Visual Specification of Metrics for Domain Specific Visual Languages

Esther Guerra, Paloma Díaz [1,2]

Computer Science Department
Universidad Carlos III
Madrid, Spain

Juan de Lara [3]

Polytechnic School
Universidad Autónoma
Madrid, Spain

Abstract

We present a Domain Specific Visual Language (DSVL) for the definition of metrics for other DSVLs. The metrics language has been defined using meta-modelling, and includes some of the more used types of product metrics. The goal is to make the definition of metrics for a DSVL easy, reducing or eliminating the necessity of coding. For this purpose, we rely on the use of visual patterns for the specification of the properties that should be measured in each metric type.

These ideas have been implemented in the AToM^3 tool, which allows the definition of DSVLs by means of meta-modelling. In this way, with the new extension, the DSVL designer is able to define a metrics suite for a DSVL. Then, an environment is generated where a number of widgets allow taking actual measures of the defined metrics on the models. We present some illustrative examples using the hypermedia design language Labyrinth.

Key words: Domain Specific Visual Languages, Metrics, Meta-Modelling, Graph Patterns, Code Generation.

1 Introduction

Diagramatic notations are pervasive in many software development activities. They are used in the planning, analysis and design phases as a means to

[1] Email:`eguerra@inf.uc3m.es`
[2] Email:`pdp@inf.uc3m.es`
[3] Email:`jdelara@uam.es`

specify, understand and reason about the system to be built. DSVLs are constrained diagramatic notations, oriented to a particular application domain. They provide high-level, powerful primitives, having the potential to increase the user productivity for the specific modelling task.

Measurement plays a central role in many engineering disciplines, such as electrical, mechanical and civil engineering [6]. However, traditionally, it has received little attention in the area of Software Engineering. The kind of entities that can be measured in this area include processes, resources and products [6]. In this paper we concentrate in the latter. Product metrics measure features of software systems (e.g. complexity, cohesion, coupling or maintainability) in order to control and improve their quality. One of the factors that may improve the use of metrics in industrial practice is their support by tools. Moreover, integrating a metrics tool in the early phases of the development can help to detect defects prior to implementation, saving time and budget. However, there is a proliferation of notations and tools that the software engineers use, and adapting and implementing metrics for them is a costly and time-consuming activity. Our goal is to provide a means to reduce such cost, by making the customization of metrics for any kind of DSVL easy (not only in the Software Engineering domain).

In this work, we propose using a DSVL (called *Metrics*) for the specification of a metrics suite for other DSVLs. *Metrics* has been defined through a meta-model which contains the main types of metrics we have identified. These include metrics for global model properties (such as number of cycles and size), single element features (e.g. methods of a class in object oriented languages), features of groups of elements (e.g. their similarity or coupling) and paths (e.g. hierarchies in object oriented languages, navigation paths in web design languages). The DSVL designer is able to customize these metrics by providing a visual pattern with the property to be measured. Visual patterns are graphical, declarative, user-friendly and intuitive. They have the advantage of saving the user the necessity of coding the metrics procedure and learning neither the API of the used tool nor the programming language in which the tool was coded. In addition, since no coding is required, it can help to minimize errors and reduce the development time, mainly when dealing with complex metrics. Nonetheless, the *Metrics* language also allows creating new metrics (different from the ones we provide) in a procedural way by coding in Python. In addition, it is possible to specify threshold values for the metrics. Thresholds may have an associated action, described either in Python or using a graph transformation system [13]. In this way, when a metric reaches one of its threshold values, the user is asked whether he wants to execute the action. This is useful if the action executes known design patterns or redesigns that improve the quality of the final product. For space constraints we concentrate only in the metrics aspect of the DSVL and leave out the discussion on actions.

These ideas have been newly implemented in the AToM^3 tool, and are illustrated with examples using the Labyrinth DSVL for hypermedia design [4]. The tool also gives support for the execution of metrics and report generation, without the necessity of coding.

