Integrating Measures and Redesigns in the Definition of Domain Specific Visual Languages

Esther Guerra[1], Juan de Lara[2], Paloma Díaz[1]
[1]Ingeniería Informática, Universidad Carlos III (Madrid, Spain)
[2]Ingeniería Informática, Universidad Autónoma de Madrid (Madrid, Spain)
eguerra@inf.uc3m.es, jdelara@uam.es, pdp@inf.uc3m.es

Abstract

The goal of this work is to facilitate the task of integrating measurement and redesign tools in modelling environments for Domain Specific Visual Languages (DSVLs), reducing or eliminating the necessity of coding. With this purpose, we have created a DSVL called SLAMMER that includes generalizations of some of the more used types of product metrics and frequent model manipulations, which can be easily customised for any other DSVL in a graphical way. The metric customisation process relies on visual patterns for the specification of the elements that should be measured in each metric type, while redesigns (as well as other actions) can be specified either personalizing generic templates or by means of graph transformation systems. The provided DSVL also allows creating new metrics, composing metrics, and executing actions guided by measurement values.

The approach has been empirically validated by its implementation in a meta-modelling tool, which has been used for several DSVLs. In this way, together with the DSVL specification, a SLAMMER model can be provided containing a suite of metrics and actions that will become available in the final modelling environment. In this chapter we show a case study for a notation in the web engineering domain.

As ensuring model quality is a key success factor in many computer science areas, even crucial in model-driven development, we believe that the results of this work benefit all of them by providing automatic support for the specification, generation and integration of measurement and redesign tools with modelling environments.

Keywords: Quality, Metrics, Redesign, Domain Specific Visual Language, Meta-Modelling, Graph Transformation.

Introduction

Diagrammatic notations are pervasive in software development, e.g. to specify, understand and reason about the system to be built. When the notations are constrained to a particular application domain, they are called Domain Specific Visual Languages (DSVLs) (Gray et al., 2004). These provide high-level, domain-specific, graphical primitives, having the potential to increase the user productivity for the specific modelling task. Being so restrictive they are less error-prone than general-purpose languages, and easier to learn.

DSVLs are frequently used in Model-Driven Software Development (MDSD) (Kent, 2002) as a means to capitalize the knowledge in a certain application domain. MDSD seeks increasing quality and productivity in software development by considering models as the primary asset, from which the application code is generated. Although its steep learning curve has been pointed out as one of its main disadvantages, its benefits outweigh the drawbacks, and the use of appropriate modelling tools can help developers to overcome this and other problems. Thus, many efforts are being currently spent in order to provide adequate tool support for the specification and generation of rich modelling environments for DSVLs (DSLTools, 2007; GMF, 2007; Lédczi et al., 2001; Pohjonen & Tolvanen, 2002) encompassing aspects of the

MDSD process, such as facilities for code generation, reporting, formal verification, or quality assessment (Guerra et al., 2006), which is the topic of the present chapter.

Software quality is defined as "*the totality of features and characteristics of a software product that bear on its ability to satisfy stated or implied needs*" (ISO/IEC 9126, 1991). By stated needs we refer to explicit system requirements, mostly functional. Quality features of this type are product correctness, completeness and reliability, and the use of formal methods can help to achieve them. Implied needs are those ones that, although may be incomplete or not specified, if they are not present in the final product then this is considered to have less quality. Some features of this type are efficiency, usability, maintainability, extensibility or cohesion. Product metrics (Fenton, 1996) measure such features in order to control and improve the quality of software products. In this chapter, we are interested in generating tools to measure the quality of software system designs specified using any arbitrary (domain specific) visual notation. We will use the term "*model quality*" to refer to the quality properties of the software system that a model represents. Note that, as in MDSD code is generated from models, it is natural to lift up the mechanisms to check the quality and correctness of applications from code to models.

However, even if measurement is a key quality control activity in most engineering domains (Basili et al., 1994; Whitmire, 1997), this is sometimes neglected in Software Engineering. A factor that may attract a more widespread use is its support by tools, which is even more critical for automation-based processes such as MDSD. Its use helps detecting defects prior to implementation, saving time and budget. The problem is that adapting, implementing and integrating measurement mechanisms for the plethora of DSVLs and tools is costly and time-consuming, and usually does not take advantage of previous developments. Our goal is to reduce such cost, by making the customisation of measures for any kind of DSVL easy.

Additional techniques to enhance system quality from its very design are redesigns and design patterns. Redesigns are design modifications that do not change the functionality but improve model quality. This concept is similar to the concept of refactoring for code (Fowler, 1999). Design patterns (Gamma et al., 1995) are a catalog of best practices that can be applied in order to solve specific problems in software design. Again, the proliferation of notations and tools can hamper the automated application of redesigns and the use of patterns.

In this chapter we propose a novel DSVL called SLAMMER (Specification LAnguage for Modelling MEasures and Redesigns). The language allows the customisation of general predefined measures and actions to be applied to a specific DSVL. Measurement and redesign tools are automatically generated from SLAMMER models and integrated in the DSVL modelling environment. SLAMMER contains the main types of product metrics we have identified. The user can customise these metrics with visual patterns or create new ones. In addition, it is possible to specify threshold values for the metrics. Thresholds may have an associated action described either using a programming language, a graph transformation system (Ehrig et al., 2006) or customising a generic predefined template. This is useful if the action executes known redesigns that improve the model quality.

These ideas have been implemented in the AToM^3 tool (de Lara & Vangheluwe, 2002), which allows the description of DSVLs by means of meta-modelling. We illustrate its use by defining a set of metrics and redesigns for Labyrinth (Díaz et al., 2001), a DSVL in the web domain.

Chapter organization. The chapter starts by studying related work. Then, it gives an introduction on meta-modelling for the generation of environments for DSVLs, and presents an example of environment generation for Labyrinth. Next section introduces the main concepts of measurement and redesign. Then, SLAMMER is presented using examples with Labyrinth. After that, we show how SLAMMER was integrated in AToM^3 and used to improve the environment for Labyrinth. Then, some methodological issues are discussed, regarding the use of these concepts in MDSD. Finally, the chapter ends with future trends and the conclusions.

State of the Art

As stated in the introduction, the purpose of the work presented in this chapter is to facilitate the generation of visual environments integrating mechanisms to quantify and improve model quality, regardless of the DSVL in which these models are specified. Therefore, the required mechanisms must be general enough to be reused or adapted to any notation. In this respect, some proposals for generic measurement and redesign are found in the literature, although they are usually oriented to a specific domain and focused on the implementation phase. For example (Mens & Lanza, 2002; Misic & Moser, 1997) 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. However, these approaches are domain dependent as the calculation of the metrics depends on the concepts defined on the "generic" meta-model. They don't exploit metrics as software remodelling tools that allow guiding redesign execution either. The approach followed in SPQR/20 (SPQR/20, 1995) also provides an implementation of the measurement function (an extended version of function points) applicable to different languages. Finally, it is also worth mentioning the attempts to define ontologies for software measurement (García et al., 2006; Martín & Olsina, 2003).

With respect to the notion of generic refactoring, this is presented in (Lämmel, 2002). The framework consists of meta-programs written in Haskell that can be instantiated for different programming languages by means of parameters. However, the parameterisation is complex and implies knowing Haskell and the abstract syntax of the specific language. Search of candidate code to refactoring is exhaustive (consuming-time) and not guided by mechanisms that help to guide its application by detecting bad smells.

Recently, the necessity of new tools for modernization and evolution of software has been recognised by the OMG with its Architecture-Driven Modernization (ADM) Task Force. It has published a Request for Proposal (RFP) for Metrics and Refactoring Packages with the purpose of defining a meta-model that enables the interchange of metrics and refactorings, respectively, being flexible enough to adopt any new kind of metric. Its main goal is to facilitate the analysis, visualization, refactoring and transformation of existing software systems.

