ARTICLE IN PRESS

Available online at www.sciencedirect.com

ScienceDirect

Journal of Visual Languages and Computing ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Journal of Visual Languages & Computing

www.elsevier.com/locate/jvlc

Visual specification of measurements and redesigns for domain specific visual languages

Esther Guerra[a,*], Juan de Lara[b], Paloma Díaz[a]

[a]Computer Science Department, Universidad Carlos III de Madrid, Avda. Universidad 30, 28911 Leganés, Madrid, Spain
[b]Computer Science Department, Universidad Autónoma de Madrid, Campus Cantoblanco, 28049 Madrid, Spain

Received 18 May 2007; accepted 25 September 2007

Abstract

Ensuring model quality is a key success factor in many computer science areas, and becomes crucial in recent software engineering paradigms like the one proposed by model-driven software development. Tool support for measurements and redesigns becomes essential to help developers improve the quality of their models. However, developing such helper tools for the wide variety of (frequently domain specific) visual notations used by software engineers is a hard and repetitive task that does not take advantage from previous developments, thus being frequently forgotten.

In this paper we present our approach for the visual specification of measurements and redesigns for Domain Specific Visual Languages (DSVLs). With this purpose, we introduce a novel DSVL called *SLAMMER* that contains generalisations of some of the more used types of internal product measurements and redesigns. The goal is to facilitate the task of defining measurements and redesigns for any DSVL, as well as the generation of tools from such specification reducing or eliminating the necessity of coding. We rely on the use of visual patterns for the specification of the relevant elements for each measurement and redesign type. In addition, SLAMMER allows the specification of redesigns either procedurally or by means of graph transformation rules. These redesigns can be triggered when the measurements reach a certain threshold.

These concepts have been implemented in the meta-modelling tool AToM^3. In this way, when a DSVL is designed, it is possible to specify measurements and redesigns that will become available in the final modelling environment generated for the language. As an example, we show a case study in the web modelling domain.

Keywords: Domain specific visual language; Meta-modelling; Measurement; Redesign; Graph transformation; Graphical pattern

1. Introduction

Diagrammatic notations are pervasive in many software development activities. They are used in the planning, analysis and design phases as a means to specify, understand and reason about the system to be built. The used notations range from general-purpose languages (such as UML for analysis and design of object oriented systems) to Domain Specific Visual Languages (DSVLs) oriented to a particular application domain (such as [1–3]). DSVLs provide high-level, powerful primitives, having the potential to increase the user productivity for the specific modelling task and the quality of

*Corresponding author. Tel.: +34 916249419.
E-mail addresses: eguerra@inf.uc3m.es (E. Guerra), jdelara@uam.es (J. de Lara), pdp@inf.uc3m.es (P. Díaz).

1045-926X/$ - see front matter © 2007 Elsevier Ltd. All rights reserved.
doi:10.1016/j.jvlc.2007.09.002

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

the systems that make use of them. They are usually less error-prone and easier to learn than general-purpose languages because the semantic gap between the user's mental model and the DSVL model is smaller. Moreover, the use of DSVLs is central to recent paradigms for software development such as the Model-Driven Development (MDD) [4]. In this paradigm, models (often expressed using a DSVL) are the primary asset, from which code is automatically generated. For this reason techniques for ensuring model quality before code generation become essential.

A usual method to quantify, control and assure the quality of systems in many engineering disciplines is measurement [5]. The kind of entities that can be measured in this area include processes, resources, products [5] and projects [6]. In this paper we concentrate on product measurements that quantify features of system designs (e.g. size, complexity, cohesion, coupling or maintainability). One of the factors that may improve the use of measurement in industrial practice is their support by software tools, which is essential in automation-based processes such as the one proposed by MDD. Moreover, having a measurement tool in the early phases of the development can help detecting defects prior to implementation, saving time and budget. However, there is a proliferation of notations that the software engineers use to model their systems, and adapting and implementing specific measurement tools for each one of them is a costly and time-consuming activity. Our goal is to provide a means to reduce such cost by increasing the level of abstraction of measurements by making them language independent and facilitating their customisation for any DSVL (not only in the Software Engineering domain). This objective has clearly an impact in both the MDD and the Visual Languages communities.

A different technique used not as much as to quantify but to improve model quality is model redesign, a process in which models are reworked in order to improve their overall quality (e.g. performance, reusability, genericity) or to include best practices identified in the corresponding discipline [7]. Model refactorings are a kind of model redesigns that in addition preserve the functionality. Recent proposals try to relate design patterns [7] and model refactorings by applying pattern-directed refactorings [8]. Again, the proliferation of notations can hamper the application of redesigns. The availability of high-level, abstract redesigns easily customisable for particular DSVLs can alleviate this problem.

In this paper, we propose a novel approach for the specification of measurements and redesigns oriented to DSVLs. The purpose is two-fold: on the one hand, reducing the cost of defining domain specific measurements and redesigns, and on the other hand, being able to automatically generate domain specific measurement and redesign tools for any DSVL starting from these high-level specifications. Therefore, this work is targeted to the Visual Language community as well as the Software Engineering community, and in particular to the MDD area. The proposal is based on the use of a novel DSVL called *SLAMMER* (Specification LAnguage for Modelling MEasurements and Redesigns) that allows the customisation of generic measurements and redesigns for a given DSVL. It has been defined through a meta-model that contains generalisations of some of the main types of measurements. These include measurements 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). SLAMMER allows customising these generic measurements by using visual patterns, creating new ones (procedurally), and composing them in order to build more complex measurements. In addition, it is possible to specify threshold values for the measurements that may have an associated action described either procedurally, by customising a generic action template or by using a graph transformation system [9]. This is useful if the action performs a redesign that improves the quality of the model or modifies it towards known design patterns.

These ideas have been implemented in the meta-modelling tool AToM^3. In this way, the DSVL designer can enrich the DSVL specification with a SLAMMER model specifying a number of measurements and redesigns for it. Starting from this definition, a modelling environment is generated for the language that integrates the measurements and actions previously defined.

The paper is organised as follows. Section 2 gives an overview of the main concepts of measurements and redesigns. Next, Section 3 presents SLAMMER, while its use is demonstrated in Section 4 by defining a set of measurements and actions for

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

Labyrinth [10], a DSVL used for web design. Section 5 explains how we have implemented the framework in the meta-modelling tool AToM^3, so that starting from a SLAMMER model it is possible to automatically generate support tools to measure and redesign models. The section includes an example where a visual editor with an integrated measurement and redesign tool is generated for Labyrinth and its use is illustrated. Section 6 compares with related work and, finally, Section 7 ends with the conclusions and proposes lines for future work.

2. Measurements and redesigns for DSVLs

2.1. Software measurement

According to [6,11], engineering disciplines require measurement mechanisms in order to provide feedback and assist in evaluation, thus creating a corporate memory and helping in the answering of questions about the object being measured. In software engineering, the measurable objects can be processes, resources, products [5] and projects [6]. In this work we focus on products and, more specifically, in models.

Software measurements[1] are enablers for obtaining quality products. They are used, for example:

- In forward engineering, in order to pinpoint anomalies and estimate the cost and effort of building software products. In particular, in MDD, models are active entities from which code is generated; therefore, quality should be assured at the model level. Our work is oriented towards quantifying model quality.
- In software reengineering, in order to acquire a basic understanding of the software, providing higher-level views, and finding violations of good design practices.
- In software evolution, in order to identify stable and unstable parts of the software, locating where refactorings and redesigns should be applied or have been applied, and identifying variations in the software quality.

Products (and in general measurable objects) contain *internal* and *external* attributes. The former can be measured in terms of the product itself. For example, if we are measuring code, its size is an internal attribute. *External* attributes can only be measured with respect to how the product relates to its environment [6]. In the case of code, the cognitive complexity, maintainability and usability are external attributes. They are obtained by testing, operating and observing the executable software. Our framework is directed to internal attribute measurement, as we want to quantify the quality of the system models from which the system itself is built.

Measurements can be *direct* or *indirect*. In the first case, the value is derived from an attribute that does not depend upon any other measure (sometimes they are also called *base measures* [12]). *Indirect* (or derived) measures are obtained by combining several direct or indirect measures. The term *indicator* is sometimes used to refer to indirect measures which have an associated analysis model made of a calculation procedure plus some decision criteria. The criteria can be thresholds, targets or patterns used to determine the need for action or further investigation [12]. As we will show in next section, our approach supports direct and indirect measures, as well as indicators with thresholds that indicate anomalies in the measurement values and may trigger redesigns for improving the quality of the model.

Further classifications of measurement methods include the *objectivity*, that is, whether they involve the human (subjective) judgment, or they are quantifications based on numerical (objective) rules. Finally, regarding the *automation* degree, measurement methods can be automatic, semi-automatic our manual. Our approach is aimed at the automation of the measurement process, and for this reason we only consider objective measurements (as subjective measures cannot be made fully automatic).

2.2. Redesigns

Redesigns are changes in a design model in order to improve some quality attribute, such as its understandability, performance, cohesion or coupling. When the redesign preserves the intended meaning of the model, it is called a model refactoring [13]. Refactorings [14] were originally defined as changes made to the software code in order to make it easier to understand and maintain without changing its observable behaviour. Model refactoring shifts refactoring techniques from code

[1]Although the term "software metric" is widely used in the software engineering community, we prefer the term "software measure" as, according to [12] and formally speaking, a metric is a function that measures the distance between two entities.

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

to models, which are higher-level representations of the system. In model-driven approaches this is the right abstraction level, as developers usually work with models (often specified using DSVLs) and code is automatically generated from them. Thus, model refactoring is just a kind of model redesign that preserves behaviour.

The need for performing refactorings and redesigns is frequently detected through so-called "Bad smells" [14]. They informally describe some design or code problem, and propose a number of refactorings to help in its solution. Some efforts have been recently oriented towards formally defining such *smells* through the use of metrics [15]. In our proposal we follow a similar approach by associating thresholds to measurements in order to detect extreme values that can fire redesigns.

3. SLAMMER: Specification LAnguage for Modelling MEasurements and Redesigns

This section presents SLAMMER, a DSVL oriented to the specification of measurements and redesigns to be applied on models conforming to a meta-model. In Section 3.1 we describe the part dealing with measurements, while in Section 3.2 we introduce the part describing actions and redesigns.

3.1. Measurements

A measurement for a certain set of entities is defined as a measurement method and a scale [16]. Thus, for the definition of a measurement, we need to specify the entities that are going to be characterised (the domain), the relevant attributes for the measurement method, the measurement method itself (which in the case of indirect measurements is a function that uses values calculated by other measurements), and the scale (the range of values it can take). The scale can be of type nominal, ordinal, absolute, interval or ratio [6]. For the last two types of scale, a measurement unit may also be specified (e.g. number of classes, lines of code). In addition, measurements may include information about normal or unusual values that point to threshold values in the measurement scale.

For example, the distance based-similarity matrix measurement [17] calculates how similar two entities are by studying the set of attributes they share. Let x and y be the entities to compare, and assume that function $b(\cdot)$ returns the set of attributes of an entity. Then, the formula: $dist(x,y) = 1 - |b(x) \cap b(y)|/|b(x) \cup b(y)|$ gives the distance between the two entities in the scale $[0,1]$. The lower the value, the more similar the entities are. Suppose we have a DSVL that allows specifying user roles and grant permissions to these roles. We can use this measure to analyse how similar each two roles are and thus detect redundancies. In order to adapt this measurement for this particular DSVL, we should give the measurement domain (each two roles in the system), the set of attributes that make two roles similar (the permissions a role holds), the measurement method (formula $dist(x,y)$) and the scale (real numbers in the interval $[0,1]$). In addition we may define as threshold all values lower than, say 0.1, in order to indicate that the measured roles are too similar and can be joined. Specifying the unit would not be necessary in this case.

Note how the measurement function and scale are language independent and do not change when the measurement is used in different DSVLs. On the contrary, the domain, the entity attributes, the units and the threshold values are language dependent and have to be specified for each particular DSVL. SLAMMER is based on this fact to define a set of predefined generic measurements that hide the measurement function and that can be customised by providing only the domain specific information. The measurement domain is specified as a list of types and built as each possible combination of the instances of the types; the attributes to be measured are given by a set of patterns; the units are given textually; and the thresholds are specified as boolean conditions evaluated on the measurement value.