2 A Taxonomy of Product Metrics

In this section we present a classification of the main types of product metrics. We have defined such a taxonomy by generalising metrics that are devoted to a specific language, notation or domain (like [2][6][11][15][16]) so that we provide an abstract metric language that is domain independent. We have distinguished four types of metrics:

- *Model-Oriented* metrics allow taking measures of the whole model. These include for example Mc Cabe's cyclomatic number (number of cycles), which is used in software engineering to measure the code complexity of a module [11].
- *Element-Oriented* metrics measure properties of individual elements in the model. In object oriented systems, these include the number of methods of a class.
- *Group-Oriented* metrics measure features of groups of elements in a system. For example, in object oriented notations they can provide an idea of the modularity (cohesion) of a system, by measuring the similarity between the different attributes and methods of classes [15].
- *Path-Oriented* metrics gather measurements involving paths between elements of the same type (or any of their subtypes). Thus, a path is made of instances of a certain type connected through some relation (which in fact can be a "complex" relation, made of several connected elements). For example, in web applications, a navigation path joins different pages by means of hyperlinks. Inheritance-related metrics can also be included here. In this group we find metrics for measuring the length of the path between two elements, or to detect start points of paths.

3 The *Metrics* Domain Specific Visual Language

We have created a DSVL for metrics specification using meta-modelling. The goal of the language is to be able to adapt to particular DSVLs some general predefined metrics (or create new ones) in an easy way. Fig. 1 shows its meta-model.

Abstract class *Metric* is the base class for all the metrics that the DSVL designer may create. It has a *name*, which must be unique. Attribute *variableTypes* indicates the domain of the metric (i.e. the types for which the metric is going to be calculated). For example, if the attribute contains the name of two types, the metric is calculated for each combination of one instance of the first type and another one of the second, resulting in a matrix.

If no type is specified, the domain of the metric is the whole model, and a scalar is obtained as a result. Attribute *subtypeMatching* specifies whether the objects in the domain must have exactly the type specified in the previous list (value *false*) or also its subtypes are allowed (value *true*). In addition, relation *dependency* allows a metric to use results calculated by others. A constraint in the meta-model forbids cycles of *dependency* relations.

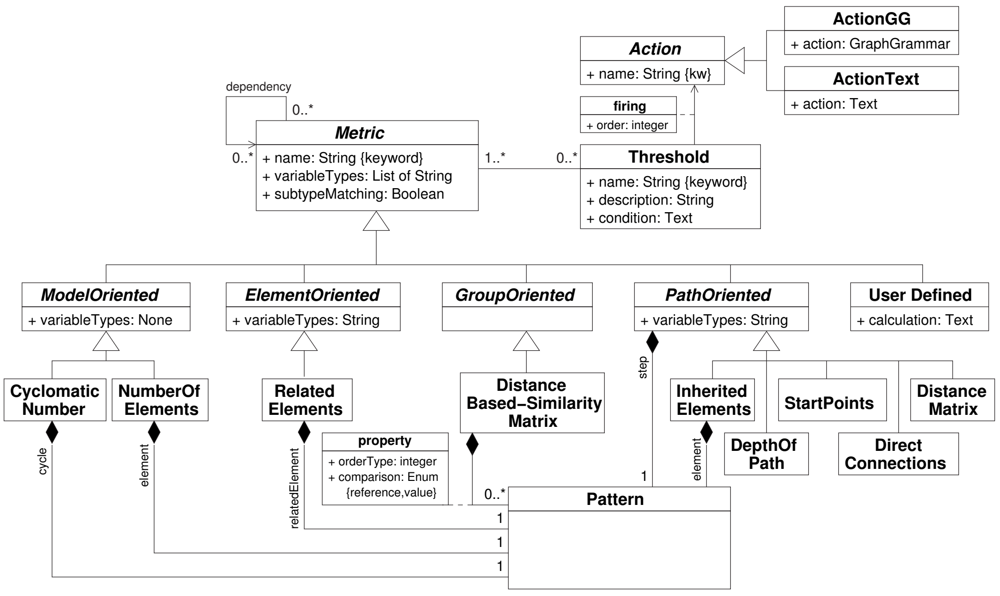


Fig. 1. Meta-Model for Metrics.

Thresholds can be associated to metrics, and contain a *name*, a *description* and a *condition*. The latter is a logical expression over metric values. Thresholds may have a number of associated actions that can be fired whenever the metric makes the threshold condition true. *Actions* can be described by means of procedural code (in Phyton), or by means of a graph transformation system. For space limitations, we leave out the discussion on actions and concentrate on metrics.