There are a variety of modelling tools that incorporate functionalities for obtaining measurements. Nonetheless, the provided metrics are usually hard-coded, oriented to a specific domain, and the extension possibilities are very limited. One exception is the SDMetric tool (SDMetric), which allows the definition of metrics for UML models using a relational-like language based on XML. In ATHENA (Tsalidis et al., 1992) the set of predefined metrics can be extended by using a textual language. The Moose Reengineering Environment (Lanza & Ducasse, 2002) 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 that can be customised for any object-oriented language by its mapping to a language independent representation called FAMIX. As it can be seen, there is a need of more general approaches neither restricted to UML nor object orientation, being more easily adaptable and intuitive.

Regarding redesign capabilities, the ones provided by modelling tools are usually oriented to a specific language, with no possibility of extension, and the parts that need to be redesigned have to be detected by hand (e.g. the Refactoring Browser (Roberts et al., 1997) for Smalltalk code or Together Technologies for Java and UML models). There are only a few that allow an automatic detection of model refactoring opportunities, such as SOUL (Tourwé & Mens, 2003). This is a language built on the VisualWorks Smalltalk environment that detects existing bad smells by using logic meta-programming, and then proposes a set of appropriate refactorings

that can solve them. Again, this tool is domain specific and the set of bad smells and refactorings cannot be enhanced.

In the area of meta-CASE tools, although there is a plethora of them (e.g. GME (Lédczi et al., 2001), MetaEdit+ (Pohjonen & Tolvanen, 2002) or the Eclipse Generic Modelling Framework (GMF)), to our knowledge none of them support the definition and customisation of metrics. Even though GMF provides a "metrics" package, it only allows defining metrics from scratch by coding them in OCL, making the process tedious, hard and time consuming. In order to define redesigns, some of them provide some transformation language, but in any case they do not provide support for the detection of the parts that should be reworked.

Meta-Modelling for Domain Specific Visual Languages

A meta-model is a model of a modelling language (Favre, 2004). That is, in order to describe a modelling language, one can make a model (e.g. using class or entity relationship diagrams) to describe the language *abstract syntax*. This contains the main concepts of the language and their relations. In addition, in order to restrict the number of valid models defined by meta-models, they may contain additional constraints expressed in textual languages such as OCL (Warmer & Kleppe, 2003).

As an example, Figure 1 shows an excerpt of the meta-model for Labyrinth, a DSVL oriented to the design of web applications (Díaz et al., 2001). In Labyrinth, a web application is modelled as a set of nodes where contents are located. Nodes and contents can be composed in order to create complex information structures. Navigation is expressed through anchors and links: a link defines a possible navigation path between nodes or contents, and the source and target of a link is defined through anchors. 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. These 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 members.

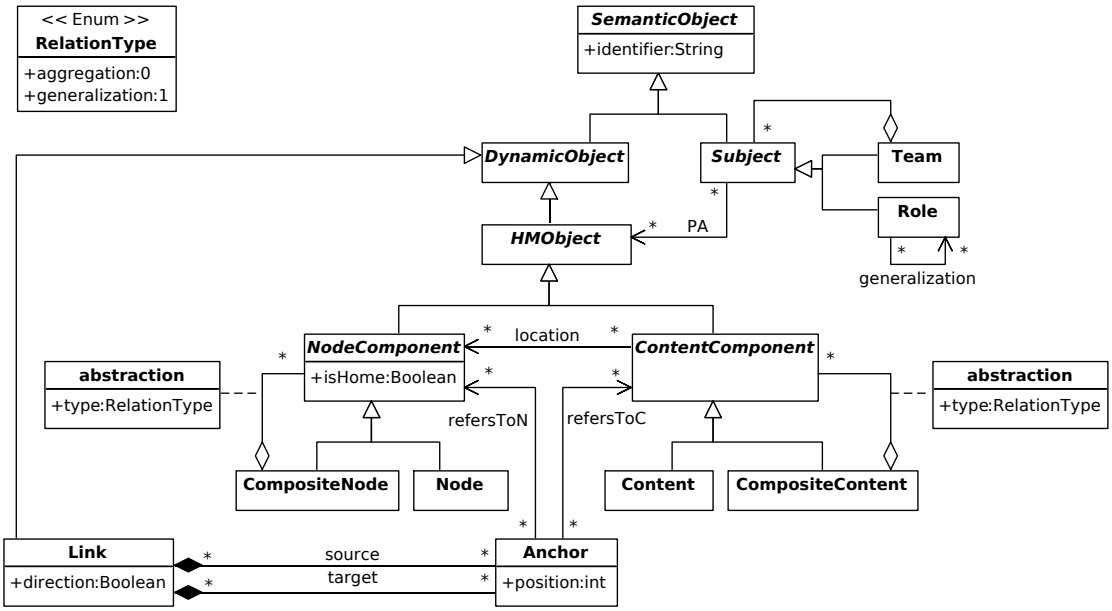

Figure 1: An Excerpt of the Labyrinth Meta-model

The meta-model of a DSVL has to be provided with information about the visualization of each one of its elements, which is known as its *concrete syntax* (de Lara & Vangheluwe, 2002). The simplest way is to assign an icon-like visualization to classes and arrow-like to associations.

Meta-modelling tools allow specifying the concrete and abstract syntax of a certain DSVL, and they automatically generate a modelling tool where end-users are allowed to edit models written in such notation. In this chapter, our purpose is to provide a mechanism to enrich such generated environment with capabilities for model quality measurement and improvement.

Multi-View Domain Specific Visual Languages

As systems become more complex, there is a trend to split their specification in smaller models, each one of them built by using the most appropriate notation. The family of notations that are used in combination for the description of the aspects of a system is called Multi-View DSVL (MV-DSVL). UML (UML, 2006) is one of its most prominent examples, although for a broader domain. It provides different diagram types for the specification of the static (e.g. class and object diagrams) and dynamics (e.g. statecharts and sequence diagrams) of a system. Similarly, the Ariadne Development Method (Díaz et al., 2005) defines a set of diagram types based on the Labyrinth meta-model to deal with various concerns of a web design, such as the information structure, navigation paths, presentation features and access control policies.

Modelling environments for MV-DSVLs must ensure not only intra-diagram consistency (i.e. conformance of a model to its meta-model), but also inter-diagram consistency for those cases when the same element belongs to different diagrams, therefore changes in one of them should be propagated to the others. Our approach (implemented in AToM^3) for the specification of such environments is to first define the meta-model of the complete language, and then define each diagram type as a subset of it (Guerra & de Lara, 2007). From this specification, a multi-view environment is generated where the end-user builds models conforming to some diagram type. Inter-diagram consistency is achieved by building a *repository* made of the gluing of the system models, from where changes are propagated to the rest of the views, as done in the Model-View-Controller pattern. This behaviour is performed by triple graph transformation (TGT) rules (Schürr, 1994) derived from the meta-model information (Guerra & de Lara, 2006). The generated multi-view environment can also check the inter-diagram semantic consistency by translating the repository into a semantic domain, executing an analysis method, and back-annotating the results into the original notation (Guerra et al., 2007).

For example, Figure 2 shows the generated multi-view modelling environment for Labyrinth by using AToM^3. The background window allows defining system diagrams of different types. One diagram called `Role Hierarchy` of type Users Diagram is being edited. The control dialog (named “Edit value”) allows setting the property values for this “view” of the system, including the corresponding model (i.e. a role hierarchy), which is shown in the right-most window.