These concepts have been included in the SLAMMER language. Part of its meta-model (the package concerning measurements) is shown in Fig. 1. In order to define this language, we have taken into account related works on the definition of ontologies concerning software measurement [12,18], as well as on the international standard for software quality ISO 15939 [16]. Concepts that do not have an operational meaning (e.g. information need and quality model in [12]) have not been included in our language, as all elements in SLAMMER have a functional, operational use. Remember that the goal of SLAMMER is, on the one hand, to provide a visual notation to facilitate the definition of measurements and redesigns for a given DSVL and, on the other hand, being able to automatically generate tools from such specifications that allow applying the defined measurements

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

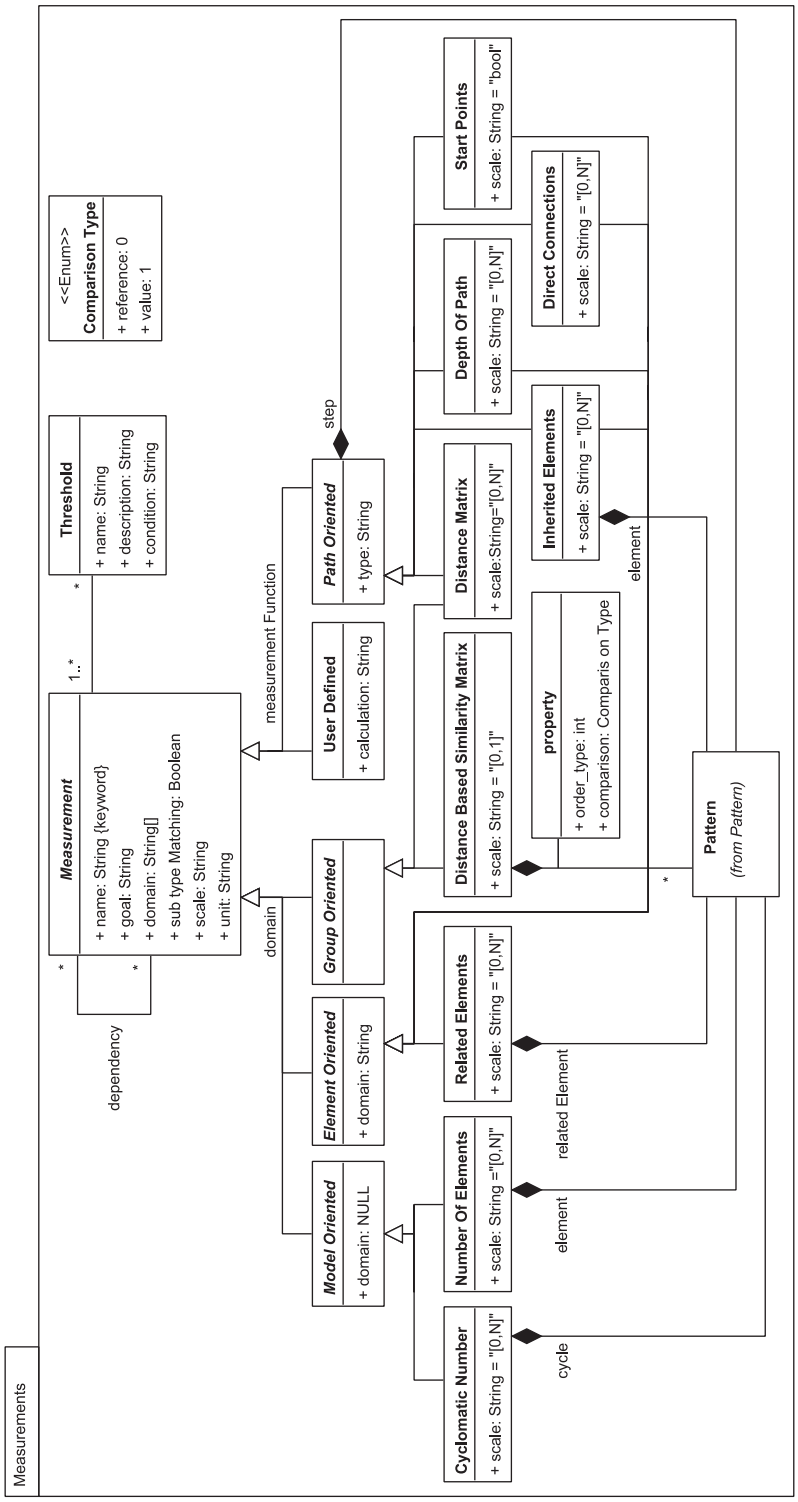

Fig. 1. The SLAMMER meta-model (measurements package).

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

and redesigns on the DSVL models for which they were defined.

The SLAMMER meta-model contains an abstract class *Measurement* which is the base class for all kinds of measures. It contains a unique *name* that identifies the measurement, as well as a *goal*. Attribute *domain* is used to specify the measurement domain as a list of types. In the previous example, the domain consisted on every possible combination of two roles, thus, this attribute should contain type role twice. Attribute *subtypeMatching* specifies if objects in the domain must have exactly the type specified in attribute *domain*, or if also any of its subtypes is allowed. This makes measurements more reusable, being defined once for a type, and used for all its subtypes. Attributes *scale* and *unit* are used to specify the range of values the measurement can take and its magnitude, respectively. In addition, relation *dependency* allows defining indirect measurements that use results calculated by other (direct or indirect) measurements. In this way, measurements can be reused and composed in order to build more complex ones. We have included a meta-model constraint that forbids cycles of recursive dependencies. Finally, a measurement can define any number of *threshold* values, which are extreme values for it. A threshold has a *name*, a *description* and a *condition*. The latter is a logical expression over values of the measure.

In the meta-model, measurements are subclassified depending either on the domain dimension (left branch in the measurement hierarchy with discriminator *domain*) or on the measurement mechanism used in order to calculate it (right branch with discriminator *measurementFunction*). In the former case, measurements can be *model-oriented* if the measure is taken on the whole model; *element-oriented* if they measure properties of an individual element; and *group-oriented* if they apply on groups of elements. For the second generalisation set (with discriminator *measurementFunction*), measurements can be *path-oriented* if they use a measurement function that traverses paths between elements of the same type (or any of their subtypes), which is specified by attribute *type*; and *user-defined* if the measurement designer is the one who provides the specific measurement function (attribute *calculation*). Concrete measurements in the meta-model can be classified from both points of view. See for example measurements *InheritedElements* and *DepthOfPath*, which are both element- and path-oriented.

Model-oriented measurements have the model as domain, and thus, attribute *domain* is set to *NULL*. SLAMMER defines two generic measurements of this kind: *CyclomaticNumber* and *NumberOfElements*. The first one calculates the number of cycles in a model, and thus it must be customised the basic step in the cycle by means of a pattern (which is a graph plus application conditions, see Section 3.1.1). Measurement *NumberOfElements* counts the number of elements that fulfill certain conditions in a model. The elements to count are given as a pattern, which allows constraining them (e.g. elements of some type that are not related to elements of some other type) and measure complex structures.

There are five element-oriented measurements, four of them being path-oriented as well. As they refer to properties of single model elements, attribute *domain* contains a single type. *RelatedElements* measures the number of elements of certain kind related to a given one. The way in which both elements are related is given as a pattern. The other four measurements will be explained below, together with the path-oriented ones.

The group-oriented measurement called *DistanceBased-SimilarityMatrix* uses the formula for distance presented in [17] (and shown at the beginning of this section), but generalised to an arbitrary number of elements of different or the same type, which are specified in attribute *domain*. Its scale is the interval $[0, 1]$. The higher the value, the less similar the compared elements are. For each type, the set of properties to be measured has to be specified. This is done with a pattern for each property, and modelled in the SLAMMER meta-model as a qualified relation *property* between the subclass and the pattern. Attribute *orderType* in the relation specifies the type of the list *domain* for which the pattern is given. In addition, the comparison can be made by reference (i.e. two objects are equal if they are the same) or by value (i.e. two objects are equal if all their fields have the same value).

Depending on the measurement function, measurements are classified as path-oriented or user-defined. For the first kind, SLAMMER allows customising the type of "node" in the path (attribute *type*), as well as the fundamental step (by means of a pattern). This can be a "composite" relation made of several nodes and edges. Attribute *domain* of each concrete path-oriented measurement is built by using the attribute *type*: if the measure is

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

element-oriented (e.g. *DirectConnections*), *domain* is equal to the type; if it is group-oriented (e.g. *DistanceMatrix*), the type is inserted into array *domain* as many times as necessary in order to obtain the correct domain dimension (e.g. twice for *DistanceMatrix*). The meta-model contains five path-oriented measures. Measurement *DistanceMatrix* returns a matrix where each position (i,j) denotes the distance between elements i and j (i.e. the number of steps to reach j starting from i). Measurement *StartPoints* informs about the elements in which a path begins (e.g. the base classes for the case of inheritance in object-oriented systems). This measurement is element-oriented as for each element it says whether it is a start point or not. Measurement *DirectConnections* calculates the number of elements than can be directly reached in one step (e.g. the number of direct children for the case of inheritance). *DepthOfPath* obtains the minimum number of steps that are necessary to reach an element starting from a start point (e.g. for inheritance this is the depth of inheritance tree). Finally, measurement *InheritedElements* is mainly applicable to inheritance hierarchies. It measures the number of elements of certain type that are inherited through an inheritance hierarchy (e.g. the number of methods that a class inherits from its ancestors). In this case, together with the path node type and fundamental step, the relation between the path node (e.g. class) and the element that is propagated (e.g. method) has to be given as a pattern.

Finally, class *UserDefined* allows specifying measurements that define additional domain specific measurement methods. The class has a field named *calculation* to include the procedural specification that calculates the measurement result 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 *domain*, and returns a value as the result of the calculation.

The fact of having such measurement categorisation explicitly in a meta-model makes the approach easily extendible. This is of high practical relevance, as the real implementation of the language (see Section 5) is based on this explicit meta-model and on a meta-modelling tool, allowing an easy maintenance of the set of measurements.

3.1.1. Graph patterns

As explained in previous subsection, patterns in SLAMMER are used in order to specify the model properties to be measured. The simplest form of pattern is made of a single positive graph, and its application to a model gives as result all occurrences of the positive graph in the model. The pattern can be initialised with a partial match, which is given as argument of the pattern, and the output can be filtered in order to return a subgraph of each positive graph occurrence. Left part of Fig. 2 shows an example pattern. The positive graph is made of objects *Role* and *Node* related through a relationship *PA*. To the right, the pattern is instantiated in graph G. In step (i) the match is initialised with role `r1`, which is received as a parameter. In step (ii), the match is extended to the complete positive graph. Two occurrences are found in G: one relates role `r1` to node `n1`, and another one relates it with node `n2`. In step (iii) the occurrences are filtered so that only the elements specified as output in the pattern are obtained as result. Thus, as the pattern specified element labelled "3" as the output, only nodes `n1` and `n2` in the matchings are given as result.

In SLAMMER we use patterns in order to customise measurements by specifying how relevant model attributes are expressed in a certain DSVL. The arguments of the pattern correspond to a value in the measurement domain, and the output are the

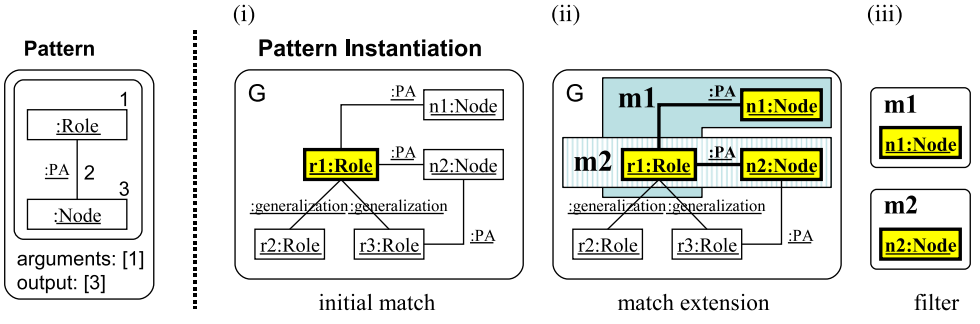

Fig. 2. Example of graph pattern and instantiation.

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

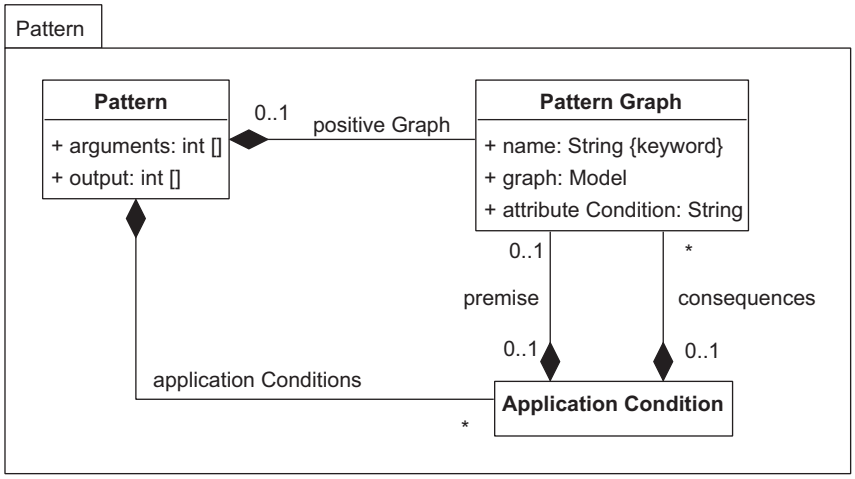

Fig. 3. The SLAMMER meta-model (pattern package).

model attributes we want to obtain. For example, the pattern in Fig. 2 can be used to customise measurement *RelatedElements* for a DSVL that defines nodes, roles and permission assignments. Such customisation counts the number of nodes that each role is allowed to access. The pattern is instantiated for each role in the model and, in each case, the measurement value is calculated as the number of times the pattern gets instantiated (two for role `r1`).