The four categories in our taxonomy of metrics are considered in the meta-model, all of them inherit from class *Metric*. Class *ModelOriented* and its children implement metrics of the first kind. The domain for the metric is not a single type of element, but the model itself. That is the reason why attribute *variableTypes* is empty. Our language contains two metrics of this kind. Metric *CyclomaticNumber* gives the number of cycles in a model. The user can customize what is considered a cycle by means of a pattern. This is made of a graph that should be found in the model, and additional graphs constraining the application of the pattern. We have used a similar approach to [5] for graph constraints. The structure of patterns is shown in Figure 2 and discussed in subsection 3.1. Metric *NumberOfElements* measures the number of elements of certain type in a model. The elements to measure are given as a pattern. In this way, we can constraint them (e.g. elements of some type that are not related to elements of some other type).

Class *ElementOriented* corresponds to the second subclassification in our taxonomy, that is, metrics for properties of single model elements. Therefore, only one type has to be specified in the domain. Subclass *RelatedElements* measures the number of elements of certain kind related to a given one. The way in which both are related is given as a pattern.

Class *GroupOriented* corresponds to the third subclassification in our taxonomy. We have included just one subclass, *DistanceBased-SimilarityMatrix*, which uses the formula for distance presented in [15]. In this way, if two objects x and y are to be compared, and assume that function $b(\cdot)$ returns the set of relevant properties for the comparison, function: $sim(x, y) = \frac{|b(x) \cap b(y)|}{|b(x) \cup b(y)|}$ gives the similarity between the two elements. The function returns a value in the $[0, 1]$ range. The lower the value, the less similar the two elements are. Then, $dist(x, y) = 1 - sim(x, y)$ gives the distance between the two elements. We have generalized this metric to an arbitrary number of elements of different or the same type. For each type, the set of properties to be measured (function $b(\cdot)$ in the previous formula) has to be specified. This is done with a pattern for each property and is modelled as a qualified relation between the subclass and the pattern. Attribute *orderType* in the relation specifies the type for which the pattern is given. In addition, the comparison can be made by reference (i.e. two objects are considered equal if they are the same), or by value (i.e. two objects are considered equal if all their fields have the same value).

Class *PathOriented* represents metrics of the fourth type in the taxonomy. Our DSVL allows customizing the type of the “node” in the path (attribute *variableTypes*), as well as the fundamental step (by means of a pattern). The result of metric *DistanceMatrix* is a matrix where each position (i, j) denotes the distance between element i and j (i.e. the number of steps to reach j starting from i). Metric *StartPoints* informs about the elements where a path begins (these are called base classes for the case of inheritance). Metric *Direct-Connections* measures the number of elements than can be directly reached in one step (e.g. the number of direct children for the case of inheritance). Metric *DepthOfPath* obtains the minimum number of steps that are necessary in order to reach an element starting from a start point (for inheritance this is the depth of inheritance tree). Finally, metric *InheritedElements* is applicable only for inheritance. It measures the number of elements of certain type that are inherited through an inheritance hierarchy. For example, the number of methods that a class inherits from its parent classes. In this metric the relation between the element in the path (e.g. class) and the element that is propagated (e.g. method) has to be given as a pattern.

A fifth metric called *UserDefined* has been added, so that DSVL designers can also define other domain specific metrics, different from the previous ones. The class has a field named *calculation* that allows the designer to include Python code to calculate the metric for a value in the domain. This code is encapsulated in a method that receives as parameters an instance of each of the types defined in the inherited field *variableTypes* and also the hosting model. The code should return a scalar value as a result of the calculation. In execution time, the method is consecutively invoked once for each value in the domain. Note that this is the only metric where the user has to code the metrics computing procedure, since in the previous ones, the use of patterns is enough for the customization (and subsequent execution) of the metrics.

3.1 Graph Patterns

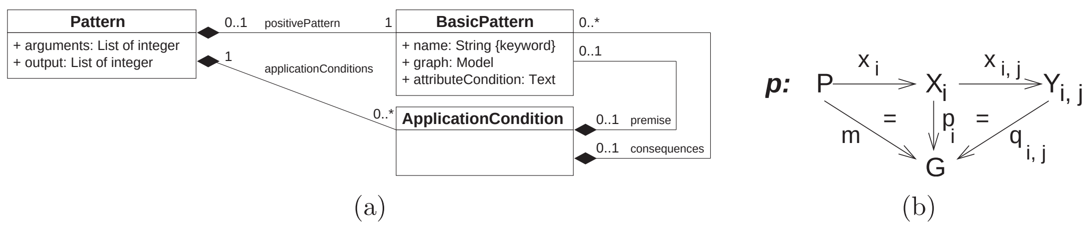


Fig. 2. (a) Meta-Model for Patterns and (b) Satisfaction of pattern p by Graph G.