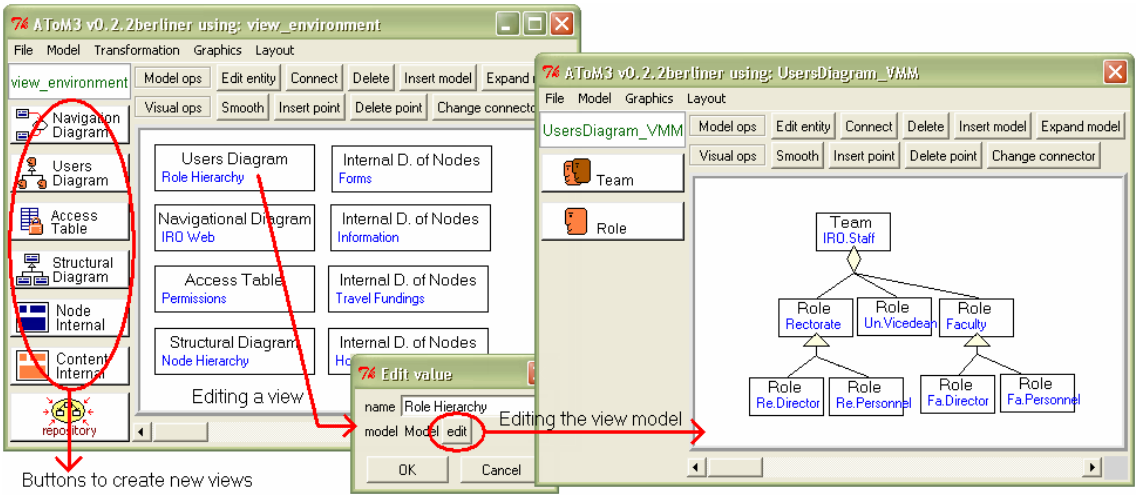

Figure 2: Generated Multi-View Modelling Environment for Labyrinth

In multi-view environments, measurement becomes more complex because the information needed for its calculation is scattered in several models (of the same or different type). Similarly, certain redesigns or model refactorings may imply parallel modifications to several of the system models. Finally, after a redesign, changes should be appropriately propagated to the rest of the models so as to recover the inter-diagram consistency.

In following sections we present our proposal for the definition of measures and redesigns for single and multi-view DSVLs, and show how using it for enriching the previously presented environment for Labyrinth. Before, we give an introduction to measurement and redesign.

Measurement and Redesign

Measurement is a basic tool for quality control in many engineering disciplines (Basili et al., 1994; Whitmire, 1997). Engineers make use of measures in order to provide feedback and assist in evaluation, creating a corporate memory and helping answering questions about the object being measured. In software engineering, the measurable objects are usually processes, resources, products (Fenton, 1996) and projects (Whitmire, 1997). Our work is focussed to measuring products, and in particular models, as they are the key concept in MDSD.

Products (and in general any measurable object) contain *internal* and *external* attributes. The former can be measured in terms of the product itself (e.g. its size). External attributes can only be measured with respect to how the product relates to its environment (e.g. its cognitive complexity, usability or maintainability), and are obtained by testing, operating and observing the executable software. Our work is directed to measuring internal attributes, as they apply on the system models instead on the system itself.

Measurement 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. Indirect (or *derived*) measures are obtained by combining several direct or indirect measures. The term *indicator* is sometimes used to refer to indirect measures that have an associated analysis model made of a calculation procedure with decision criteria. The criteria can be a threshold, a target or a pattern used to determine the need for action or further investigation (García et al., 2006). As we will see in next section, SLAMMER supports direct and indirect measures, as well as indicators with thresholds. Thresholds indicate anomalies in the metric values (e.g. extreme values) and may trigger redesigns for improving the quality of the model.

Further classification of measures includes the *objectivity*, that is, whether they involve human (*subjective*) judgement, or they are quantifications based on numerical rules (i.e. *objective* methods). Finally, regarding the *automatization* degree, measurement methods can be *automatic*, *semi-automatic* or *manual*. Our approach is aimed at the automatization of the measurement in tools, thus we only consider objective metrics (as subjective measures cannot be made fully automatic).

Redesigns are changes in a design model for improving some quality attribute, such as understandability, performance, cohesion or coupling. When the redesign preserves the intended meaning (or behaviour) of the model, it is called model refactoring (Mens, 2006). Refactorings (Fowler, 1999) were originally defined as changes to software code in order to make it easier to understand and modify, without changing its observable behaviour. Model refactoring shifts code refactoring techniques to the model level. In MDSD, this is the right abstraction level, as the application code is generated from the models, which is then frequently treated as a “black box” (i.e. the generated code is not manually adapted).

The need for performing refactorings and redesigns is commonly detected through so-called “*bad smells*” (Fowler, 1999). They informally describe some design or code problem, and have a number of associated actions (one or more refactorings) to help in its solution. Some efforts

have been recently placed in formally defining such *smells* through the use of metrics (Munro, 2005). In our proposal, we follow this trend by using thresholds associated to metrics in order to detect product anomalies, and possibly correct them through redesigns. Although automated, these redesigns usually require human supervision, either for additional input or simply for confirming that they are adequate in the given situation.

SLAMMER: Specification LAnguage for Modelling MEasures and Redesigns

SLAMMER is a novel DSVL that tries to facilitate the definition of measures and redesigns for a given DSVL, as well as to provide a framework for the automatic (model-driven) generation of measurement and redesign tools that can be integrated in the final modelling environment for the DSVL. SLAMMER can be used for any kind of DSVL (which may be used for describing structure, behaviour, or any other system perspective). SLAMMER has been defined by means of a meta-model that takes into account related works on ontologies for software measurement (García et al., 2006), as well as on the international standard for software quality ISO 15939 (ISO/IEC 15939, 2002). In addition, it is based on the use of visual techniques (e.g. graphical patterns, graph transformation) to achieve its purposes.

In this section, we start by introducing the concept of graphical pattern and its instantiation in the context of SLAMMER, as patterns will be used to configure measures and redesigns. Then, we present the part of the SLAMMER meta-model for the definition of measures and actions. We illustrate the SLAMMER concepts with examples for Labyrinth.

Graphical Patterns in SLAMMER

In SLAMMER, the simplest form of pattern is a single positive graph. The application of a pattern to a model gives as result all occurrences of the positive graph in the model. The pattern can be initialised with a partial match, given as an argument of the pattern, and the output can be filtered in order to return a subgraph of the positive graph occurrences. Figure 3 shows to the left an example pattern. The positive graph is made of objects *Role* and *Node* related through a permission assignment (relationship *PA*). To the right, the pattern is instantiated in graph G. In step (i) the match is initialised with the role `r1`, which is received as argument. In step (ii) the match is extended to the complete positive graph of the pattern. Two occurrences of the positive graph are found in G: one relating role `r1` to node `n1`, and another one relating it with node `n2`. In step (iii) the matchings are filtered so that only the elements specified as output in the pattern are obtained as result. Thus, as the pattern specified node `n` as the output, only nodes `n1` and `n2` in the matchings are given as result.

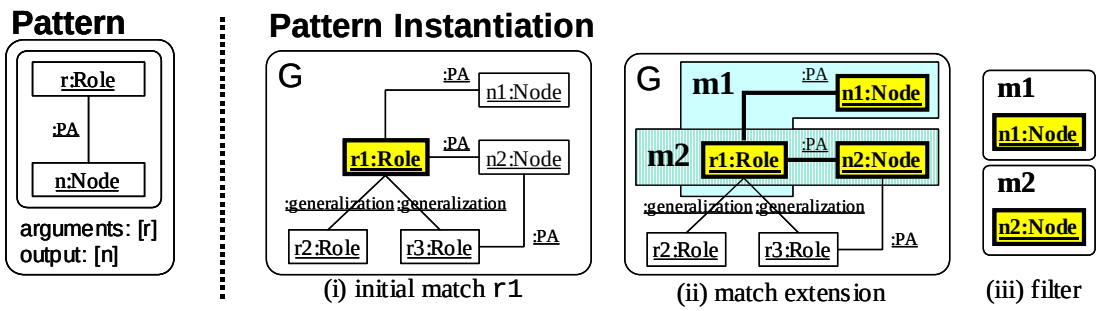

Figure 3: Example of Graph Pattern and Instantiation

The number of instantiations of a pattern can be restricted by means of one or more *application conditions* (Ehrig et al., 2006). These are made of a premise graph and a set of consequence graphs. If a pattern specifies some application condition, the pattern instantiation process is as follows. First, all occurrences of the positive graph are found in the model. Then, for each application condition, if an occurrence of the premise graph is found then some of the

consequence graphs have also to be found for the occurrence of the positive graph to be considered valid. There are two special cases of 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 on the model for the positive graph occurrence to be valid.