The structure of class *Pattern* is shown in Fig. 3. A pattern is made of a positive graph and has attributes *arguments* and *output*. A *PatternGraph* is made of a graph and an attribute condition that is expressed in some procedural language. In addition, we consider a more complex form of patterns than the one in the previous example, by considering application conditions that restrict the number of valid occurrences in the pattern instantiation process. An application condition is made of a premise graph and a set of consequence graphs. In this case, in order to instantiate the pattern, first all occurrences of the positive graph are found in the model; next, for each application condition, if the premise is found then some of the consequence graphs have also to be found for the occurrence to be valid. There are two special application conditions. If only a premise is specified and no consequence, then it is called a Negative Application Condition (NAC), and finding the premise in the model makes invalid the positive graph occurrence. On the other hand, if the premise is isomorphic to the positive graph and some consequence is specified, it is called a Positive Application Condition (PAC). In this case, some of the consequences have to be found in the model for the positive graph occurrence to be valid.

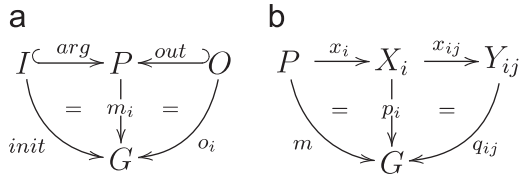

Fig. 4. Formalisation of pattern instantiation: (a) simple pattern instantiation and (b) application conditions.

Formally, a pattern is defined as $p = (P, I \overset{arg}{\hookrightarrow} P, O \overset{out}{\hookrightarrow} P, \bigwedge_{i\in I}(x_i \Rightarrow \bigvee_{j\in J_i} x_{ij}))$, where P is the positive graph, I is the subgraph containing the input arguments, O is the subgraph containing the output, and $x_i : P \to X_i$ and $x_{i,j} : X_i \to Y_{i,j}$ are injective morphisms (X_i is the *premise* and $Y_{i,j}$ are the *consequences* in the meta-model).

Fig. 4 shows two diagrams explaining the pattern instantiation process. Fig. 4(a) depicts a simple pattern P with input I and output O. Both I and O are subgraphs of P. A match m_i: $P \to G$ is valid if it commutes with the initial match *init*: $I \to G$. Moreover, the result of the pattern instantiation can be given as the matches o_i: $O \to G$ such that the right triangle commutes with the corresponding m_i. Fig. 4(b) shows the instantiation of a pattern with application conditions (output and arguments are omitted). Given a match m for P, if a morphism $p_i : X_i \to G$ is found, then some morphism

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

$q_{i,j} : Y_{i,j} \rightarrow G$ must also be found, such that both triangles in the figure commute. Technically, morphisms m, p_i and $q_{i,j}$ are clan-morphisms [9], as instances of abstract classes may appear in P, X_i and $Y_{i,j}$ that can be mapped to instances of some class in their inheritance clan (i.e. the same class or any subclass). We also require the typing of $Y_{i,j}$ be more concrete (or equal) than the type of X_i, and this one more concrete (or equal) than the type of P. Except for the notion of match initialisation and outputs, our patterns are similar to the notion of *graph constraint* found in the graph transformation literature [9].

3.2. Actions and redesigns

Next, we present the part of the SLAMMER meta-model for the specification of actions, which is shown in Fig. 5. In SLAMMER, *Actions* can be redesigns to be executed on the models, as well as any specific task such as generating a report or printing a model. They are composed of an ordered sequence of *Tasks*, which in their turn can be described by means of procedural code (class *TaskText*), using a graph transformation system [9,13] (class *TaskGG*), or by customising one of the generic model manipulation templates proposed by SLAMMER (concrete classes that inherit from class *TaskTemplate*). The same task can belong to different actions, thus favouring reusability and maintainability. Actions can be defined to be applied either when measurements reach a certain threshold value (relation *fires*) or independently from measurement values. In the first case, the action is executed for each value in the domain for which the measurement makes the threshold condition true. However, as problems derived from automatic redesigns are well-known [19], we provide actions with attribute *execution* that selects whether this is automatically executed or it needs human supervision (i.e. a confirmation is required before executing the action for each domain value that make the threshold condition true).

Graph grammar tasks allow specifying model redesigns by means of graph transformation, which is a visual, declarative means to specify model manipulations [9,13]. A graph grammar is made of rules with a left- and a right-hand side (LHS and RHS) that contain graphs. In order to apply a rule to a graph, an occurrence (a matching morphism) must be found between the LHS and the graph. Then, the identified occurrence is substituted by the RHS. Rules may have application conditions that are similar to the concept of graph constraint explained in previous subsection. A graph grammar is made of a set of rules and an initial graph. Applying the grammar means executing its rules until none of them is applicable. Examples of graph transformation for the definition of redesigns will be shown in Section 4.2.3.

SLAMMER provides four generic model manipulation tasks. Note that making actions general enough for being applicable to any kind of visual language is an intricate problem, as usually actions are close to the domain of application. However, when working with visual languages, many actions can be decomposed in some of the following basic tasks: merging two elements in a single one, breaking an element in two different ones, or moving relations from one element to another, where in addition the elements can be connected. These basic tasks are illustrated in Fig. 6, which correspond to the predefined SLAMMER tasks *Merge*, *Split*, *Move* and *Pull*. These four classes inherit from abstract class *TaskTemplate*, which defines the type to which the task applies, a boolean attribute *subtypeMatching* to allow applying the task to the subtypes of the type, and finally the action, which in fact is specified in each concrete subclass. These basic model manipulations can be customised for a DSVL by providing the domain specific information and extra task execution parameters depending on the task.

Task *Merge* merges two entities (of the type specified in the inherited attribute *type*) in a single one that brings together all the relationships of the original entities (see Fig. 6). If originally both entities defined the same relationship, the merged entity contains it twice. Attribute *rel_duplication* allows selecting whether this is allowed or if duplicated relationships are deleted after the merging. Attribute *att_merging* specifies the attribute merging mechanism, either as the concatenation of the original values or just by taking one of them. We are aware of the fact that with the current definition it is not possible to specify different merging policies depending on the specific entity, relation type or attribute. This could be done by specifying attributes *rel_duplication* and *att_merging* for each relation type and attribute of the entity, respectively. However, this would be quite tedious for the designer. As we want to make the customisation process easy, we adopt the solution of specifying just a general merging policy. Note that an action is

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

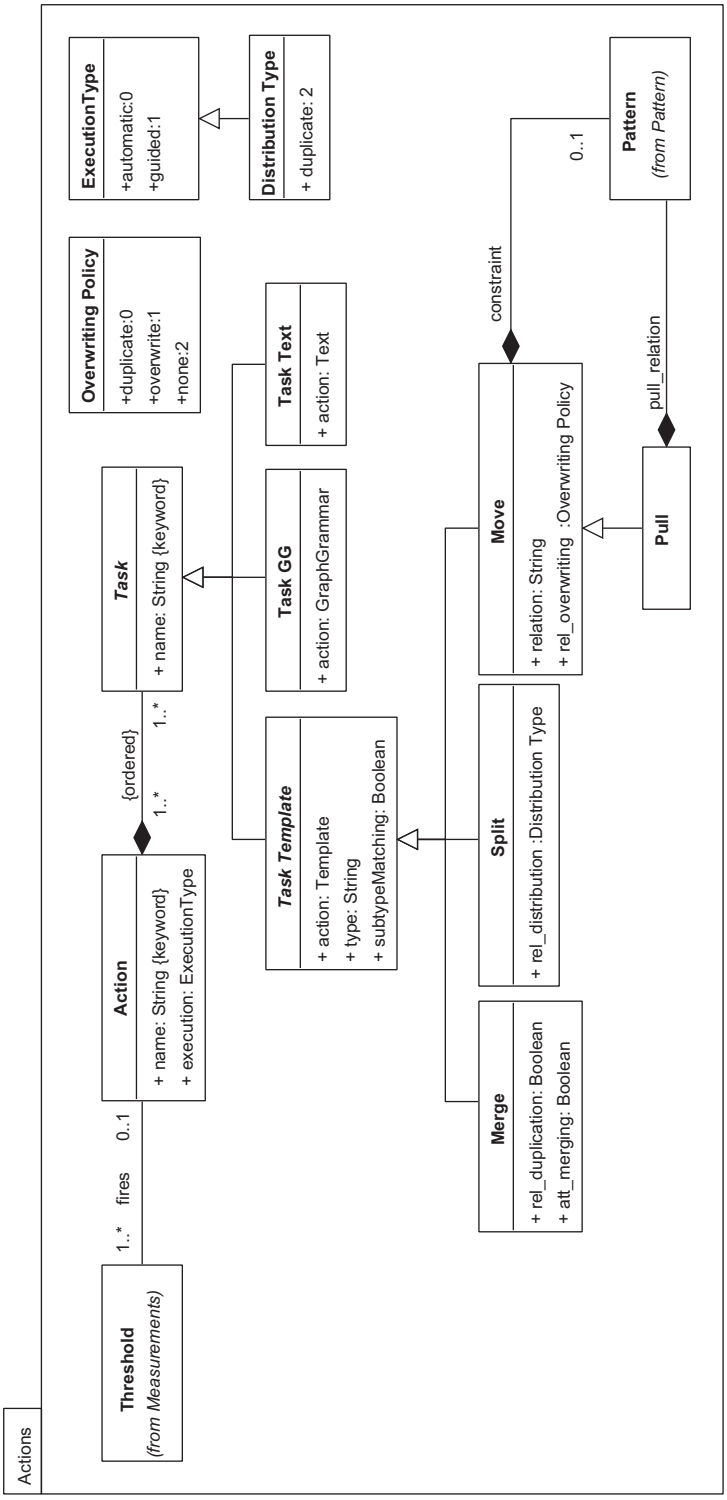

Fig. 5. The SLAMMER meta-model (actions package).

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

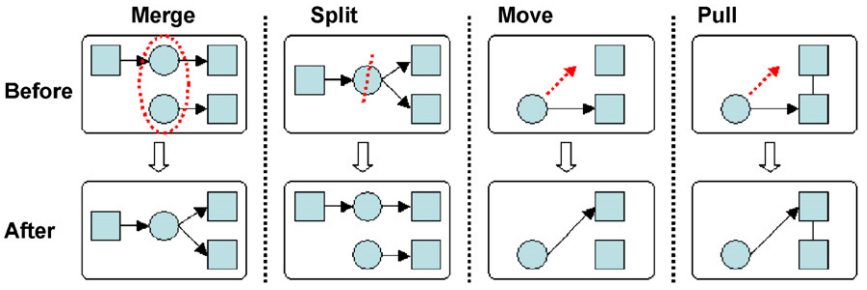

Fig. 6. Basic model manipulations for visual languages.

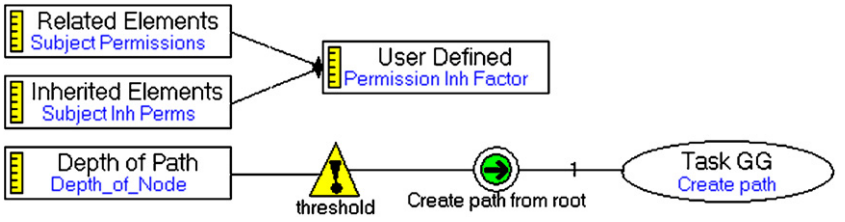

Fig. 7. SLAMMER model example.

made of a sequence of tasks, and thus, it is possible to define exceptions to the policy by means of additional tasks to be executed before or after the merging.

Task *Split* divides in two an entity of the specified type. The relations of the original entity are duplicated, or distributed between the new ones either randomly in equal parts or guided by the user selection. This option is chosen assigning values *duplicate*, *automatic* or *guided* to attribute *rel_distribution*, respectively. Attribute values are copied to the new entities. In case of keywords, an incremental sequential number is concatenated to their value in order to avoid keyword duplication.

Task *Move* moves relationships between entities of the same type (see Fig. 6). In addition to the entity type, it is necessary to specify the relation type to be moved (attribute *relation*), and the overwriting policy in case the relation already exists in the target entity (attribute *rel_overwriting*). Possible values for the overwriting policy are *duplicate* if we want to move the relation maintaining the existing one in the target, so that the target entity finally has the relation twice; *overwrite* if the relationship is moved and overwrites the one in the target; and *none* if the relation is not moved. By default all relations of the specified type are moved. However, it is also possible to restrict the number of relations to move by means of a pattern that receives as arguments the elements that take part in the activity (i.e. the relation to move and the source and target elements). In this case the task is applied for those relations that satisfy the pattern.