Fig. 2 (a) shows the structure of a *pattern*. It is made of a positive graph condition, and a number of extra graph application conditions composed of a premise graph and a set of consequence graphs. In this way, in order for a pattern to be satisfied by a graph, an occurrence of the positive graph condition has to be found. Then, for each application condition, if the premise graph is found, some of the consequence graphs have to be found as well. The pattern can also be initialized with a partial match (whose elements are given by *arguments*) and produce some output (the elements in the positive graph condition identified by *output*). Note how a *BasicPattern* is made of a graph condition and an attribute condition, which is expressed in some textual language (Python in our case).

Formally, a pattern p is defined using a similar approach to [5] for application conditions, as $p = (P, \bigwedge_{i \in I}(x_i \Rightarrow \vee_{j \in J_i} x_{i,j}))$, where P is the main positive pattern (*positivePattern* in the meta-model) and $x_i : P \rightarrow X_i$ and $x_{i,j} : X_i \rightarrow Y_{i,j}$ are injective morphisms (X_i is the *premise* and $Y_{i,j}$ are the *consequences* in the meta-model). In this way, a graph G satisfies p (written $G \models p$), if a morphism $m : P \rightarrow G$ is found. In addition, if an x_i is specified and a morphism $p_i : X_i \rightarrow G$ is found, then some morphism $q_{i,j} : Y_{i,j} \rightarrow G$ must also be found, such that both triangles in Fig. 2 (b) commute. Technically, morphisms m, p_i and $q_{i,j}$ are clan-morphisms [1], as instances of abstract classes may appear in P, X_i and $Y_{i,j}$, which are mapped into instances of some class in their inheritance clan. We also require the typing of $Y_{i,j}$ be more concrete than the type of X_i, and this one more concrete than the type of P.

There are two special cases in the application conditions. If for some i no consequence graph is specified, then X_i is a negative application condition (NAC). On the other hand, if for some i, $P \cong X_i$ and $x_i = id$, then $Y_{i,j}$ (for

$j \in J_i$) are positive application conditions.

4 Implementation in AToM^3 and Example

AToM^3 [9] is a meta-modelling tool for the specification of multi-view DSVLs [7]. It has a generative approach, because starting from a meta-model, it generates an environment for the defined language. We have recently improved AToM^3 by adding a tool for the specification of metrics. In this way, the *Metrics* tool enrichs the generated environments for the DSVLs with the possibility to apply customized metrics to the models. This tool was created in AToM^3 itself, using the meta-model in Fig. 1. We completed the meta-model with some elements for the customization of the DSVL environments where the metrics are to be executed. In particular, we added an abstract class *UIButton* (with a single boolean attribute *button*) as the parent of classes *Metrics* and *Actions*. Attribute *button* is set to true if we want to generate a button to execute the metric or action in the DSVL environment. In addition, class *Metric* was provided with two additional attributes. The first one (*genReport*) is of type boolean and is selected in order to obtain a report in *pdf* format with the metric result. Finally, *report* is an enumerate type to select whether the report should show all the obtained values, or only the ones making some threshold condition true. From the *Metrics* meta-model, we used AToM^3's code-generating capabilities to obtain a tool for metrics specification. However, code had to be added by hand for metrics execution control and pattern matching.

Labyrinth [4] is a DSVL for the design of hypermedia and web applications. Hypermedia systems are described as a set of nodes where contents (text, images, etc.) are placed. Links establish the way in which users can navigate in the system. Besides, users can assume roles and belong to different teams from which they receive a set of permissions. Roles and teams can execute certain functions if a relation *permission* exists between them. Besides, roles and teams can be nested in hierarchical structures by means of relation *composition*.