Figure 4 shows to the left an example of pattern with two application conditions. Its positive graph is made of an object *Node*, the PAC specifies that an object *Team* must have permission to access the node, and the NAC forbids an object *Role* to have access to the node. To the right, the pattern is instantiated in graph G. In step (i) all the matches of the positive graph are found. As the pattern has no arguments, there is no starting initial match, and thus all nodes in G are valid instantiations of the positive graph. In step (ii) the application conditions are evaluated for each match. An occurrence of the PAC and no occurrence of the NAC are found for match `m1`, therefore the match is valid. For match `m2` no occurrence of the PAC is found, thus the match is discarded. Finally, for match `m3` the PAC is satisfied, but an occurrence of the NAC is found, thus the match is also discarded. This is why in step (iii) only match `m1` is obtained as result.

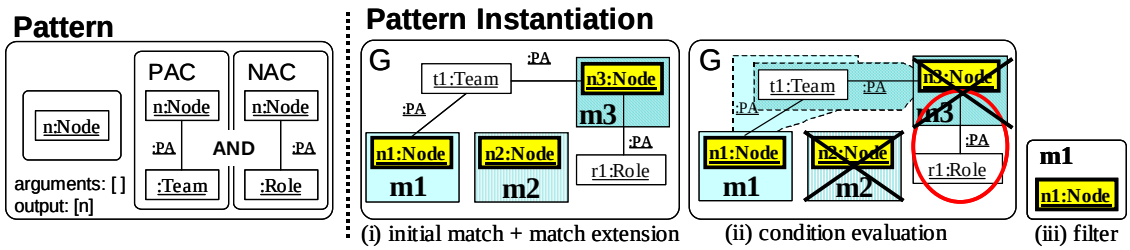

Figure 4: Example of Graph Pattern with Positive and Negative Application Conditions and Instantiation

Figure 5 shows the package of the SLAMMER meta-model dealing with pattern definition. In SLAMMER we use patterns in order to customise generic measures and task templates for concrete DSVLs. Patterns allow visually specifying how model attributes (i.e. features that are going to be measured or modified) are expressed in a DSVL, as next subsection explains.

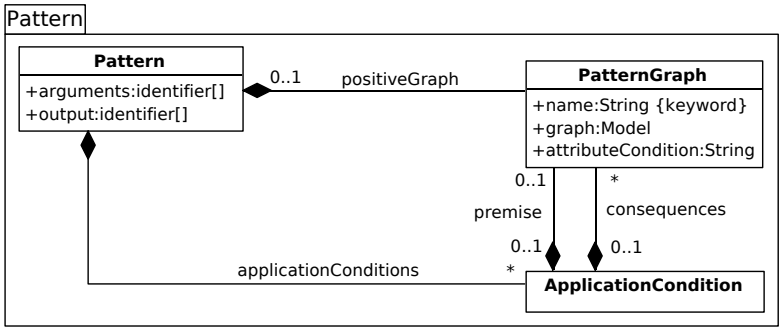

Figure 5: Domain Specific Visual Language SLAMMER. Package “Pattern”

Specification of Measures in SLAMMER

A measure can be specified by providing the set of entities that are going to be characterized by the measurement (the *domain*), the relevant *attributes* for the measurement method, the *measurement method* itself (a function in the case of indirect metrics), the *scale* (the range of values it can take) and, in case of scales of type interval or ratio, a *measurement unit* (e.g. number of classes, lines of code). In addition, measures may include information about normal or unusual value ranges, pointing to *threshold* values in the measurement scale. It must be noted that the measurement method is domain independent and remains always the same. On the contrary, the domain, the properties to be measured and the threshold values are domain

dependent, and have to be specified for each DSVL where we want to perform the measurement. SLAMMER uses this idea in order to specify a set of predefined generic metric templates that hide the measurement function and can be customised by providing only the domain-specific information in each case. The metric domain is specified as the list of types that conform the domain space, the attributes to be measured are given as a set of patterns, the units are given as text, and the thresholds are boolean conditions evaluated on the metric value.

The package of the SLAMMER meta-model concerning the definition of measures is shown in Figure 6. Concrete classes inheriting from class *Measure* define metric templates that can be customised by giving the domain and properties for a specific DSVL. All measures contain a unique identifier *name* and a *goal*. Attribute *domain* is used to specify the metric domain as a list of types. Attribute *subtypeMatching* specifies if objects in the domain must have exactly the type specified in attribute *domain*, or also any of its subtypes is allowed. This makes measures 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 measure can take and its magnitude, respectively. In addition, relation *dependency* allows a measure to use results calculated by other ones and thus metrics composition. In this way, measures can be reused and composed in order to build more complex composite ones. A meta-model constraint forbids cycles of recursive dependencies. A measure may have 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.

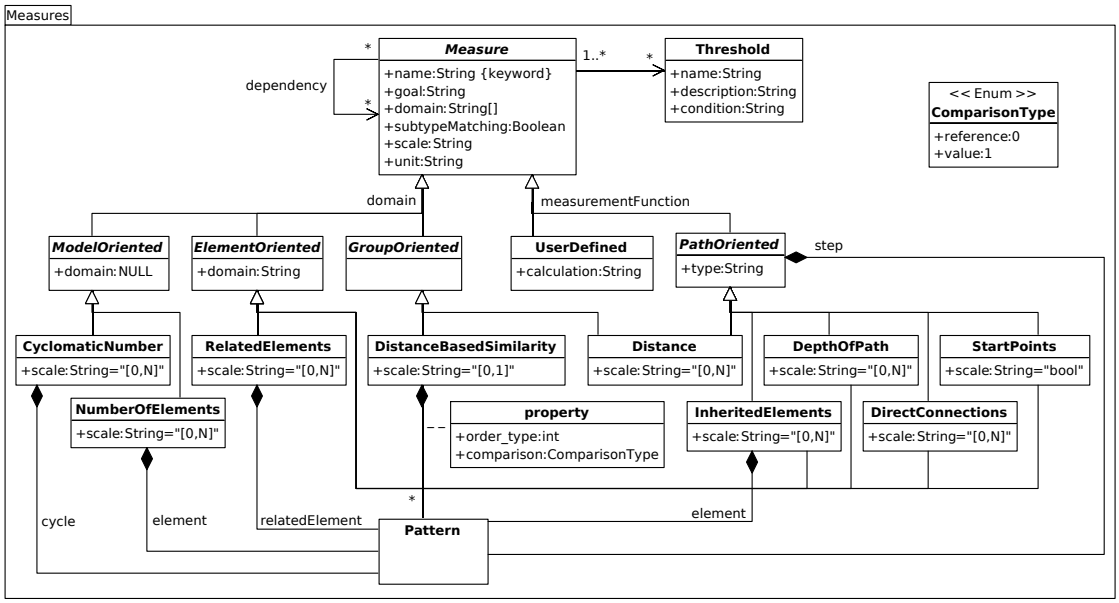

Figure 6: Domain Specific Visual Language SLAMMER. Package “Measures”

Concrete measures in the SLAMMER meta-model are organized depending on its domain dimension and on the measurement function used to calculate the metric value. From the domain dimension point of view, they can be *model-oriented* if they take measures of global model properties (such as number of cycles and size); *element-oriented* if they refer to element features (e.g. permissions assigned to a role); and *group-oriented* if they measure features of groups of elements (e.g. their similarity or coupling). From the measurement function point of view, we sort out them either as *path-oriented* if they use a measurement function that traverses paths between elements of the same type (e.g. a navigation path joins nodes by means of anchors and links, and an inheritance path joins subjects or classes by means of inheritance relations) or any of their subtypes, which is specified by attribute *type*; or as *user-defined* if the measurement function is provided by the user (and is different from the ones already provided by the SLAMMER meta-model).

