new metrics. Mens and Lanza [13] discuss extensions of graph-based metrics where a typed multigraph (to reflect the explicit nature of types in OO and metamodels) is the foundation of metric specification.

5.4 Language Usability and Expressivity

The Cognitive Dimensions framework [5] evaluates a programming language based on its structure, and provides a model in which to trade off design decisions for that language. These properties apply to a visual notation, and include the abstraction gradient, closeness of mapping, consistency, diffuseness/terseness, error-proneness, hidden dependencies, progressive evaluation, viscosity, and visibility. Such metrics are useful in conjunction with the metrics developed in this paper. The authors of [15] discuss several approaches to measuring the understandability of a particular notation. Future work extending from this paper may include the application of these measurement techniques to provide an additional objective measure to the metrics pre-

sented in this paper for a domain-specific language.

In [19] the authors discuss metrics for usability of a language-specification environment, and the languages that it generated, based on feedback from human subjects and objective measurements. The questions were focused on ease of use for a small language and various lifecycle questions about that language, including refactoring, and extensions to larger languages. Measurements focused on the time to specify the required portions of the language.

6. CONCLUSION

This paper presented an analog to the cyclomatic complexity of software: the complexity of using a metamodel. The preliminary graphs are built using rudimentary methods, and as these methods improve, the applicability of the metric will be more suitable for validation in experiments, or across bodies of models. Future work includes the development of *constraints* for the state model generator which will take form as *guards*, which permits advanced syntactic and semantic structural constraints to be taken into account, while reusing the state model generator. A logical next step is the consideration of attribute creation (i.e., whether it is necessary to produce the model), as any consistency of attribute relationships among models may dramatically increase complexity of creating valid models.

Given the ability to check constraints before leaving (for example) a model to create associations, it is possible to ensure that the appropriate number of objects, associations, etc., exist. This enables consideration of the number of guards on a state model as another level of complexity of the graph—another source of decision points. Generally, the more decision points, the more difficult it is to use a language, but more work is needed in order to determine the appropriate metric for this diffculty.

Acknowledgments

This work is supported by AFOSR under award #FA9550-091-0519, and NSF under awards CNS-0915010 and CNS-0930919. The author is grateful to the anonymous reviewers for their kind and insightful comments that led to improvements in the final version of this paper.

7. REFERENCES

[1] J.-P. Belle. Towards a syntactic signature for domain models: proposed descriptive metrics for visualizing the entity fan-out frequency distribution. *SAICSIT '02: Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology*, Sep 2002.

[2] L. Briand, S. Morasca, and V. Basili. Response to: Comments on "property-based software engineering measurement: Refining the additivity properties". *Software Engineering, IEEE Transactions on*, 23(3):196 – 197, 1997.

[3] E. Eessaar. On metamodel-based design of software metrics. *Balancing Agility and Formalism in Software Engineering*, 5082:40–54, Jan 2008.

[4] K. Ehrig, J. Küster, and G. Taentzer. Generating instance models from meta models. *Software and Systems Modeling*, 8(4):479–500, 2009.

[5] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131 – 174, 1996.

[6] E. Guerra, J. D. Lara, A. Malizia, and P. Díaz. Supporting user-oriented analysis for multi-view domain-specific visual languages. *Information and Software Technology*, Jan 2009.

[7] M. Halstead. Natural laws controlling algorithm structure? *SIGPLAN Notices*, 7(2), Feb 1972.

[8] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[9] Á. Lédeczi, Á. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *IEEE Computer*, 34(11):44–51, Nov 2001.

[10] A. Martens, H. Koziolek, S. Becker, and R. Reussner. Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms. *WOSP/SIPEW '10: Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*, Jan 2010.

[11] McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308 – 320, 1976.

[12] J. McQuillan and J. Power. On the application of software metrics to UML models. *MoDELS'06 Workshops*, 4364:217–226, 2006.

[13] T. Mens and M. Lanza. A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science*, 72(2):57–68, Jan 2002.

[14] M. Monperrus, J. Jézéquel, B. Baudry, and J. Champeau. Model-driven generative development of measurement software. *Software and Systems Modeling*, page Online First, 2010.

[15] S. Patig. A practical guide to testing the understandability of notations. *APCCM '08: Proceedings of the fifth on Asia-Pacific conference on conceptual modelling*, 79, Jan 2008.

[16] D. Pittman and J. Miller. Software metrics for non-textual programming languages. *AUTOTESTCON, '97*, pages 198 – 203, 1997.

[17] G. Poels, G. Dedene, L. Briand, S. Morasca, and V. Basili. Comments on "property-based software engineering measurement: refining the additivity properties". *Software Engineering, IEEE Transactions on*, 23(3):190 – 197, 1997.

[18] T. Ruhroth, H. Voigt, and H. Wehrheim. Measure, diagnose, refactor: A formal quality cycle for software models. *Software Engineering and Advanced Applications, 2009. SEAA '09. 35th Euromicro Conference on*, pages 360 – 367, 2009.

[19] C. Schmidt, B. Cramer, and U. Kastens. Usability evaluation of a system for implementation of visual languages. *Visual Languages and Human-Centric Computing, VL/HCC 2007. IEEE Symposium on*, pages 231 – 238, 2007.