The Ariadne Development Method [3] is based on Labyrinth to build hypermedia and web applications. It proposes a set of artefacts or diagrams that are views of the Labyrinth meta-model. We have used AToM^3 to develop an environment supporting the different Ariadne artefacts. The first step was defining the meta-model for Labyrinth, as it is shown in the background window in Fig. 3. More details regarding the definition of this multi-view DSVL can be found in [7]. Once the meta-model was created, nine different metrics were defined using the *Metrics* tool. This tool can be opened using the button labelled as "Metrics&Redesign" to the left of the window at the background. The *Metrics* tool is shown in the window labelled "2", and contains the specification of the metrics.

One of the defined metrics is called *Subject_Similarity*, and is of type

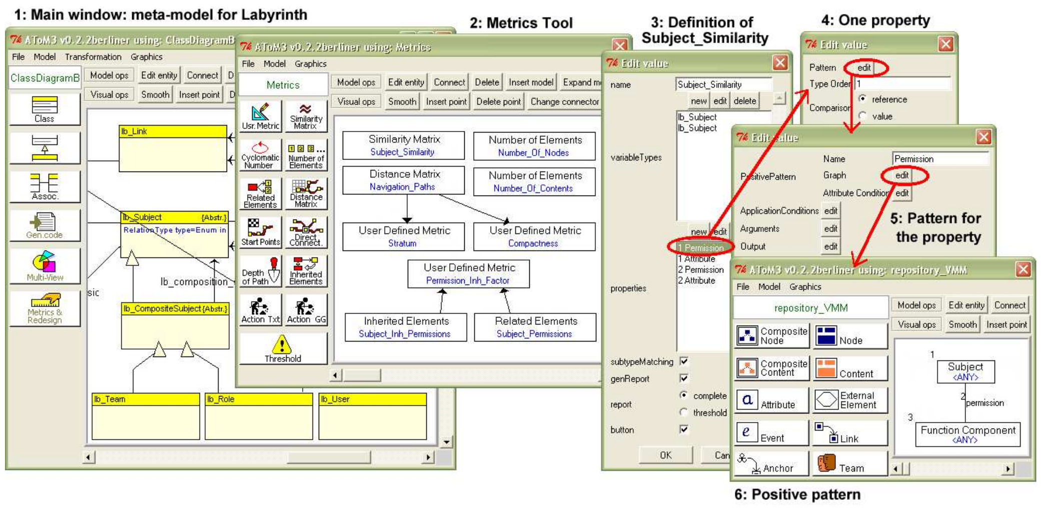


Fig. 3. Metrics Definition for the DSVL Labyrinth.

DistanceBased-SimilarityMatrix. The dialog box to the right (window 3) corresponds to the customization of the attributes for this metric. Attribute *variableTypes* contains type *lb_Subject* twice, as we want to compare subjects to subjects. As it is shown in the meta-model of the main window, a subject is an abstract class that has two concrete subclasses: *lb_Role* and *lb_Team*. In fact, we really want to compare subclasses *lb_Role* and *lb_Team*, because subject is abstract and never appears in any diagram. Therefore, the attribute *subtypeMatching* is checked. The list of properties for evaluating the similarity has also to be customized. In this case we take into account *permissions* and *attributes* for the comparison. This is the reason why the list *"properties"* in window 3 contains items *Permission* and *Attribute* twice (once for each subject). Window 4 shows the specification of the property *Permission*. The visual pattern that describes such property is shown in windows 5 and 6. In particular, window 6 shows the positive graph condition of the pattern. It collects the permissions of a certain subject for function execution. In this way, the access policy can be validated at design time. The argument of this pattern is the element labelled "1" (the subject to be compared) and the output is element "3" (the function). That is, the subject to be compared is passed as a partial match to the pattern, and all connected functions are returned as the result. Nonetheless, these details are hidden to the DSVL designer, who only has to specify the properties as patterns.

Metrics *Number_Of_Nodes* and *Number_Of_Contents* in window "2" are customizations of the model-oriented metric *NumberOfElements*. They count the number of *nodes* and *contents* in our system, providing a measure of its size. In both cases the pattern *element* simply contains an element of the type to be counted.