SLAMMER contains generalizations or abstractions of some of the more used types of metrics in software engineering, together with mechanisms for their combination. That is, we are not inventing new metrics, but reusing metrics that have been validated by other researchers and shown to work for specific purposes. In SLAMMER metrics are visually customised for a given DSVL by means of graphical patterns (class *Pattern* in the meta-model) that identify how domain specific features are expressed in such language. The arguments of the pattern correspond to a value in the metric domain, and the output is the set of model attributes we want to obtain. In the remaining of this subsection, we explain the generic metrics included in SLAMMER.

NumberOfElements allows counting the number of elements of certain type in a model. This is a model-oriented measure because it calculates a property of the model itself, and thus it is not necessary to specify the domain (i.e. it is the complete model). The type of the element to be counted is given as a pattern. As patterns are indeed models plus application conditions, we can count not only elements of a certain type, but also complex structures made of sets of different related elements.
As an example, the *Number of Navigational Contexts* (NNC) (Abrahao et al., 2003) is used in the web domain as indicator of the navigational model size. In Labyrinth, a navigational context is a node component that participates in a navigational link through the corresponding anchor. We can use SLAMMER in order to adapt the NNC to Labyrinth by customising a measure of type *NumberOfElements* with the pattern shown in Figure 7. This pattern has an application condition which allows counting the number of node components (simple and composite, see Labyrinth meta-model in Figure 1) that are source (consequence graph 1) or target (consequence graph 2) of a navigational link. Thus, one of the consequence graphs of the application condition has to be found, and we indicate it with an "OR". The output of the pattern is the element to be counted, that is, the node component.

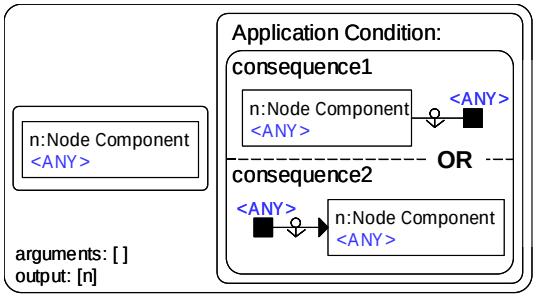

Figure 7: Customisation Pattern for Metric "Number of Navigational Contexts"

CyclomaticNumber counts the number of cycles in a model, thus being model-oriented. In this case a pattern showing the structure of a cycle in the given DSVL must be provided.

RelatedElements counts how many elements are related to a given element type, which is specified by attribute *domain*. This measure is element-oriented, and thus, it is calculated for each element of the specified type in a given model. The relation between the elements is given as a pattern, which allows expressing complex relations made of several elements as well.
For example, we can instantiate a measure of this type for Labyrinth, and customise it so as to count the number of nodes each role has permission to access. In this case attribute domain should contain type "Role", and the related element should be specified by the pattern shown in Figure 3. The metric is calculated for each role in the model and, in each case, the metric value is calculated as the number of times the pattern gets instantiated (two for role `r1`).

DistanceBasedSimilarity compares how similar a set of entities is by studying the set of attributes they share (Simon et al., 1999). It can take values in the interval [0, 1]: the higher the value, the bigger the distance between the entities, and the less similar they are. The types of the entities to compare are given as a list in attribute *domain*. For each one of the types, it must also

be specified which are the properties used for the comparison. This is done with a pattern for each property (qualified relation *property* in the meta-model). The properties define an attribute *order_type* that relates them with the corresponding type in the list given by attribute *domain*. The comparison can be made either by reference (i.e. two objects are considered equal if they are the same) or by value (i.e. two objects are equal if all their fields have the same value).
This measure can be applied to Labyrinth in order to analyse how similar are each two roles in the system, and thus detect redundancies in the defined security policy (Guerra et al., 2006). In this case, the domain contains type *Role* twice and the properties that make similar two roles are the permissions they define (expressed with a pattern).

Distance, as well as the following measures, allows measuring different properties of path-like structures where the nodes in the path have the same type and are connected through some specific relation. For example, the structure of a web navigation map is path-like, since we have information nodes that are connected through anchors and links. Another example is the users' hierarchy provided by Labyrinth, which contains subjects (i.e. roles and teams) connected by means of inheritance relations. In this measure, as well as in the remaining ones, it must be specified the element type to which the measure applies (attribute *domain*), as well as the fundamental step (e.g. the inheritance relationship in the users' hierarchy), which is specified as a pattern. Thus, the measure calculates the minimum number of necessary steps to reach each element from the other ones. From the point of view of the domain dimension, it is a group-oriented metric as it measures a property of a group of two model elements.
For example, Figure 8 shows a pattern specifying what a step is in the Labyrinth navigation map (i.e. two nodes related through a link and two anchors). The target node of a navigation step (output) is the source of the following step (argument). We may use such pattern to customise *Distance* so as to define the Minimum Path Between Navigational Contexts (MPBNC) (Abrahao et al., 2003) for Labyrinth. This gives a measure of the usability of a navigational map by counting the number of links that must be traversed to reach certain information node from another one, and can be used to detect unreachable nodes. In the present example, assigning type "Node Component" as metric domain and selecting subtype matching would complete the customisation process.

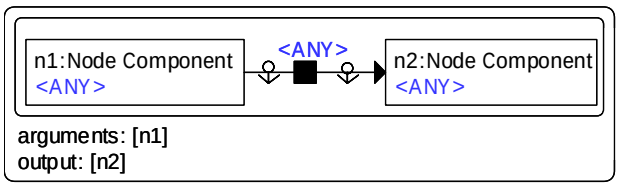

Figure 8: Customisation Pattern for Metric "Minimum Path Between Navigational Contexts"

StartPoints identifies all elements where a path begins, but to which no path arrives. These are the base classes in object-oriented notations.

DepthOfPath counts the minimum number of steps that are necessary in order to reach an element from a starting point. For example, it can be used to calculate the depth of the inheritance tree in object-oriented notations, or the Depth of a Node (D) (Botafogo et al., 1992) in web notations, which is the distance from the root node to a particular node in a navigation map. The bigger the distance, the harder becomes to reach the node. In order to adapt metric D for Labyrinth, it should be specified what a step is in the Labyrinth navigation map, which can be done with the same pattern that was shown in Figure 8.

InheritedElements applies to notations having some concept of inheritance. It calculates how many elements of certain type are inherited through the inheritance hierarchy. In this case, together with the type and the fundamental step, a pattern must be specified with the element to be inherited.
For example, Figure 9 shows the two necessary patterns for the definition of the metric Subject Inherited Permissions, which counts the number of inherited permissions through the hierarchy

of roles and teams defined in Labyrinth. The pattern to the left specifies what a step in such hierarchy is, that is, two subjects joined by either a generalization (consequence graph 1) or an aggregation (consequence graph 2). The pattern to the right indicates which is the inherited element, that is, the permission to access a hypermedia object (i.e. a node or a content).

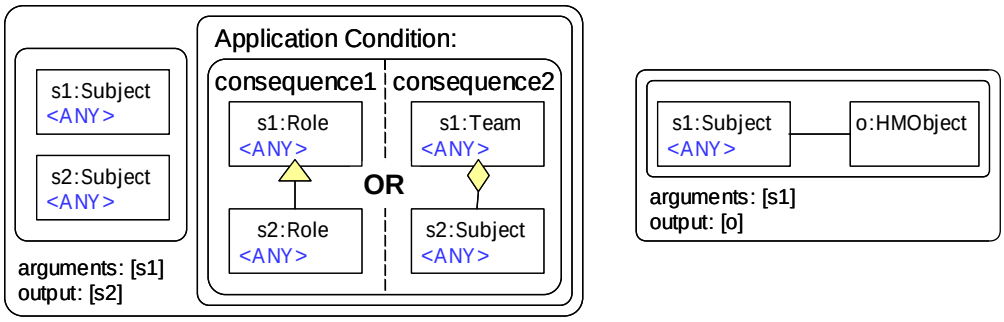

Figure 9: Customisation Patterns for Metric "Subject Inherited Permissions"

Finally, **DirectConnections** calculates the number of elements than can be directly reached in one step in a path-like structure. As before, only the type to which the measure applies as well as the fundamental step must be specified. This measure can be used by Labyrinth, for example, to calculate how many members belong to a team. Note that this information can be scattered in different user diagrams.