Finally, *Pull* specialises task *Move* to those cases where the task is performed only if the involved entities are related. The relation is specified as a pattern with the entities as arguments and no output.

3.3. Concrete syntax for SLAMMER

SLAMMER is a DSVL. Therefore, we provide a visual representation to its meta-model elements. Measurements are visualised as white rectangles with the metric type and name inside. Dependencies between measurements are represented as arrows, where the arrowhead indicates the data flow direction. Thresholds are drawn as a yellow triangle with an exclamation mark inside and the threshold name below. Actions are represented as green circles with a black arrow inside and the action name below. If its execution mode is automatic, a second circle surrounds the figure. Tasks are ellipses with the task type and name inside. Finally, relationships between entities are represented with lines.

As an example, Fig. 7 shows a SLAMMER model using its concrete syntax representation. The model contains the definition of four measurements. One of them (`Permission Inh Factor`) uses the results of other two measurements. Metric

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

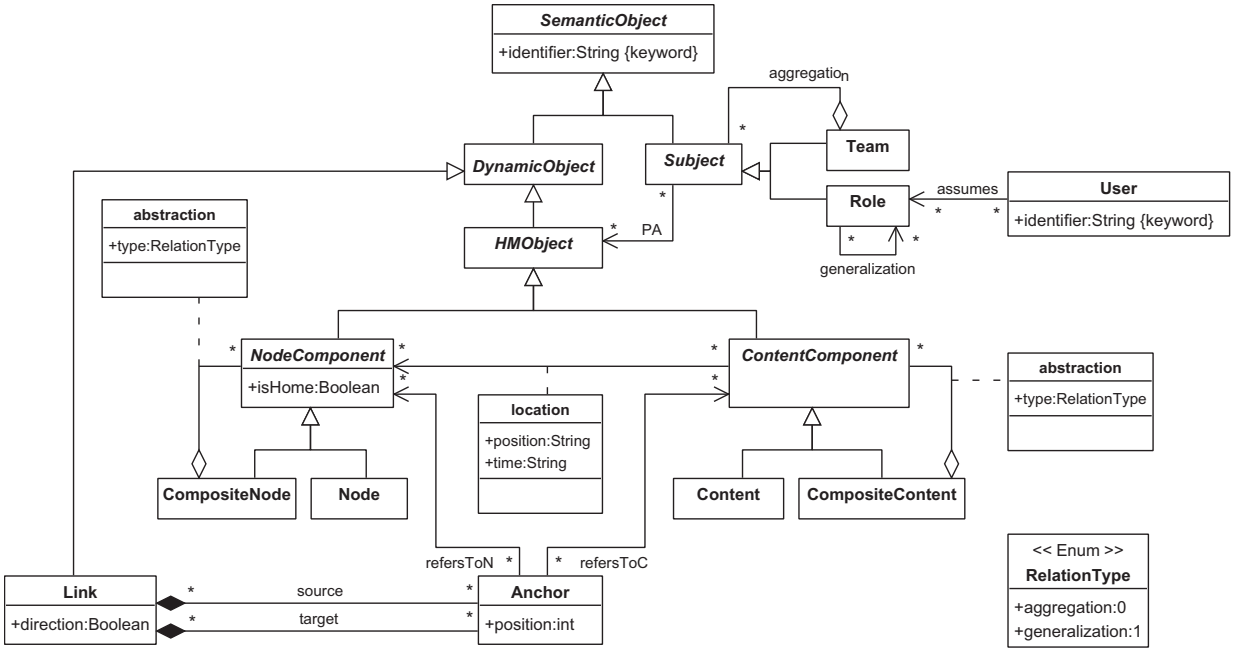

Fig. 8. Labyrinth meta-model.

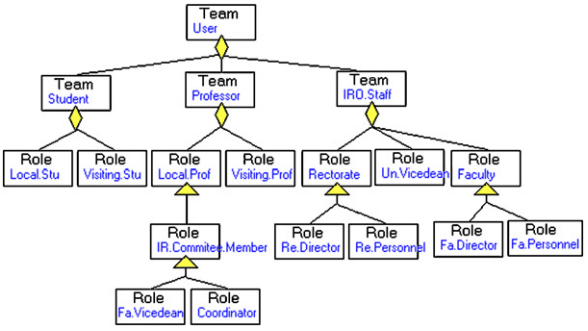

Fig. 9. IRO users diagram.

`Depth_of_Node` has an associated threshold that automatically triggers a task. The latter is called `Create Path` and is specified through a graph grammar.

4. Example

This section illustrates the use of SLAMMER by defining a set of measurements and actions for Labyrinth [10], a DSVL used to design web applications.

4.1. Labyrinth, a DSVL for web design

Fig. 8 shows an excerpt of the Labyrinth meta-model. Roughly, in Labyrinth web applications are modelled as a set of nodes where contents are located. Navigation features are expressed through anchors and links. Besides, users can assume roles and belong to different teams from which they receive a set of permissions concerning the nodes and contents they are allowed to visit. In addition, roles and teams can be nested in hierarchical structures where permissions assigned to more general roles are inherited by more specific roles, and permissions assigned to teams are propagated to their components.

The Ariadne Development Method (ADM) [1] is based on Labyrinth to define a set of diagram types that capture the different aspects of web applications. Next we show some ADM example diagrams that belong to the design of a system for the

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

management of the International Relations Office (IRO) of a university. Fig. 9 shows the *Users diagram*, which contains the hierarchy of system roles and teams. The defined roles can belong to one of the three following teams: `Student`, `Professor` or `IRO.Staff`. Students and professors can be local or visiting. In addition, local professors can belong to the international relations committee as coordinators. One of the local professors is the department vice-dean for international relations. With respect to the IRO staff, there are three roles defined: `Rectorate`, `Faculty` and `Un. Vice-dean`. The first two are specialised in order to differentiate between the director and the regular personnel.

Fig. 10 shows an excerpt of the *Navigational diagram* that includes the navigational map for role `Local.Prof`. Each navigation step between two nodes is made of two objects of class *Anchor* (drawn as a small anchor), one object of class *Link* (a black square), and the appropriate associations between them. In the diagram, the arrowheads indicate the direction of the navigation.

The *Access table* in Fig. 11 shows (part of) the permission assignment to IRO roles and teams, which are allowed to access only those nodes and contents to which they are related. For example, the IRO staff can access nodes `Information`, `Travel Fundings` and `Forms` but not to nodes `Assign Erasmus`, `Modify Erasmus` and `List sch. holders`. On the contrary, role `Coordinator` can access to all of them. Recall that permissions are inherited through the role and team hierarchy, and thus descendents of the subjects in the access table inherit the permissions defined by their ancestors.

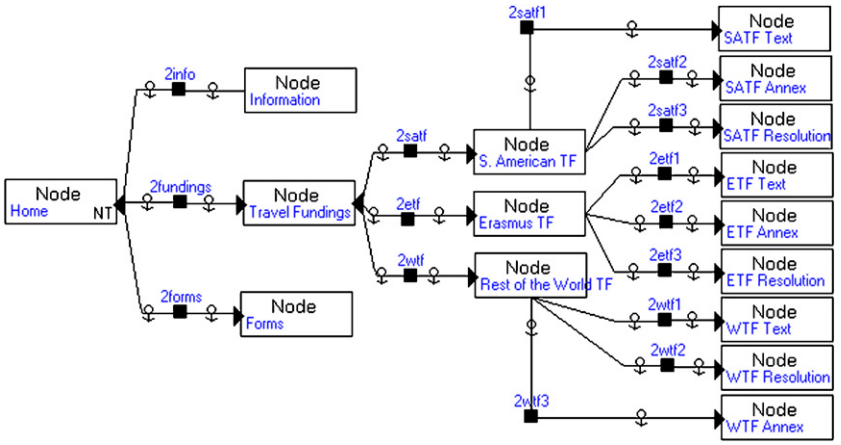

Fig. 10. IRO navigational diagram (partially shown).

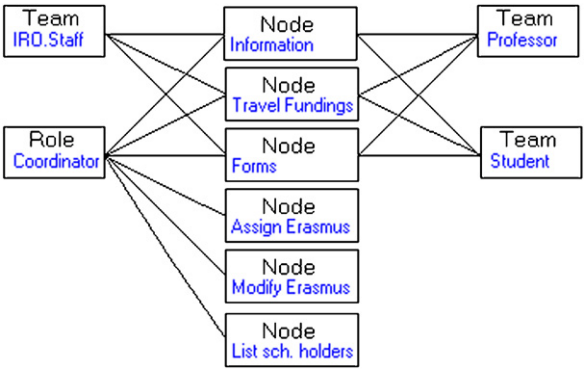

Fig. 11. IRO access table (partially shown).

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

Table 1
Specification of navigational measurements for Labyrinth, by using SLAMMER

Name	Goal	SLAMMER type	Pattern	Dependencies
NNC	Size	NumberOfElements	Fig. 12(a)	–
NNL	Size	NumberOfElements	Fig. 12(b): args = [ ], out = [2]	–
DeNM	Size	UserDefined	–	NNC, NNL
BNM	Usability	NumberOfElements	Fig. 12(b): args = [ ], out = [3]	–
MPBNC	Usability	DistanceMatrix	Fig. 12(b): args = [1], out = [3]	–
ROC	Usability	UserDefined	–	MPBNC
RIC	Usability	UserDefined	–	MPBNC
D	Usability	DepthOfPath	Fig. 12(b): args = [1], out = [3]	–
FONC	Maintainability	DirectConnections	Fig. 12(b): args = [1], out = [3]	–
FINC	Maintainability	DirectConnections	Fig. 12(b): args = [3], out = [1]	–
C_p	Connectivity	UserDefined	–	NNC, MPBNC
S	Linearity	UserDefined	–	NNC, MPBNC

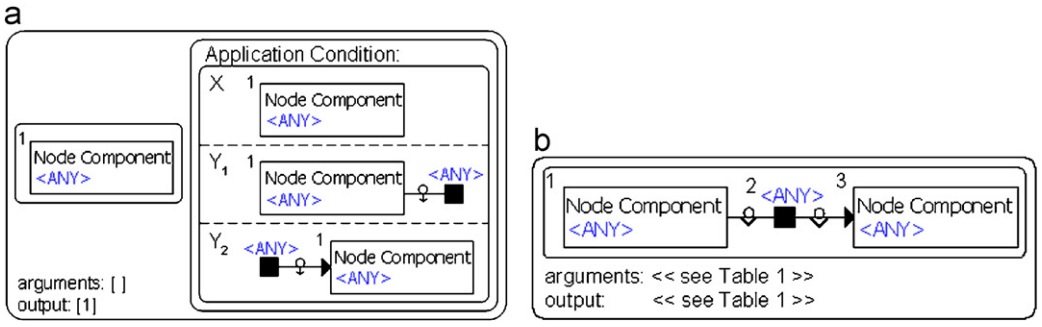

Fig. 12. Graphical patterns used by Labyrinth's navigational measurements.

4.2. Using SLAMMER to define measurements and actions for Labyrinth

4.2.1. Navigational measurements

In this subsection we use SLAMMER in order to customise for Labyrinth a well-known set of navigational measurements. The considered measurements are summarised in Table 1 and explained next. For each one of them, the table shows its name (the abbreviation), its goal, the SLAMMER measurement used to define it, the corresponding customisation graphical pattern, and its dependencies with other measurements. In the table, element-oriented measurements have as domain the type "NodeComponent", while group-oriented ones have as domain the tuple ["NodeComponent", "NodeComponent"]. In all cases attribute *subtypeMatching* is set to true as we want to take measures from both composite and single nodes, which are subclasses of class *NodeComponent* (see Labyrinth meta-model in Fig. 8).

The *number of navigational contexts* (*NNCs*) [20] is an indicator of the navigational model size. It can be used to detect navigational trees poorly structured likely due to the identification of user requirements with navigational targets. In Labyrinth, a navigational context is a node component that participates in a navigational link. Thus, we define NNC as a measurement of type *NumberOfElements* where the element to count is specified with the pattern shown in Fig. 12(a). In this way, the measurement counts node components (simple and composite) that are source (consequence graph Y_1) or target (consequence graph Y_2) of a navigational link. The output of the pattern is the element to be counted, that is, the node component.

The *number of navigational links* (*NNLs*) [20] is another indicator of the navigational model size that counts the NNLs, as in certain web notations not all links have a navigational purpose. We define it for Labyrinth as a measurement of type *NumberOfElements*, where the elements to count (i.e. links

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

between two node components) are specified by using the pattern shown in Fig. 12(b) with element labelled "2" as output.