Metric *Navigation_Paths* is used to calculate the length of navigation paths. It is a customization of metric *DistanceMatrix*. This metric gives a measure of the minimum number of hyperlinks that a user has to navigate from a page to another one. It is very useful to detect isolated pages or pages of hard access. For this metric, *variableTypes* contains a node of information. Hyperlinks in Labyrinth are expressed with a class named *Link* connected to source and target classes *Anchor*, which in their turn are connected to the source and target *Nodes*. This navigation step has been easily expressed with a pattern, as it is shown in Fig. 4. The argument of the pattern is the element labelled "1", and the output is the one labelled "9". Thus, the target node of a navigation step (output) will be the source of the following step (argument). As before, the implementation details are hidden to the DSVL designer, who only has to specify the navigation step as a visual pattern, without coding.

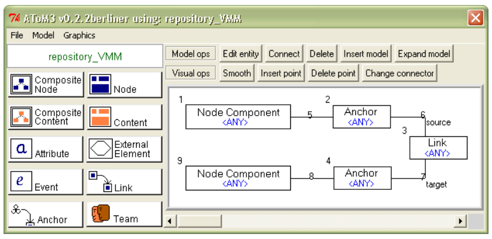
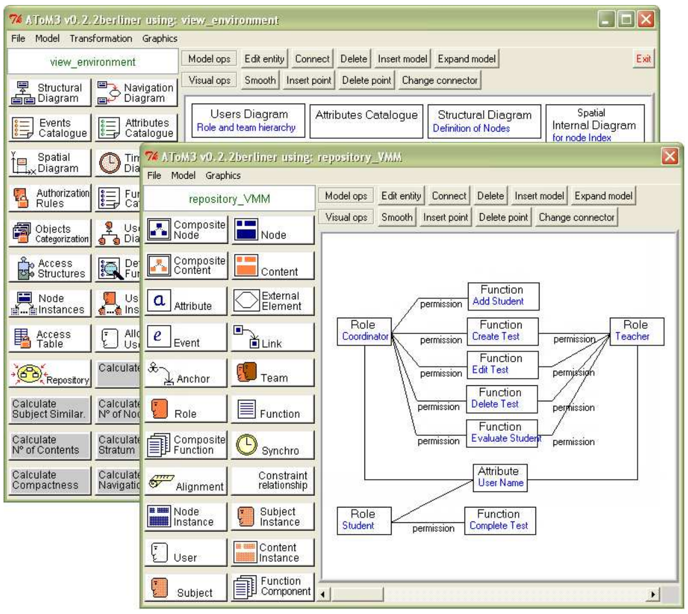


Fig. 4. Pattern for the Specificacion of a Step in Metric *Navigation_Paths*.

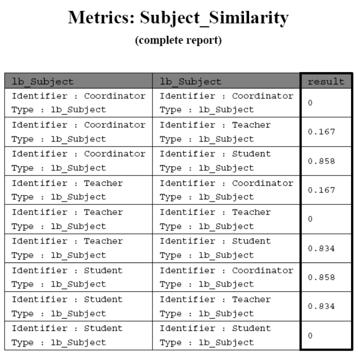
User defined metrics *Stratum* and *Compactness* [2] are oriented to the hypermedia domain. The first one is a measure of the linearity of the navigation path and may take values between 0 and 1. The lower the value, the less linear is the path. *Compactness* is a measure of the degree of connectivity of the navigation graph and also takes values between 0 and 1. The lower the value the less connected is the graph. Both metrics are based on the calculation of a distance matrix using the length of the paths between two nodes. This is the reason of the dependency relations between these two metrics and the distance matrix *Navigation_Paths*.

Finally, we have defined three metrics that are generalizations of existing metrics in the object oriented domain. Metric *Permission_Inh_Factor* (PIF) calculates the inherited permission ratio, being an indicator of the reuse. It is a particularization of metrics Method and Attribute Inheritance Factor (MIF and AIF respectively) in the object oriented domain [16]. It is the sum of all the permissions inherited by subjects (roles and teams) divided by the total number of defined permissions (locals and inherited). We have defined auxiliary metrics *Subject_Inh_Permissions* and *Subject_Permissions* to calculate the factors of this division. The first one is a customization of the path-oriented metric *InheritedElements*, and the second one of the element-oriented metric *RelatedElements*. Then, the PIF metric can be calculated from them using a couple of dependency relations. For the two auxiliary metrics no button is generated in the final environment (attribute *button* is set to false).