Specification of Actions in SLAMMER

Figure 10 shows the portion of SLAMMER dealing with actions. These are usually redesigns, although other tasks (e.g. generating a report or printing a model) can be specified. Actions are made of reusable tasks expressed either procedurally, by means of a graph grammar (Ehrig et al., 2006), or by customising task templates. They can be applied either when some measure reaches certain threshold value (relation *fires*) or directly by the end-user independently from metric values. In the first case, the action is executed for each value in the domain for which the measure makes the threshold condition true. Attribute *execution* in class *Action* selects whether this action is automatically executed, or it needs human supervision to confirm it.

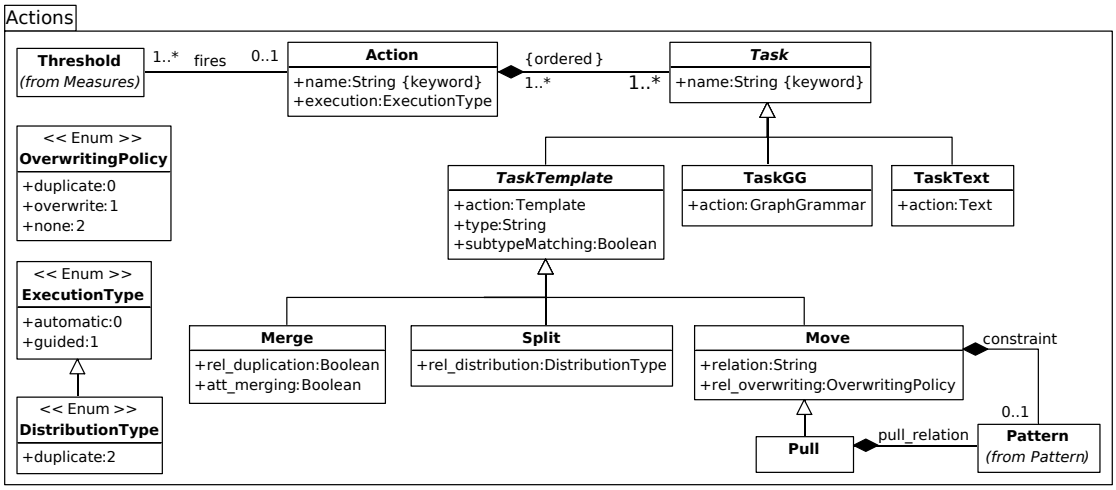

Figure 10: Domain Specific Visual Language SLAMMER. Package "Actions"

SLAMMER defines four customisable tasks: merge, split, move and pull. **Merge** collapses two elements into a single one that brings together all the relationships of the formers. If the original entities defined the same relation, 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 as the concatenation of the original values or taking one of them. For example, this task can be used to compact two consecutive Labyrinth nodes with little information, so as to make the navigation lighter.

Split divides in two an entity of the specified type. Relations of the original element are redistributed between the new ones either randomly in equal parts or guided by the user (controlled by attribute *rel_distribution*). The task could be used, for example, in order to divide nodes with a large amount of information, so as to avoid a cognitive overload to the user.

Move moves relationships between entities of the same type. 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; *overwrite* if the relationship is moved and overwrites the one in the target; and *none* if the relation is not moved. It is possible to restrict the number of relations to be moved by means of a pattern that receives as arguments the elements that take part in the action (i.e. the relation to move and the source and target elements). In this case the action is applied only if the pattern is satisfied.

Finally, **Pull** specializes task *Move* to those cases where the involved entities must be related. The relation is specified as a pattern with the entities as arguments and no output.
As an example, we can customise a task *Pull* for Labyrinth so as to pull up permissions to a parent role if all its direct children already define them. This is a model refactoring with the aim of promoting reuse of permissions by taking advantage of the inheritance concept. The task should be defined for type “Role” and relation “PA” (the one used for permission assignment in the Labyrinth meta-model). In order to pull up a permission, an inheritance relation must exist between the source and target roles, which is specified by the pattern to the left in Figure 11. This pattern corresponds to relation *pull_relation* in the SLAMMER meta-model. In addition, as we only want to pull up those permissions defined by all children roles, we constraint the applicability of the task by means of the pattern to the right in the same figure, which corresponds to relation *constraint* in the SLAMMER meta-model. The pattern receives the permission to move and the source and target roles as input. The application condition checks the existence of such permission in each target role’s child. Note that the model refactoring should be completed with an additional task that removes permissions in children roles if defined by their parents. The second task could be defined by means of a graph grammar, and be combined with the previous task to conform a single action.

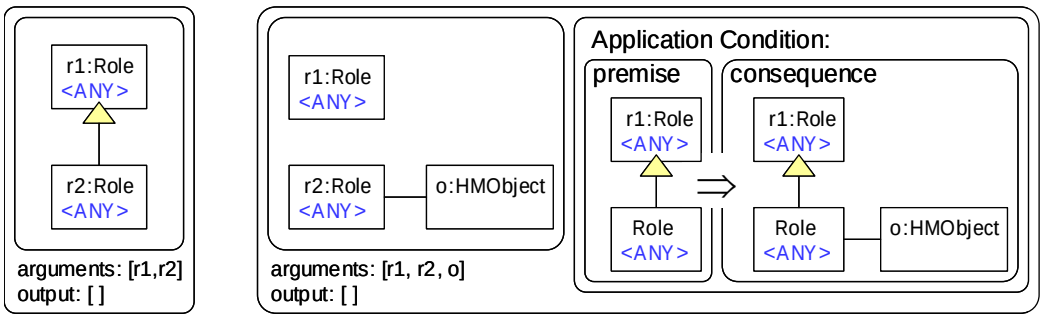

Figure 11: Specification of Pull Task

In order to specify tasks by means of graph transformation we can use *TaskGG* objects. For example, Figure 12 shows a graph grammar task made of a rule that creates a navigational path from the root node of a web design (with attribute *isHome* to true) to a given node which is not root. The elements to be added by the rule application are shown in a coloured polygon and labelled as “new”. These elements form also a NAC, and thus the rule is not applied if such path already exists. We can use this task to create direct links from the home page of a web

application to those nodes that are not reachable or where a high number of navigational steps are required to access them. In addition, it is possible to use a metric to detect to which nodes apply this redesign. For example, a customisation of *DepthOfPath* can be defined so as to count the number of steps to reach any node starting from the home page. Then, if we associate an appropriate threshold value to the metric (e.g. 0, which means that it is not possible to reach the node), we can detect the candidate nodes, and thus automatically fire the action on them.

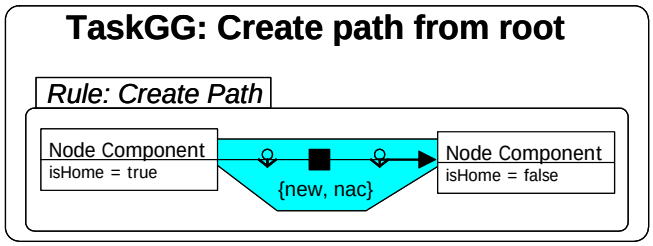

Figure 12: Specification of Graph Grammar Task

Implementation in AToM^3

Starting from the meta-models shown in previous sections, we have built a tool for SLAMMER that allows complementing a DSVL meta-model with a SLAMMER model, and generating a measurement and redesign tool for the given DSVL. For this purpose we took advantage of the code generation capabilities provided by AToM^3. Thus, we defined the SLAMMER meta-model in AToM^3, and automatically obtained a tool for building SLAMMER models. A code generator that synthesizes tools from the SLAMMER models was added to this tool. The synthesized tools generated this way make accessible the defined metrics and actions to the modelling environment generated for the DSVL. Finally, the new tool was integrated into AToM^3 itself.