A third indicator of the navigational model size is the *density of the navigational map* (*DeNM*) [20], with formula *NNC*/*NNL*. Thus, we define a *UserDefined* measurement with incoming dependencies from the previously defined NNC and NNL. Its calculation method performs the quotient.

The *breadth of the navigational map* (*BNM*) [20] is a measure of the navigational model usability at the first level. It counts the NNCs that are directly reachable from the initial context. The larger the number, the harder to use the system, since many navigational possibilities are presented at once. We can define this measurement as a customisation of *NumberOfElements*, where the pattern specifying the element is shown in Fig. 12(b). In this case the attribute *isHome* of the node component labelled "1" must be true as the measurement takes into account just the initial context, and the output is element labelled "3" as we want to obtain the node components that are reachable from the initial one.

The *minimum path between navigational contexts* (*MPBNCs*) [20] is also an indicator of the navigational model usability, which gives the number of links that have to be navigated from each navigational context to the rest. We define it as a *DistanceMatrix* measurement where the navigational step for Labyrinth is customised by pattern in Fig. 12(b) with argument "1" and output "3". That is, the target node component of a navigation step (output) will be the source of the following step (argument).

The *relative out* and *in centrality* (*ROC*/*RIC*) [21] measure how easily a context can access or be accessed from other contexts, respectively, and can be used to identify root nodes in tree structured web systems. They are calculated as the normalised sums of the distances to/from any other context. Thus, we define them as *UserDefined* measurements that depend on measurement MPBNC, as this latter produces the distance matrix necessary for the calculation.

The *depth of a node* (*D*) [21] is the distance from the initial navigational context to the given node. It indicates the ease with which a node is reached and, in this way, the importance for the user. The bigger the distance, the harder becomes reaching it. This can be intentional when nodes contain low relevance information, but sometimes it can identify missing links or nodes with difficult access. This measurement is customised from *DepthOfPath*, where the navigation step is given by the pattern in Fig. 12(b) with element "1" as argument and element "3" as output.

The same pattern is used to customise the *fan-in* and *fan-out of a navigational context* (*FINC*/*FONC*) [20], but as measurements of type *DirectConnections*. They count the NNL that call a navigational context or that are called by it, respectively. Thus, given a node component, we are interested in knowing how many others are directly connected through a navigational step to it. Note that FINC has interchanged the elements that belong to the input and to the output of the pattern. These measurements are indicators of the navigational map maintainability: the larger they are, the greater the interdependency and the lower the reusability.

Compactness (C_p) [21] measures the degree of connectivity of a navigational model. It takes values from 0 to 1. The higher the value, the easier is to reach any node in the map, perhaps leading to disorientation. A low value can indicate an insufficient number of links. Compactness is calculated by the formula $C_p = (\mathrm{Max} - \sum_i \sum_j D_{i,j})/\mathrm{Max} - \mathrm{Min}$, with $\mathrm{Max} = (n^2 - n) * k$; $\mathrm{Min} = (n^2 - n)$; $n =$ NNCs; $D_{i,j}$ = distance between contexts i and j. Note that n is equal to the NNC, and that the distances in the formula are the ones calculated by the MPBNC. Thus, we define compactness as a *UserDefined* measurement with dependencies NNC and MPBNC.

The *Stratum* (*S*) [21] measures the degree of linearity of a navigational map, and takes values between 0 and 1. High values indicate linear web sites that, although easily navigable, are often tedious to browse. As before, the stratum formula can be calculated by using the NNC and MPBNC.

The set of presented navigational measurements have been defined in a generic way irrespective of the kind of user, but in some cases the availability of a link (and thus the availability of the target node) depends on the roles of the user navigating the system. For the sake of simplicity we have not shown this possibility, although it could be done by first obtaining a derived view [22] containing just the navigational diagram for the particular user, and then measuring this diagram.

In any case, note how the patterns used to customise the majority of the navigational measurements are the same, just changing the elements that belong to the arguments or to the output. Indeed,

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

Table 2
Specification of security measurements for Labyrinth, by using SLAMMER

Name	Goal	SLAMMER type	Pattern	Depen.
Subject permissions (SP)	Auxiliary	RelatedElements	Fig. 13(a)	–
Subject inherited permissions (SIP)	Reusability	InheritedElements	Fig. 13(a,b)	–
Permission inheritance factor (PIF)	Reusability	UserDefined	–	SP, SIP
Subject similarity (SS)	Cohesion	DistanceBasedSimilarityMatrix	Fig. 13(a)	–

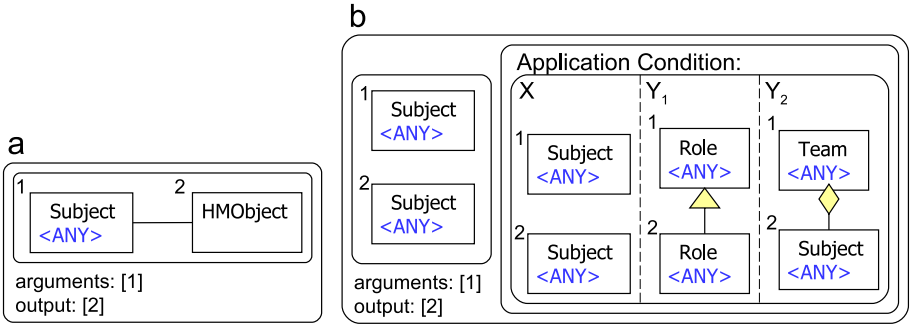

Fig. 13. Graphical patterns used by Labyrinth's security measurements.

some measurement definitions (such as MPBNC, FONC and D) use the same customisation patterns, although their goal and internal method calculation greatly differ. This is because in all of them it must be specified what a navigation step is. Thus, the definition of new navigational measurements for Labyrinth by using SLAMMER turns out to be an easy task, as we can reuse the pattern.

4.2.2. Security policy measurements

Table 2 summarises a second set of SLAMMER-based measurements, this time oriented towards measuring properties of the role-based access control used by Labyrinth. As Labyrinth supports the concept of inheritance (of permissions) in its role hierarchy, we can take advantage of some known measurements in the object-oriented domain and adapt them to the Labyrinth concepts. The domain of the element-oriented measurements in the table is made of all objects of class *Subject*, and the one for group-oriented ones is a tuple made of two *Subjects*. Subtype matching is selected in order to consider objects of both classes *Team* and *Role*.

The auxiliary measurement *subject permissions* (*SPs*) counts the number of hypermedia objects (i.e. nodes and contents) to which a subject has direct permission to access (i.e. a relation *PA* exists between both). We do not use this measurement as an indicator of any system property, but it will be used by a subsequent measurement. We have defined it as a *RelatedElements* measurement customised with the pattern shown in Fig. 13(a), where the argument is the subject for which the measure is taken, and the output is the hypermedia object to count.

The *subject inherited permissions* (*SIPs*) is a customisation of the path-oriented measure *InheritedElements*, and calculates the number of permissions a subject inherits. It must be customised by a pattern with the structure of a step in the hierarchy (see Fig. 13(b)) and another one specifying the inherited element (see Fig. 13(a)). Note that a single pattern can express several valid steps in the subject's hierarchy, which are the generalisation in the case of roles (Y_1) and the aggregation in the case of teams (Y_2).

Measurement *permission inheritance factor* (*PIF*) calculates the inherited permission ratio, and it is an indicator of the reuse. It is a particularisation of measurements method and attribute inheritance factor (MIF and AIF, respectively) in the object-oriented domain [6]. PIF is the sum of all permissions inherited by some subject in the system divided by the sum of all permissions defined (local and inherited) by subjects. Thus, we define a *UserDefined* measurement with dependencies from the two previous measurements.

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

Finally, the *subject similarity* (*SS*) [17] measurement is an indicator of the cohesion and can help detecting redundancies in the security policy. It calculates how similar two subjects are by studying their assigned permissions. The measurement takes values in the interval [0, 1]. The lower the value, the more similar the subjects are. We define this measurement as a *DistanceBasedSimilarityMatrix* customised with two properties, each one of them specifying how the permission assignment is expressed in Labyrinth, and thus, using the graphical pattern shown in Fig. 13(a). Each property is associated to one of the subjects to compare (i.e. attribute *order_type* for the first property is "1" as it refers to the first subject, and it takes value "2" for the second property).

4.2.3. Actions

Next, we exemplify the definition of actions in SLAMMER by defining a small set of illustrative redesigns for Labyrinth, which are summarised in Table 3. We also discuss how some of these actions could be guided by threshold values of some of the previous measurements.

Action *Collapse subjects* merges two roles or teams into a single one, which brings together all permissions defined by the formers. The action can be used in order to compact two subjects that are quite similar because they define the same permissions, or because although defining different permissions, they are assigned to the same users. Sometimes this is due to a bad security policy design. In the first case the action could be guided by a combination of threshold values obtained from measurements SS and PIF: the first one detects when two subjects are similar but it does not take into account inheritance information, and the second one calculates the inherited permission ratio. If both values are close to 0, the subjects are good candidates for the action. This action is made of a single task of type *Merge* that applies to roles and teams, therefore attribute *type* contains "Subject" and *subtypeMatching* is true. In addition, attribute *rel_duplication* is false in order to avoid redundant permissions to the same objects, while attribute *att_merging* is set to true.

Table 3
Specification of actions for Labyrinth, by using SLAMMER

Name	Made of tasks…	Guided by…
Collapse subjects	(1) Merge: "Merge subjects"	SS, PIF
Collapse nodes	(1) Merge: "Merge nodes"	–
Create path from root	(1) TaskGG: "Create path"	D
Eliminate redundant permissions	(1) Pull up: "Pull up permissions" (2) TaskGG: "Flatenning" (3) TaskGG: "Eliminate redundancy"	–

Similarly, action *Collapse nodes* merges two node components. It can be applied when the two nodes are similar, so as to make more maintainable the navigational model by minimising its size. Node couplings could be detected by customising a *DistanceBasedSimilarityMatrix* in order to compare pairs of nodes. The action could also be used to compact consecutive nodes with little information and make the navigation lighter for the user by minimising the number of navigational steps. In this case the user should provide the nodes to collapse. As before, the action is a customisation of task *Merge* with the same attribute values but to be applied on type *NodeComponent*.

The remaining actions show examples of the use of graph transformation rules to express redesigns [13]. For example, action *Create path from root* is made of a *TaskGG* called "Create path" that creates a navigational path from the root node to a given node. The graph grammar is made of just a rule which is shown in Fig. 14. The elements to be added by the rule application are shown in a coloured

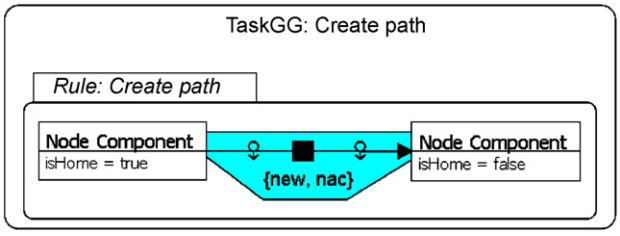

Fig. 14. Graph grammar task for action "create path from root".

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

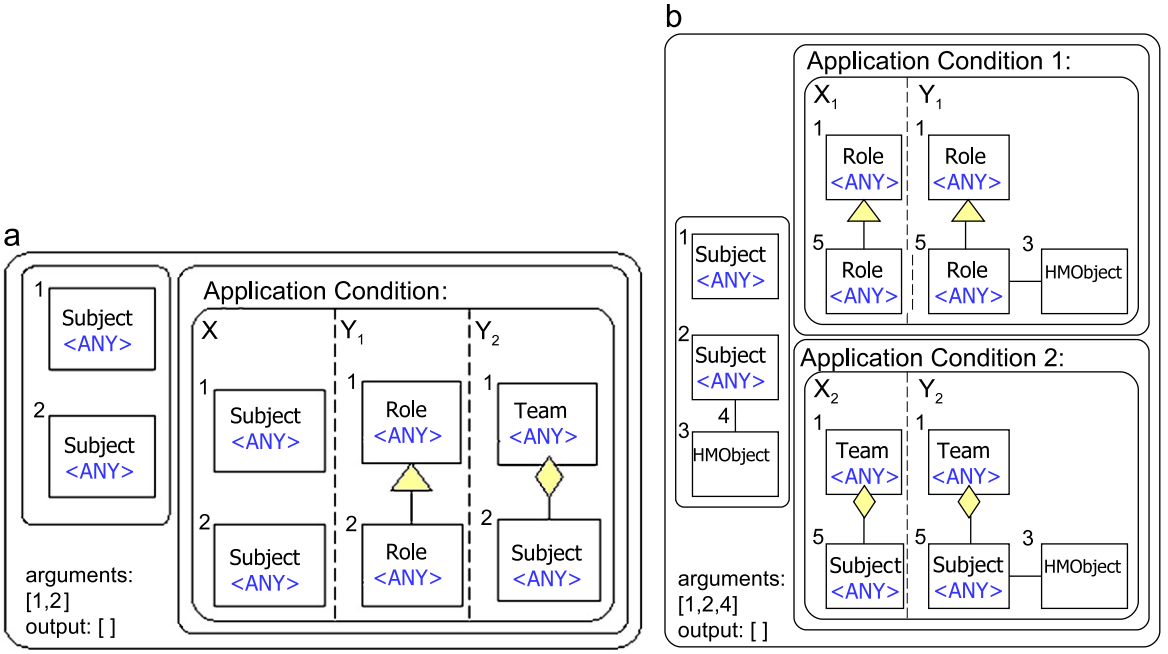

Fig. 15. Graphical patterns for the customisation of task "pull up permissions".

polygon and labelled as "new". The rule is not applied if such path already exists, which is expressed by the NAC. The action is useful if we want to create a direct link from the application home page to some navigational node, perhaps because the node is not reachable or because a high number of navigational steps are required in order to access it. Measurement *Depth of a Node* can be used to detect nodes with difficult access. If we associate an appropriate threshold value to the measurement (e.g. 0, which means that it is not possible to reach the node), we can use it to detect the candidate nodes, and thus automatically fire the action on them. In this way the newly created path solves the problem and facilitates the navigation.