Fig. 5 (a) shows the environment generated for Labyrinth from the previous definition. In the window at the background, the buttons in white allow the creation of new instances of the Ariadne artefacts. A new artefact is represented as a box in the window canvas, which later can be edited to



(a) Generated Environment



Metrics: Subject_Similarity

(complete report)

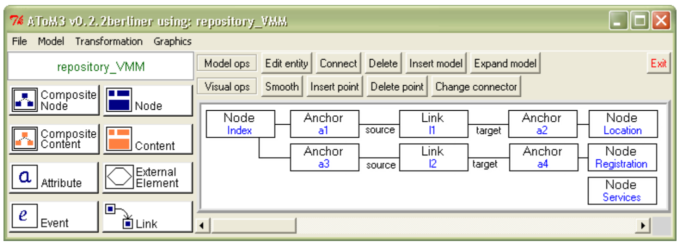
lb_Subject	lb_Subject	result
Identifier : Coordinator Type : lb_Subject	Identifier : Coordinator Type : lb_Subject	0
Identifier : Coordinator Type : lb_Subject	Identifier : Teacher Type : lb_Subject	0.167
Identifier : Coordinator Type : lb_Subject	Identifier : Student Type : lb_Subject	0.858
Identifier : Teacher Type : lb_Subject	Identifier : Coordinator Type : lb_Subject	0.167
Identifier : Teacher Type : lb_Subject	Identifier : Teacher Type : lb_Subject	0
Identifier : Teacher Type : lb_Subject	Identifier : Student Type : lb_Subject	0.834
Identifier : Student Type : lb_Subject	Identifier : Coordinator Type : lb_Subject	0.858
Identifier : Student Type : lb_Subject	Identifier : Teacher Type : lb_Subject	0.834
Identifier : Student Type : lb_Subject	Identifier : Student Type : lb_Subject	0

(b) Generated Report

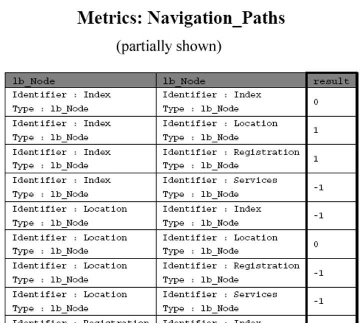
Fig. 5. Calculation of Metric *Subject_Similarity* on a Model.

include the model. The buttons in grey allow executing the previously defined metrics on the current models. The execution of a metric on a model generates a *pdf* document where the result is shown as a table. For example, Fig. 5 (b) shows the document obtained after executing metric *Subject_Similarity* on the model shown in the foreground window to the left. It can be observed that roles *Coordinator* and *Teacher* are quite similar (distance of 0.167) because they share four functions and one attribute. On the other hand, these two roles are quite different from role *Student*. The application designer could use these results to improve the design by adding a parent role common to both *Coordinator* and *Teacher*, and pulling up the common properties. This could be done using the actions in our *Metrics* DSVL. Note also how sometimes (as in the present case), metrics are not taken on isolated diagrams (which may only contain partial information), but on a *repository* model (see [7]), which contains the union of all the diagrams created by the user.

Fig. 6 shows another sample of metrics execution. To the right it is partially shown the report generated by the execution of metric *Navigation_Paths* on the model to the left. The report shows the minimum number of steps to reach a node from another one. A number of steps equals to -1 indicates that the second node is not reachable from the first one. Node *Services* is isolated since it has a distance -1 from and to any other node.



(a) A Model



Metrics: Navigation_Paths
(partially shown)

lb_Node	lb_Node	result
Identifier : Index Type : lb_Node	Identifier : Index Type : lb_Node	0
Identifier : Index Type : lb_Node	Identifier : Location Type : lb_Node	1
Identifier : Index Type : lb_Node	Identifier : Registration Type : lb_Node	1
Identifier : Index Type : lb_Node	Identifier : Services Type : lb_Node	-1
Identifier : Location Type : lb_Node	Identifier : Index Type : lb_Node	-1
Identifier : Location Type : lb_Node	Identifier : Location Type : lb_Node	0
Identifier : Location Type : lb_Node	Identifier : Registration Type : lb_Node	-1
Identifier : Location Type : lb_Node	Identifier : Services Type : lb_Node	-1
Identifier : Registration	Identifier : Index	

(b) Generated Report

Fig. 6. Calculation of Metric *Navigation_Path* on a Model.