In order to be able to configure (to a certain degree) the features of the tools generated from the SLAMMER models, we have slightly modified the SLAMMER meta-model previously shown. In particular, we have added an abstract class *UIButton* as the parent of classes *Measure*, *Action* and *Task*. This class has a single boolean attribute *button* that controls whether a button should be generated in the tool user interface in order to execute the corresponding measurement process, action or task. This is useful, for example, in case we want to prevent the direct calculation of a metric that is only used as auxiliary metric by others. In addition, class *Measure* has been provided with additional attributes to allow obtaining PDF reports with all the measurement results, or only the ones making some threshold condition true.

In addition, we have provided SLAMMER with the concrete syntax shown in Figure 13, where five metrics and two actions are being defined by using the generated tool for SLAMMER. In particular, measures are represented as rectangles with the measure type and name inside. Dependencies between measures are represented as arrows, where the arrowhead indicates the data flow direction. For example, in the model of the figure, the result obtained by metric `NNC` is used to calculate metrics `Compactness`, `Stratum` and `DeNM`. Thresholds are shown as triangles with an exclamation mark inside, and related to the measures for which they are defined. Actions are depicted as circles with an arrow inside and the action name below. If its execution mode is automatic, it is shown as a double circle, as in the case of action `Create path from root`. Finally, tasks are visualized as ellipses with the task type and name inside. The tasks that are executed for a given action are related to it by means of lines, with the execution order above.

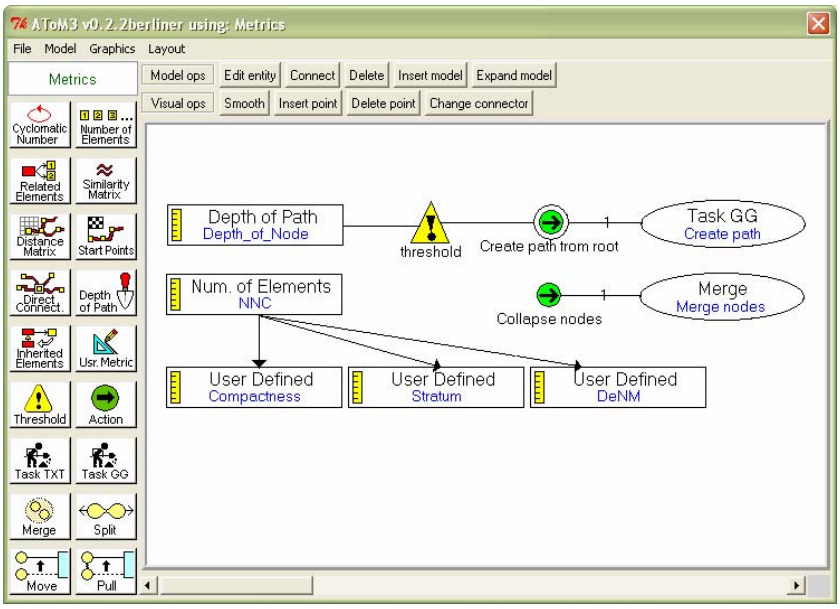

Figure 13: Generated Tool for SLAMMER

Figure 14 summarizes the process of defining, generating and using a modelling tool for a DSVL with AToM^3. The left part of the figure shows the specification of the DSVL by the DSVL designer. In step 1 (of the left part), the DSVL definition is given by a meta-model. In the case of a MV-DSVL, the different diagram types (or viewpoints) have also to be specified. In addition, a quality expert can design a SLAMMER model with the metrics and actions for the particular DSVL. The metrics are usually customisations of the suite offered by SLAMMER, thus 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 with patterns. Although we have separated the roles of defining the DSVL meta-model and the specification of metrics and redesigns, in many occasions it is the same person who performs both activities.

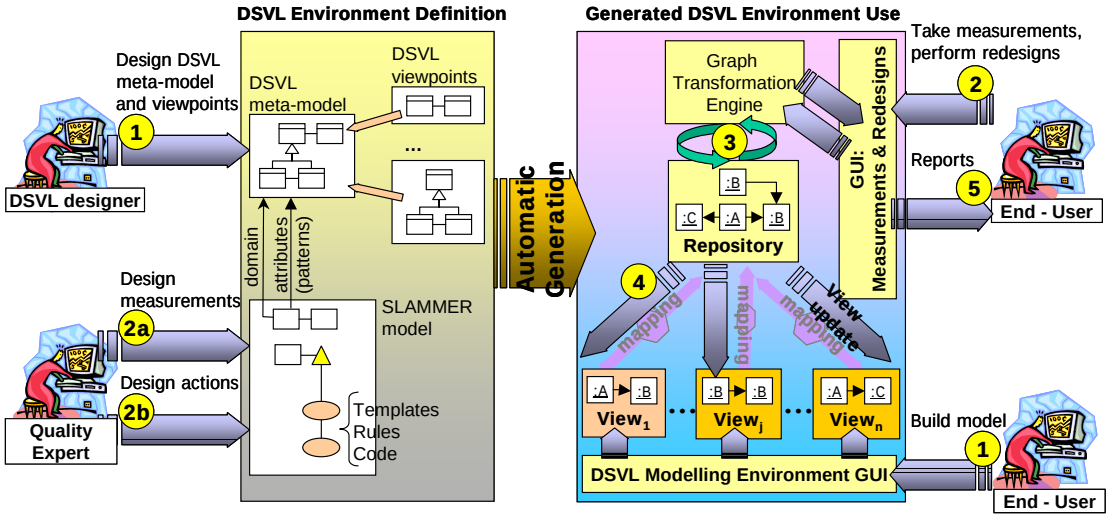

Figure 14: Integrating Measurement and Redesign Tools in Modelling Environments

Starting from this definition, AToM^3 is able to automatically generate a modelling tool for the (MV-)DSVL. The use of such environment is schematised to the right of Figure 14. The end-user interacts with the generated tool user interface in order to build his models (step 1 of the right part). The tool automatically builds a repository with the gluing of the different models (or system views) and provides intra- and inter-diagram consistency. The repository properties can be evaluated (step 2) by using the metric specifications provided by the quality expert during the

definition of the modelling environment, and the results are shown to the user as PDF reports (step 5). Note that measurement is performed in the repository, as it is the only model that contains all the system information. In addition, extreme values of metrics can trigger actions that modify the repository model with the purpose of improving the value of the metrics (step 3). Transforming the repository can leave the system design in an inconsistent state, as some elements can be added, edited or even deleted by the redesign. For this reason, once the redesign has been performed, the changes are propagated by the same consistency TGT rules that provide inter-diagram consistency in multi-view environments (step 4).

Enriching the Labyrinth Environment with Metrics and Redesigns

Figure 15 shows a screenshot of the definition process of metrics and actions for Labyrinth. Window 1 in the background is the tool generated for SLAMMER and contains the metrics and actions defined for Labyrinth. In particular, the figure shows the customisation of the metric named `Depth_Of_Node` of type *DepthOfPath*, which is the upper one to the left in window 1. The metric 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 metric with a pattern. Window 3 contains the definition of the positive graph of such pattern, a navigation step in Labyrinth made of two nodes joined by a link and two anchors.