The last action example (*Eliminate redundant permissions*) illustrates the composition of tasks to perform a single action. The purpose of this action is to eliminate redundancies from the subjects' hierarchy by taking advantage of the inheritance of permissions that Labyrinth defines. With this purpose, the action first pulls up permissions to a parent subject if all its children define the permission, and then removes permissions assigned to a subject if some subject's ancestor already defines them. The action is made of the three tasks that are shown in Figs. 15 and 16. The first task pulls up the permissions by means of a customisation of the *Pull* template. This is defined for type *Subject* (with subtype matching) and relation *PA* (the one used for permission assignment, see Labyrinth metamodel in Fig. 8). In order to pull up a permission, an inheritance relation of generalisation or aggregation must exist between the source and target subjects. This is specified by pattern in Fig. 15(a). In addition, as we only want to pull up those permissions defined by all children of a subject, we constrain its applicability by means of the pattern shown in Fig. 15(b). The pattern receives as input the relation to move and the source and target subjects. The application conditions check the existence of the permission to be moved in every target subject's child, being the parent either a role (application condition $X1 - Y1$) or a team (application condition $X2 - Y2$). Note the expressive power of patterns in order to specify complex structures and conditions.

In the same action, the elimination of redundant permissions is performed by the two graph grammar tasks shown in Fig. 16, which are executed afterwards. Recall that executing a grammar means (non-deterministically) applying its rules as long as possible. Task "Flattening" performs the flattening of the hierarchy by copying permissions from each parent to its descendants. Two different rules consider the flattening of permissions assigned to

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

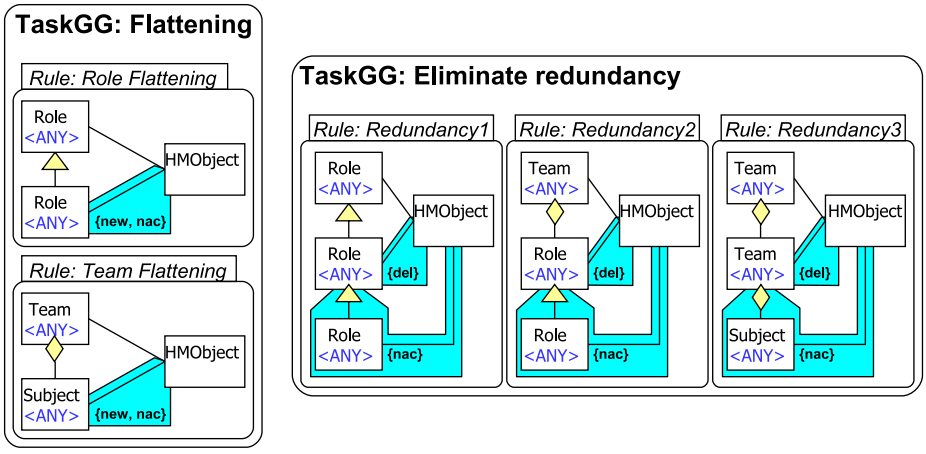

Fig. 16. Graph grammar tasks for action "eliminate redundant permissions".

teams and to roles, respectively. Task "Eliminate redundancy" goes up the hierarchy removing redundant permissions. It is made of three rules that consider the three possible hierarchical combinations: a role with a child role, a team with a child role, and a team with a child team.[2] The NACs in the rules forbid removing a permission if some direct children defines it. In this way the deletion starts from the deeper roles in the hierarchy to the higher ones. By performing an initial task that flattens the hierarchy we are able to eliminate redundancies produced not only when a children and its parent define the same permission, but also when an arbitrary number of ancestors exist in between. This action cannot be guided by measurement values, but it is performed in the whole subjects hierarchy. However, the modelling of roles and teams usually does not involve a big set of entities, and therefore this is not a real problem.

5. Tool support

One of the goals of the present work is to provide visual editor developers with a tool that facilitates the task of integrating mechanisms to measure and improve the quality of the models in the developed editors. In previous sections we showed how the use of SLAMMER could facilitate the specification of measurements and actions for DSVLs. In this section we show the tool that we have created to support SLAMMER and how it is used.

[2]A team cannot generalise a role.

5.1. Framework implementation

The new tool has been built by using AToM^3 [23], an environment that allows the description of DSVLs by means of meta-modelling and its manipulation by means of graph transformation. In AToM^3, the abstract syntax of a DSVL is specified with a meta-model (either a class or an entity relationship diagram), while its concrete syntax is defined by simply providing icons for each class and arrows for each association of the meta-model. Starting from this definition, a visual editor is generated that allows the creation of models conforming to the given meta-model.

In the case of SLAMMER, its abstract syntax meta-model is made of the sum of the meta-models shown in Figs. 1, 3 and 5. As we wanted to automatically generate measurement and redesign tools from SLAMMER models, we enriched its meta-model with attributes oriented to customise the environment to be generated. In particular, we added an abstract class *UIButton* as the parent of classes *Measurement*, *Action* and *Task*. This class has a single boolean attribute *button* that, if set to true, generates a button to execute the measurement, action or task. This allows, for example, not to generate buttons for auxiliary measurements. In addition, class *Measurement* was provided with two additional attributes. The first one (*genReport*) has boolean type and is selected in order to obtain a report in PDF format with the measurement result. The second one (*report*) is an enumerate type to select whether the report should show all the obtained values, or only the ones making some

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

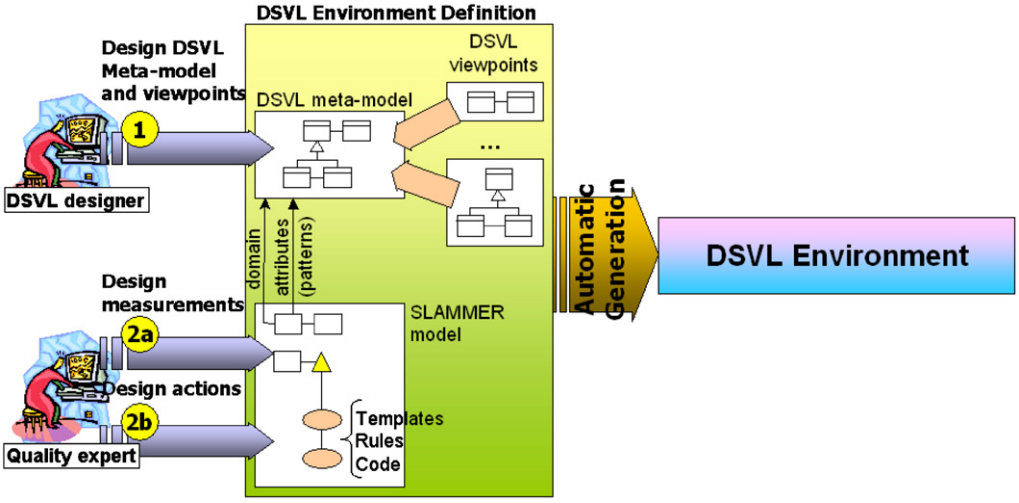

Fig. 17. Definition of DSVL environment.

threshold condition true. Starting from the enriched SLAMMER meta-model, we used the code generation capabilities provided by AToM^3 in order to generate a visual editor for it. The resulting tool, which was integrated in AToM^3, allows building SLAMMER models and this way customising measurements and actions for a given DSVL. In addition, we had to manually add a code generator to the SLAMMER tool, able to synthesise a measurements and redesigns tool from the SLAMMER models.

The process of defining a DSVL environment by using AToM^3 is shown in Fig. 17. First, the meta-model of the language is specified. In the case of multi-view DSVLs (i.e. families of DSVLs or diagram types where each notation captures a perspective of the system, e.g. Labyrinth), first the whole language meta-model is specified, and then the different diagram types are given as subsets or projections of it (see [24] for a detailed explanation). Note that overlapping is allowed thereby inter-dependencies can arise. Then, the new SLAMMER tool is used in order to define measurements, thresholds and actions for the particular DSVL. The measurements can be defined as customisations of the suite offered by SLAMMER, therefore only the domain (elements of the DSVL) and the specific attributes to measure (specified as patterns) have to be given. Actions are made of tasks that can be specified either procedurally (by using Python), by means of graph grammars, or by customising task templates by means of patterns. The definition of the DSVL and of the measurements and actions is performed by experts in the respective areas (probably two different people). By following a model-driven approach, a visual editor for the DSVL is automatically generated that allows specifying models conforming to the DSVL meta-model(s), measuring the models, and performing the predefined redesigns (likely guided by threshold values of measurements) in order to improve the design quality.

The working scheme of the generated environment is summarised in Fig. 18. The end-user who uses the language can specify diagrams or system views conforming to the DSVL meta-model, or conforming to some diagram type meta-model in the case of multi-view DSVLs. Syntactic and static semantics consistency between diagrams is achieved by means of triple graph transformation systems [22] that internally build a *repository* by merging the different system diagrams through their common elements. The transformation systems create mappings between the repository elements and the corresponding elements in the diagrams so that changes in one system diagram or in the repository are propagated to the other diagrams if necessary. These consistency mechanisms are provided by the environment and hidden to the user. In addition, the user can take measures in the repository model by using the user interface that is automatically generated from the SLAMMER model that was specified during the DSVL definition. Note that the SLAMMER measurements and actions are defined on the language generated by the whole DSVL meta-model, instead of associating them to specific diagram types. The reason is that measurements and

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

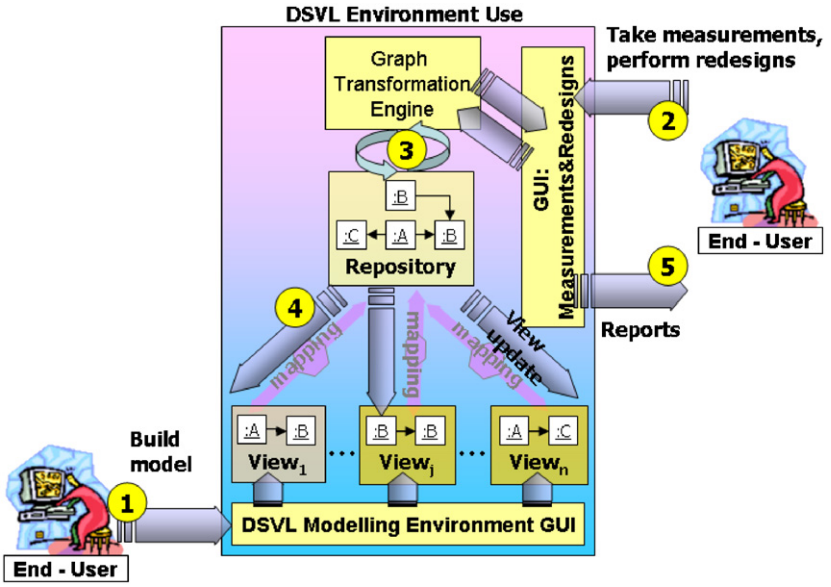

Fig. 18. Using the generated DSVL environment.

redesigns are usually performed on the whole system, and not in isolated views, as they may need information from different diagrams. Measurement results are shown to the user in the form of reports. The user interface also allows performing the actions that were specified in the same SLAMMER model. If the execution of an action produces a modification of the repository, then the changes are propagated to the system views by using the same mechanisms that provide system consistency after a diagram change. Note that model consistency is a crucial aspect of model redesign and that, if a system diagram changes, the changes should be propagated to the other ones in order to maintain the system consistent.

Next section illustrates the previous concepts by showing the visual editor we have built for Labyrinth by using the support tool.