5 Conclusions and Future work

In this work, we have presented a taxonomy of product metrics, together with a DSVL (called *Metrics*) for specifying metrics for other DSVLs. Our language makes easy the customization of metrics by means of graph patterns. We have implemented these concepts in the meta-modelling tool AToM^3 and shown some examples in the hypermedia domain. The example showed how the use of metrics is especially interesting in the early phases of development in order to improve the design or detect quality defects prior to implementation.

There are a variety of tools which incorporate functionalities for obtaining metrics. Some of them are for the implementation phase [8], and some others for the analysis and design phases [14]. Nonetheless, the set of metrics they provide is usually hard-coded and the possibilities of extension are very limited. One exception is the *SDMetric* tool [14], which allows the definition of metrics for UML using a relational-like language based on XML. Our approach is more general, as we are not restricted to UML, but we can define metrics for any DSVL. In addition, our *Metrics* language is visual, allowing the customization of metrics in a graphical and declarative way. In the area of meta-CASE tools, our work is also original. There is a plethora of this kind of tools (such as GME [10] or MetaEdit+ [12]), but to our knowledge none of them support the definition of metrics.

The presented meta-model is complete in the sense that it is possible to define any metric different from the already generalized ones by using the *UserDefined* metric. Nonetheless, we are currently working in generalizing additional metrics, and on their application to the hypermedia domain. We are also working in general analysis techniques for multi-view DSVLs.

Acknowledgements: This work has been partially supported by the Spanish Ministry of Education and Science with projects MD2 (TIC200303654) and MOSAIC (TSI2005-08225-C07-06).

References

[1] Bardohl, R., Ehrig, H., de Lara, J. and Taentzer, G. 2004. *Integrating Meta-Modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation.* FASE. LNCS 2984, pp.: 214-228. Springer.

[2] Botafogo, R. A., Rivlin, E., and Shneiderman, B. 1992. *Structural analysis of hypertexts: identifying hierarchies and useful metrics.* In ACM Trans. Inf. Syst., 10(2):142–180.

[3] Díaz, P., Montero, S., Aedo, I. 2005. *Modeling hypermedia and web applications: the Ariadne Development Method.* Information Systems, Vol 30(8). pp.: 649-673.

[4] Díaz, P., Aedo, I., Panetsos, F. 2001. *Modeling the Dynamic Behavior of Hypermedia Applications.* IEEE Trans. on Soft. Eng., 27 (6). pp.: 550-572.

[5] Ehrig, H., Ehrig, K., Habel, A., Pennemann, K.-H. 2004. *Constraints and Application Conditions: From Graphs to High-Level Structures.* ICGT'04, LNCS 3256, pp.: 287-303. Springer.

[6] Fenton, N. E., Pfleeger, S. L. 1998. *Software Metrics: A Rigorous and Practical Approach (2nd edition).* PWS.

[7] Guerra, E., Díaz, P., de Lara, J. 2005. *A Formal Approach to the Generation of Visual Language Environments Supporting Multiple Views.* Proc. of IEEE VL/HCC'05, Dallas, USA, pp.: 284-286.

[8] Jmetric home page: `http://www.it.swin.edu.au/projects/jmetric`

[9] de Lara, J., Vangheluwe, H. 2002. *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling.* FASE'02, LNCS 2306, pp.: 174-188. Springer.

[10] Lédczi, A., Bakay, A., Marói, M., Vögyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G. 2001. *Composing Domain-Specific Design Environments.* IEEE Computer. pp.: 44-51.

[11] McCabe, T. J. 1976. *A complexity measure*, IEEE Transactions on Software Engineering SE-2, 308–319.

[12] Pohjonen, R., Tolvanen, J-P. 2002. *Automated Production of Family Members: Lessons Learned.* Proc. of PLEES'02, Seattle, USA. pp.: 49-57.

[13] Rozenberg, G. (ed). 1999. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol 1.* World Scientific.

[14] SDMetric home page: `http://www.sdmetrics.com`

[15] Simon, F., Löffler, S., Lewerentz, C. 1999. *Distance Based Cohesion Measuring.* Proc. 2nd European Software Measurement Conference, pp.: 69-83.

[16] Vázquez, P.J., Moreno, M. N., García, F. J. 2001. *Métricas Orientadas a Objetos.* (In Spanish). Technical Report DPTOIA-IT-2001-02. University of Salamanca.