5.2. Defining an environment for Labyrinth

ADM [1] is based on the Labyrinth meta-model in order to define a family of diagram types where each one of them captures a different aspect of a web application (e.g. navigation, security, etc.). In order to define a visual editor for it, we defined the Labyrinth meta-model and six of its diagram types in AToM^3. Then, we used the new tool generated for SLAMMER, which is integrated in AToM^3, in order to define the set of measurements, thresholds and actions presented in Section 4. Fig. 19 shows a screenshot in the measurement and redesign editing process for Labyrinth. Window 1 in the background is the new tool that we have implemented and contains the SLAMMER model specified for Labyrinth. In particular, the figure shows the editing of measurement `Depth_of_Node`, which is the upper one to the left in window 1. The measurement counts the number of necessary steps to reach a node starting from the root node. The editing of its attributes is shown in dialog box 2. By clicking on button "step" a new window is opened where the user customises the basic step for the measurement, which is given as a pattern. In the figure, window 3 contains the definition of the positive graph of such pattern, which in this case is made of two nodes joined by a link and two anchors. In the SLAMMER model, measurement `Depth_of_Node` defines a threshold value equal to 0 that automatically triggers action `Create path from root` for those nodes that have depth 0 and are not root. The action is made of a single task (the one that was shown in Fig. 14) that creates a direct link from the web root node to a given node. In this way, if some node is not reachable from the root (i.e. it has a depth equals to 0), a direct link is created from the root to the node.

The background window in Fig. 20 corresponds to the environment generated from the previous definition of Labyrinth. The six first buttons in the

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

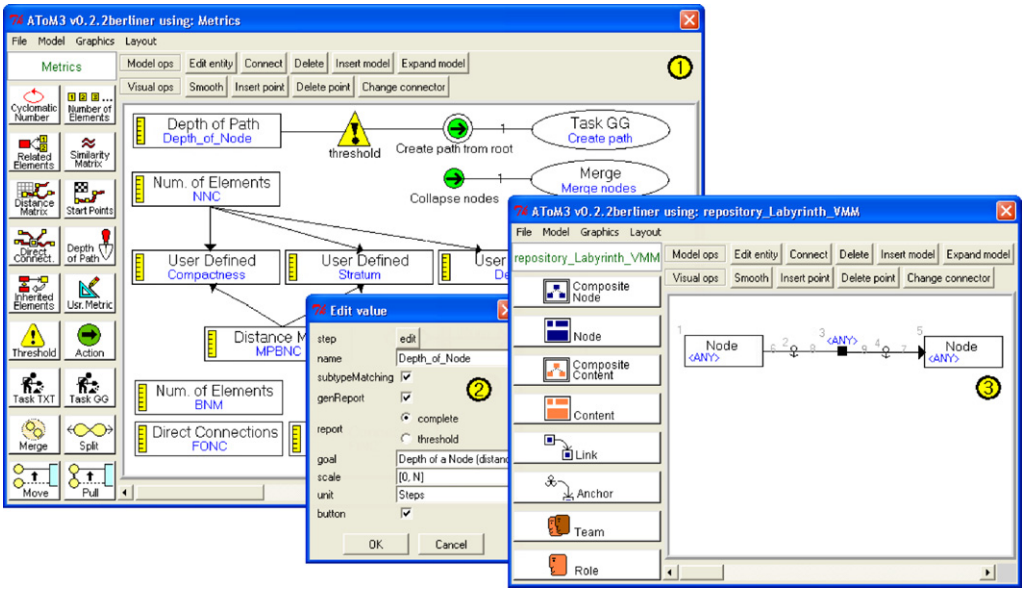

Fig. 19. SLAMMER model definition for Labyrinth.

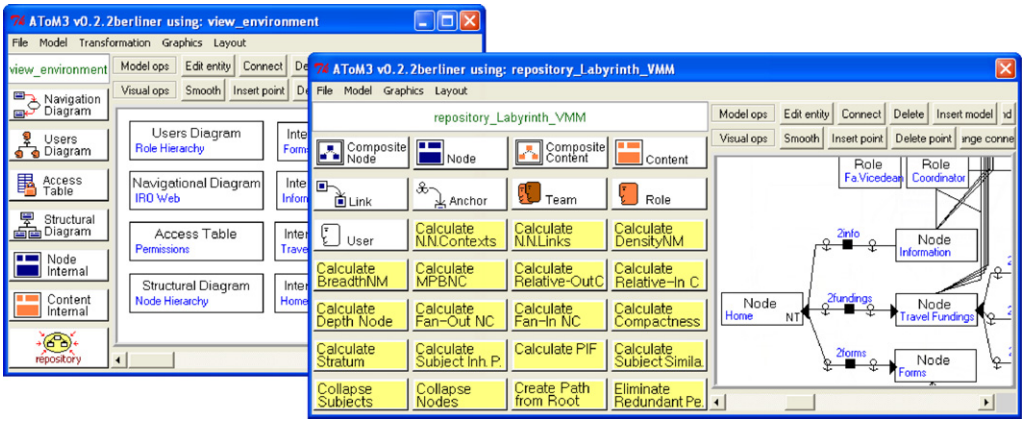

Fig. 20. Generated environment for Labyrinth.

user interface allow creating instances of the different diagram types. The canvas contains several instance diagrams that belong to the modelling of the IRO application introduced in Section 4. The last button in the user interface allows opening a window with the repository, which is partially shown to the right of the same figure. A set of buttons is present in the repository user interface that allows taking measurements and performing actions. These buttons were automatically generated from the information stored in the SLAMMER model provided during the definition of the environment. Calculating a measurement or performing an action just implies clicking on the corresponding button.

Fig. 21 shows some reports generated as PDF documents by the execution of the defined measurements on the example IRO models. The measurement results are shown as a table where each record corresponds to a value in the measurement domain. The first report corresponds to the execution of measurement NNC, which gives 16 as result (our system defines 16 nodes that take part in the navigational map). As this is a model-oriented metric, the obtained value gives a measure of the model and thus the resulting table contains just a

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

a

Metric: NNC
(complete report)

Nodes
16

b

Metric: MPBNC
(complete report)

lb_NodeComponent_repository_Labyrinth	lb_NodeComponent_repository_Labyrinth	Steps
Identifier : Home Type : lb_Node_repository_Labyrinth	Identifier : Erasmus TF Type : lb_Node_repository_Labyrinth	2
Identifier : Home Type : lb_Node_repository_Labyrinth	Identifier : Information Type : lb_Node_repository_Labyrinth	1
Identifier : Home Type : lb_Node_repository_Labyrinth	Identifier : List sch. holders Type : lb_Node_repository_Labyrinth	-1
Identifier : Home Type : lb_Node_repository_Labyrinth	Identifier : ETF Annex Type : lb_Node_repository_Labyrinth	3
Identifier : Home Type : lb_Node_repository_Labyrinth	Identifier : SATF Annex Type : lb_Node_repository_Labyrinth	3
Identifier : Home	Identifier : WTF Annex	

Fig. 21. Some reports generated by the execution of measurements: (a) NNC and (b) MPBNC.

a

Metric: D
(complete report)

lb_Node_repository_Labyrinth	Steps
Identifier : Home Type : lb_Node_repository_Labyrinth	0
Identifier : Information Type : lb_Node_repository_Labyrinth	0
Identifier : Travel Fundings Type : lb_Node_repository_Labyrinth	1
Identifier : Forms Type : lb_Node_repository_Labyrinth	1
Identifier : S. American TF Type : lb_Node_repository_Labyrinth	2
Identifier : Erasmus TF Type : lb_Node_repository_Labyrinth	2
Identifier : Rest of the World TF	

b

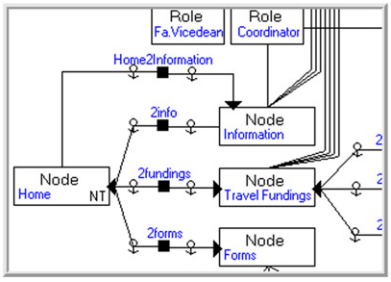

Fig. 22. Example of action execution guided by measurement threshold: (a) report and (b) repository.

record. The second report is partially shown and contains the group-oriented measurement MPBNC. Each record in the table contains the minimum path length to reach the node in the second column from the node in the first column. A value equals to −1 implies that the node in the second column is not reachable from the one in the first column.

Fig. 22(a) shows another report generated as result of the execution of measurement "depth of node". In the report we can see, for example, that nodes `Travel Fundings` and `Forms` have a depth equal to 1 because a link has to be traversed to reach them from the root node `Home`, and that node `Information` is not reachable from the root node because it has a depth 0. This latter case identifies a design error as all nodes should be reachable from the root. Thus, if we look at the repository in Fig. 20, we can see that a link exists from node `Information` to the root, but not the other way round (i.e. the link direction is wrong). As explained before, this measurement was defined with an associated action that is automatically triggered for those nodes for which the measurement is equal to the threshold value 0. The action creates a direct link from the root to the node that makes the threshold condition true, which in this case is node `Information`. An excerpt of the repository model after the action execution is shown in the Fig. 22(b), where a link has been created from the root node `Home` to node `Information`.

As example of action not guided by measurement values, Fig. 23 shows to the left the access table of our system example, and to the right the access table after applying action "Remove redundant permissions" by clicking on the last button generated on the repository user interface. The action first pulls up to parent subjects those permissions specified by

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

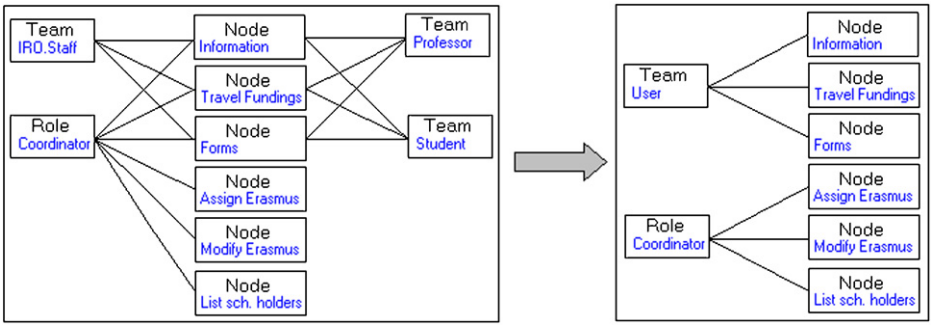

Fig. 23. Access table after applying action "remove redundant permissions".

all their children. For this reason permissions to nodes `Information`, `Travel Fundings` and `Forms` are moved from `Student`, `Professor` and `IRO.Staff` to the common parent team `User` (see users diagram in Fig. 9). Then, the action eliminates redundancies in the permission assignments, and thus permissions from role `Coordinator` to nodes `Information`, `Travel Fundings` and `Forms` are removed because `Coordinator` inherits them from team `User`. The execution of this action takes into account information from different types of diagrams (i.e. from the users diagrams and the access table). The repository, which is the model where the action takes place, gathers all this information.

After applying an action, the user can either confirm the changes so that they are propagated to the rest of the system diagrams by means of the consistency mechanisms integrated in the tool, or discard them and restore the system to the state before the action execution.

6. Related work

The necessity of new tools for modernisation and software evolution has been recently recognised by the OMG. Currently, its Architecture-Driven Modernisation (ADM) Task Force [25] is working on the development of standards for meta-data based modernisation tools. Its main goal is to facilitate the analysis, visualisation, refactoring and transformation of existing software systems. With this purpose it has published a Request for Proposal (RFP) for a Metrics Package [26] that aims at the definition of a meta-model enabling the interchange of quantifiable measurements, as well as its categorisation. The metrics package should be flexible enough to adopt any new kind of metric. Similarly, another RFP is expected for the definition of a Refactoring Package concerning the definition of meta-model based refactorings. The present paper contains ideas oriented towards this direction.

The usefulness of generic measurements and redesigns following the approach "*define once, reuse everywhere*" used in this work is reflected in the number of proposals that can be found in the literature. However, they are usually oriented to a specific domain and focused more on the implementation than on the design phase. For example, the Goal Question Metrics approach (GQM) [27] is oriented to the discovery of reusable measurement patterns. Software project goals are identified and refined to lower level characterisation goals, from which the measurement patterns are derived. Nonetheless, the approach does not provide how these patterns can be afterwards categorised and reused in different problems or domains. Closer to our work, [28,29] present meta-model based approaches in order to specify generic metrics for object-oriented systems. They define meta-models that include domain abstract concepts, such as *class* or *attribute*. A generic metric is defined by using the meta-model concepts, and customised for a specific language by mapping the language concepts and the meta-model ones. On the contrary, our framework decouples the metrics meta-model and the language concepts, making the metrics totally independent of the domain and facilitating their integration with any DSVL. Our use of patterns allows a higher level of abstraction and reusability. In addition, our meta-model includes entities modelling actions and its relation to metrics, making it more complete for software remodelling. In [28], Mens presents an interesting mechanism for metrics extension and composition, which is similar to our concept of dependency between metrics. Finally, it is also worth mentioning the attempts to define ontologies for software measurements [12,18]. Our work on the

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

definition of SLAMMER was inspired by some of this related research.

With respect to generic refactoring, [30] presents this concept together with an implementation based on functional strategic programming in Haskell. The framework consists of meta-programs that can be instantiated for different programming languages by means of parameter passing. As in our approach, the generic refactorings can be applied to any kind of language. However, the parameterisation part seems rather complex and implies knowledge of Haskell and the abstract syntax of the specific language. Seeking candidate code to refactor is a time-consuming task and is not guided by mechanisms helping in its location (thresholds in our framework). Refactoring is never guided by the user, thus making the introduction of inadverted errors possible. Ref. [3] presents a DSVL for domain model evolution. Domain models are represented using a tree structure, and its evolution is expressed as a rewriting operation that gives as a result the tree conforming to the new evolved language. Transformations are expressed as a sequence of *Transforms* establishing a mapping between the objects before and after the evolution. Mappings are expressed as patterns of object diagrams relating the classes in the initial and evolved meta-models. Modifications can result in changes to the semantics or the removal/replacement of existing syntactical patterns.

Different techniques can be used in order to guide the application of redesigns by detecting *bad smells*, such as the study of invariants [31], logic meta-programming [32] or measurements [17]. The last approach is one followed in this paper. The use of measurements seems to be a lighter technique that allows narrowing the search for big systems and smarter strategies for certain types of bad smells.

Regarding tools, the necessity of new modernisation interoperable tools has been recognised by the OMG with a RFP [26] for metrics and refactoring packages. Although there is a variety of modelling tools that incorporate functionalities for obtaining metrics [33–35], the set of metrics they provide is usually hard-coded, oriented to a specific language, and the possibilities of extension are very limited. Some exceptions exist that cover some of the mentioned limitations. For example, the *SDMetrics* tool [34] allows the definition of metrics for UML models using a relational-like language based on XML. Another exception is the Moose Reengineering Environment [33], which implements an engine for language-independent object-oriented software metrics. It provides more than 30 predefined software object-oriented metrics with no possibility of extension, but customisable for any object-oriented language by its mapping to the language independent representation FAMIX. In both cases, our approach is more general, as we are neither restricted to UML nor object orientation, but we can define metrics for any DSVL. In addition, SLAMMER is visual, allowing the customisation of metrics in a graphical, declarative and intuitive way. Moreover, we can define actions (also visually) to be executed when certain thresholds are reached, in order to improve the metric value.

Similarly, there is an increasing set of tools incorporating refactoring or redesign capabilities, although again they are usually oriented to a specific language, where the parts of the application that should be refactored have to be detected by hand, and with a set of predefined refactoring which is not extensible by the user (e.g. the Refactoring Browser [36] for Smalltalk code, Together [35] for Java code and UML models). There are very few tools allowing an automatic detection of refactoring opportunities. One example is SOUL [32], a logic meta-programming language built on the VisualWorks Smalltalk environment that automatically detects existing bad smells (by using logic meta-programming) from which the set of appropriate refactorings is proposed. Again, this tool is domain specific and the set of bad smells that can be detected, as well as the refactorings proposed, are hard-coded and cannot be enhanced.

In the area of meta-CASE tools, our work is also original. There is a plethora of this kind of tools (such as GME [37] or MetaEdit+ [2]), but to our knowledge none of them gives helpful support to the definition of metrics. Although the Eclipse GMF [38] provides a plug-in for specifying measurements by using OCL, metrics have to be programmed from scratch, making the process tedious, hard and time-consuming. In addition, it does not support advanced features such as report definition or guided actions.

In the area of visual editors, the field is moving towards an easy specification and generation of richer modelling tools for DSVLs. There are many approaches for the generation of tools, which are merely visual editors. However, the MDD process needs more functional tools, integrating for example quality control aspects. Some tools (e.g. OpenArchitectureWare [4,39], which, however, does not

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

provide support for DSVLs) are moving towards this direction by integrating a number of additional tools helping in common MDD tasks, such as code generation, model transformation and reporting. The fact that some of these tools are integrated in the Eclipse framework may make easier the interoperability with further tools. However, it is our view that all these related tools have to be customised (probably using the DSVL meta-model as the core of the customisation) and tightly integrated for the given domain.

Altogether our approach has the advantage of being language independent, easily customisable for arbitrary DSVLs. The customisation is done visually and in addition, actions can be specified using the declarative and visual approach of graph transformation or by customising generic task templates by graphical patterns. Concerning tool support, we have integrated SLAMMER with the meta-modelling tool AToM^3. This integration allows the generation of richer modelling environments for (multi-view) DSVLs, which are provided with tools for taking measurements and performing redesigns.

7. Conclusions and future work

The contributions of this work are the following. First, we have proposed a novel approach for the specification of measurements oriented to DSVLs, and based on graphical patterns. The approach reduces the cost of defining domain specific measurements, and allows generating domain specific measurement tools starting from these high-level specifications. Second, the concept of customisable, high-level redesigns is also novel. Third, both concepts have been formalised in the SLAMMER DSVL. In this language, generic measurements are customised for a given DSVL by means of visual patterns, and actions are specified either by customising some predefined template, by graph transformation rules, or by procedural code. Actions can implement model redesigns with or without user intervention, and may be invoked either directly or as result of some measurement threshold value. Fourth, we have provided an implementation of these ideas in the meta-modelling tool AToM^3. This allows the automatic generation of richer modelling environments for DSVLs, which are now provided with tools that measure and redesign models, where redesigns can be guided by specific values obtained from measurements. Moreover, we have shown the applicability of the framework for multi-view DSVLs. In this case, taking measurements and performing redesigns usually span several diagrams, therefore they take place in the repository. To our knowledge, no other meta-modelling tool offers these capabilities.

In the future, we intend to include new general measurements and redesigns in the language. It can also be interesting to study how to support other kinds of measurements, for example subjective and dynamic ones. The latter can be suitable in case of having executable models, with a precise operational semantics. In addition, we are starting the study of mechanisms for richer customisable template tasks. Providing further analysis tools (e.g. statistical) for studying the results, as well as more powerful visualisation facilities for the results, is also up to future work.

Acknowledgements

Work supported by projects MODUWEB (TIN2006-09678) and MOSAIC (TIC2005-08225-C07-06) of the Spanish Ministry of Science and Education.

References

[1] P. Díaz, S. Montero, I. Aedo, Modeling hypermedia and web applications: the Ariadne Development Method, Information Systems 30 (8) (2005) 649–673.
[2] R. Pohjonen, J.-P. Tolvanen, Automated production of family members: lessons learned, in: Proceedings of PLEES'02, Seattle, USA, 2002, pp. 49–57.
[3] J. Sprinkle, G. Karsai, A domain-specific visual language for domain model evolution, Journal of Visual Languages and Computing 15 (3–4) (2004) 291–307.
[4] M. Völter, T. Stahl, Model-Driven Software Development, Wiley, New York, 2006.
[5] N.E. Fenton, S.L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, second ed., PWS, 1998.
[6] S.A. Whitmire, Object Oriented Design Measurement, Wiley Computer Publishing, Wiley, New York, 1997.
[7] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns, Elements of Reusable Object-Oriented Software, Professional Computing Series, Addison-Wesley, Reading, MA, 1995.
[8] J. Kerievsky, Refactoring to Patterns, Addison-Wesley, Reading, MA, 2004.
[9] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer, Fundamentals of Algebraic Graph Transformation, Monographs in Theoretical Computer Science, Springer, Berlin, 2006.
[10] P. Díaz, I. Aedo, F. Panetsos, Modeling the dynamic behavior of hypermedia applications, IEEE Transactions on Software Engineering 27 (6) (2001) 550–572.

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002

[11] V.R. Basili, G. Caldiera, H.D. Rombach, Goal Question Metric Paradigm, Encyclopedia of Software Engineering, Wiley, New York, 1994, pp. 528–532.
[12] F. García, M.F. Bertoa, C. Calero, A. Vallecillo, F. Ruíz, M. Piattini, M. Genero, Towards a consistent terminology for software measurement, Information and Software Technology 48 (2006) 631–644.
[13] T. Mens, On the Use of Graph Transformations for Model Refactoring, Generative and Transformational Techniques in Software Engineering, 2006, pp. 219–257
[14] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, Reading, MA, 1999 See also: ⟨www.refactoring.com⟩.
[15] M.J. Munro, Product metrics for automatic identification of bad smell design problems in Java source-code, in: Proceedings of 11th International Software Metrics Symposium (METRICS 2005), IEEE Computer Society, Silver Spring, MD, 2005.
[16] ISO/IEC 15939: 2002 Software Engineering—Software Measurement Process.
[17] F. Simon, S. Löffler, C. Lewerentz, Distance based cohesion measuring, in: Proceedings of the Second European Software Measurement Conference, 1999, pp. 69–83.
[18] M.A. Martín, L. Olsina, Towards an ontology for software metrics and indicators as the foundation for a cataloging web system, in: Proceedings of LA-WEB, IEEE Computer Society, Silver Spring, MD, 2003.
[19] F.W. Calliss, Problems with automatic restructurers, SIGPLAN Notices 23 (3) (1988) 13–21.
[20] S. Abrahao, N. Condori-Fernández, L. Olsina, O. Pastor, Defining and validating metrics for navigational models, in: Proceedings of the Ninth International Software Metrics Symposium, 2003, pp. 200–210.
[21] R.A. Botafogo, E. Rivlin, B. Shneiderman, Structural analysis of hypertexts: identifying hierarchies and useful metrics, ACM Transactions on Internation System 10 (2) (1992) 142–180.
[22] E. Guerra, J. de Lara, Model view management with triple graph transformation systems, in: Proceedings of the ICGT'06, Lecture Notes in Computer Science, vol. 4178, Springer, Berlin, 2006, pp. 351–366.
[23] J. de Lara, H. Vangheluwe, AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling, FASE'02, Lecture Notes in Computer Science, vol. 2306, Springer, Berlin, 2002, pp. 174–188, See also the AToM^3 home page at: ⟨http://atom3.cs.mcgill.ca⟩.
[24] E. Guerra, P. Díaz, J. de Lara, A formal approach to the generation of visual language environments supporting multiple views, in: Proceedings of the VL/HCC'05, IEEE Computer Society, Silver Spring, MD, 2005, pp. 284–286.
[25] Architecture-Driven Modernization, Home page: ⟨http://adm.omg.org⟩.
[26] Request for Proposal of the ADM: Metrics Package ⟨http://www.omg.org/docs/admtf/06-09-01.pdf⟩.
[27] M. Lindvall, P. Donzelli, S. Asgari, V. Basili, Towards reusable measurement patterns, in: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05), 2005, pp. 21–28.
[28] T. Mens, M. Lanza, A Graph-Based Metamodel for Object-Oriented Software Metrics, Electronic Notes in Theoretical Computer Science, vol. 72(2), 2002.
[29] V.B. Mišić, S. Moser, From formal metamodels to metrics: an object-oriented approach, in: Proceedings of TOOLS, 1997, pp. 330–339.
[30] R. Lämmel, Towards generic refactoring, ACM SIGPLAN Workshop on Rule-based Programming, ACM Press, New York, 2002, pp. 15–28.
[31] Y. Kataoka, M.D. Ernst, W.G. Griswold, D. Norkin, Automated support for program refactoring using invariants, in: Proceedings of the International Conference on Software Maintenance, 2001, pp. 736–743.
[32] T. Tourwé, T. Mens, Identifying refactoring opportunities using logic meta programming, in: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering, 2003, pp. 91–100.
[33] M. Lanza, S. Ducasse, Beyond language independent object-oriented metrics: model independent metrics, in: Proceedings of QAOOSE'02, pp. 77–84.
[34] SDMetrics, Home page: ⟨http://www.sdmetrics.com⟩.
[35] Together Technologies, Home page: ⟨http://www.borland.com/us/products/together⟩.
[36] D. Roberts, J. Brant, R. Johnson, A refactoring tool for smalltalk, Theory and Practice of Object Systems 3 (1997) 253–263.
[37] A. Lédczi, A. Bakay, M. Marói, P. Vögyesi, G. Nordstrom, J. Sprinkle, G. Karsai, Composing domain-specific design environments, IEEE Computers (2001) 44–51.
[38] The Eclipse Graphical Modeling Framework, Home page: ⟨http://www.eclipse.org/gmf⟩.
[39] OpenArchitectureWare, Home page at: ⟨http://www.openarchitectureware.org/⟩.

Please cite this article as: E. Guerra, et al., Visual specification of measurements and redesigns for domain specific visual languages, Journal of Visual Language and Computing (2007), doi:10.1016/j.jvlc.2007.09.002