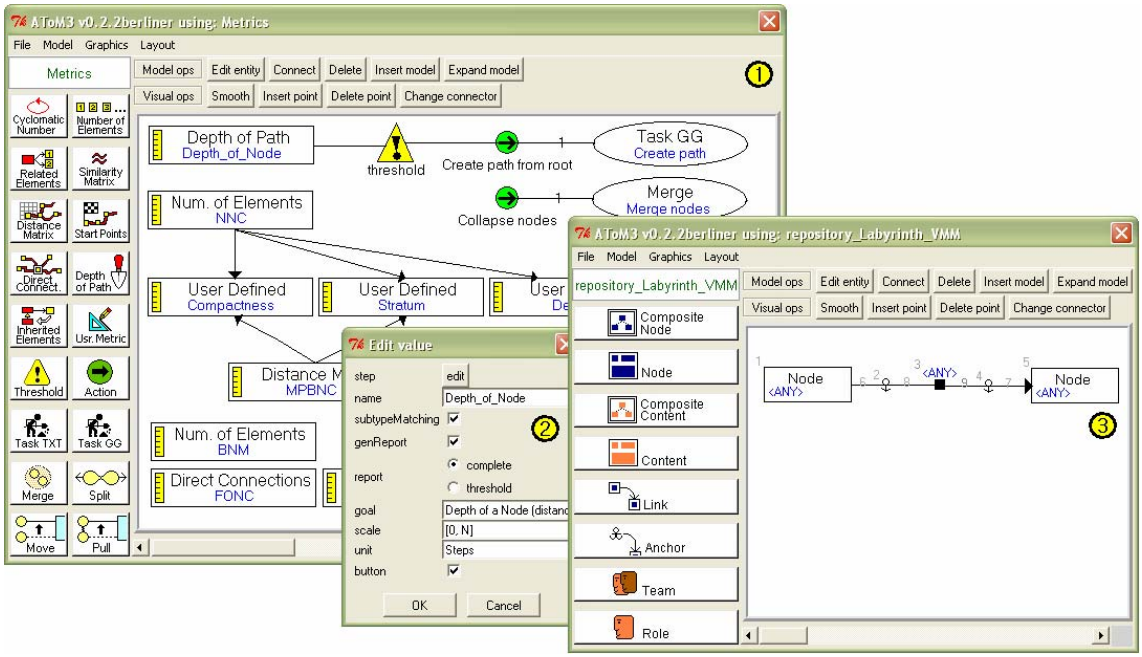

Figure 15: Customisation of "Measurement & Action" Tool for the Labyrinth Environment

Note that metric `Depth_of_Node` defines a threshold value 0 for those nodes that are not root (yellow triangle). Action `Create path from root` (green circle) is executed for those nodes that make the threshold condition true. The action is made of the task shown in Figure 12, which creates a 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 equal to 0), a link is created from the root to the node.

Figure 16 shows the environment automatically generated from the previous definition. In the repository interface (window to the right), a button is generated for each metric and action (if they had checked its attribute *button*, as done for metric `Depth_of_Node`, see Figure 15). Calculating a metric or performing an action just implies clicking on the corresponding button.

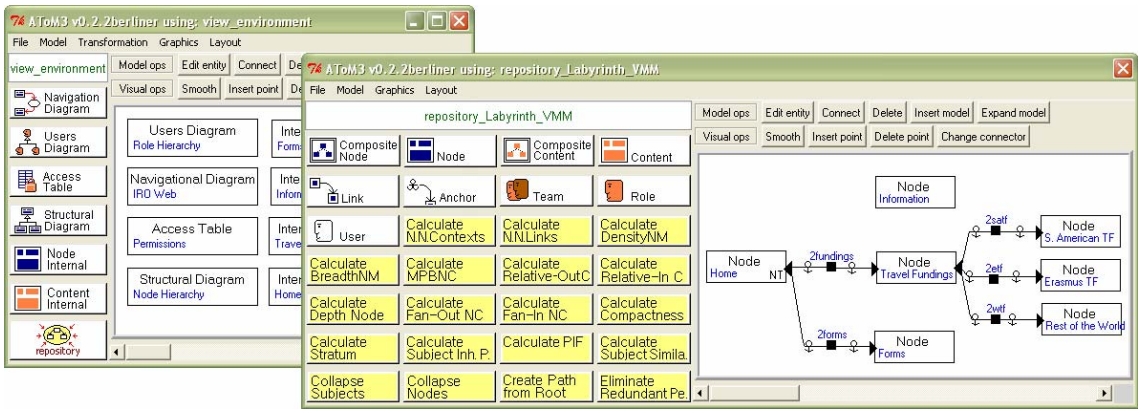

Figure 16: Generated Environment, Enriched with Measurements and Actions

Figure 17 shows to the left the generated report as result of the execution of metric `Depth_of_Node` in the (navigation) model shown in Figure 16. In the report we can see, for example, that node `Information` is not reachable from the root node `Home`, as it has a depth of 0, and that nodes `Travel Fundings` and `Forms` have a depth equal to 1, as a step is necessary to reach them from the root node. This metric has an associated action that is fired when the metric reaches a value of 0. Thus, it is executed for node `Information`. The resulting model is shown in the same figure to the right, where a link has been created from the root node `Home` to node `Information`. Note that the action is not executed for node `Home` because, although it has also depth 0, the threshold is fulfilled only for nodes that are not root.

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 : Erasmus TF Type : lb_Node_repository_Labyrinth	2
Identifier : S. American TF Type : lb_Node_repository_Labyrinth	2
Identifier : Rest of the World Type : lb_Node_repository_Labyrinth	2

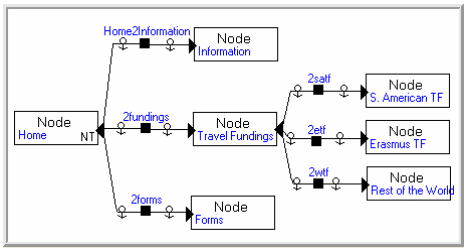

Figure 17: Generated Report and Model Resulting from Action Execution

Using SLAMMER in a MDSD Process

In MDSD processes, models no longer passive entities used for documentation, but they play an active role, typically being used for analysis and code generation (in addition to documentation itself). Thus, models have to be formally defined, and a common trend in software engineering is the use of meta-models to check the conformity of models. The modelling languages used in MDSD can be either general purpose, such as UML, or domain-specific (Pohjonen & Tolvanen, 2002), such as Labyrinth. In the case of general purpose modelling languages, customisations and profiles are a common practice. In MDSD processes, and more in particular in product family engineering (Stahl & Völter, 2006), DSVLs are frequently used for the customisation of the variability of system families. In this case, developers are faced with the problem of generating modelling environments for the DSVLs. It is towards this scenario where a high automation is needed, together with customised tools, where our approach for the easy integration of measurement and redesign tools is directed.

One of the most successful scenarios for MDSD is product line engineering. Two processes are present in product line engineering (Greenfield et al., 2004): the product line development and the specific product development. The first process aims at analysing, designing and implementing reusable assets than can be used in the latter process so as to obtain the final product. In the specific product development, an application is generated by using a product configurator that is responsible of generating code and assembling the existing reusable components. In the most general case, the configurator is a DSVL plus a code generator. Note that this process is not very different from other MDSD processes (Stahl & Völter, 2006) in which a reference architecture has to be defined (i.e. a fixed part of the applications to be generated), together with a code generator, and a DSVL or some other means to express the characteristics of the application to be generated (Czarnecki & Eisenecker, 2000).

Figure 18 shows a simplified scheme (e.g. we have not represented iterations) of a product line engineering process, showing how our approach can be integrated. This can be considered as an additional twist (with the addition of the generative techniques) of the classical process of developing for reuse/with reuse (Karlsson, 1995). To the right, the figure shows the product line development process where the framework and predefined components (i.e. the common part of the product family), the DSVL for configuration and the code generator are built. For simplicity, we don't explicitly show the usual process of first building one or more applications of the family, and then generalizing and exploiting that knowledge in the framework, components and generator. In addition, we propose building a SLAMMER model capturing additional domain knowledge. This includes known good modelling practices with the DSVL, which can be expressed as measures (with associated thresholds) and common redesigns and model refactorings. We have separated two roles in this process: one to design the DSVL and the other one as quality expert to design measures and redesigns. Note that the same person or group of people can assume both roles.

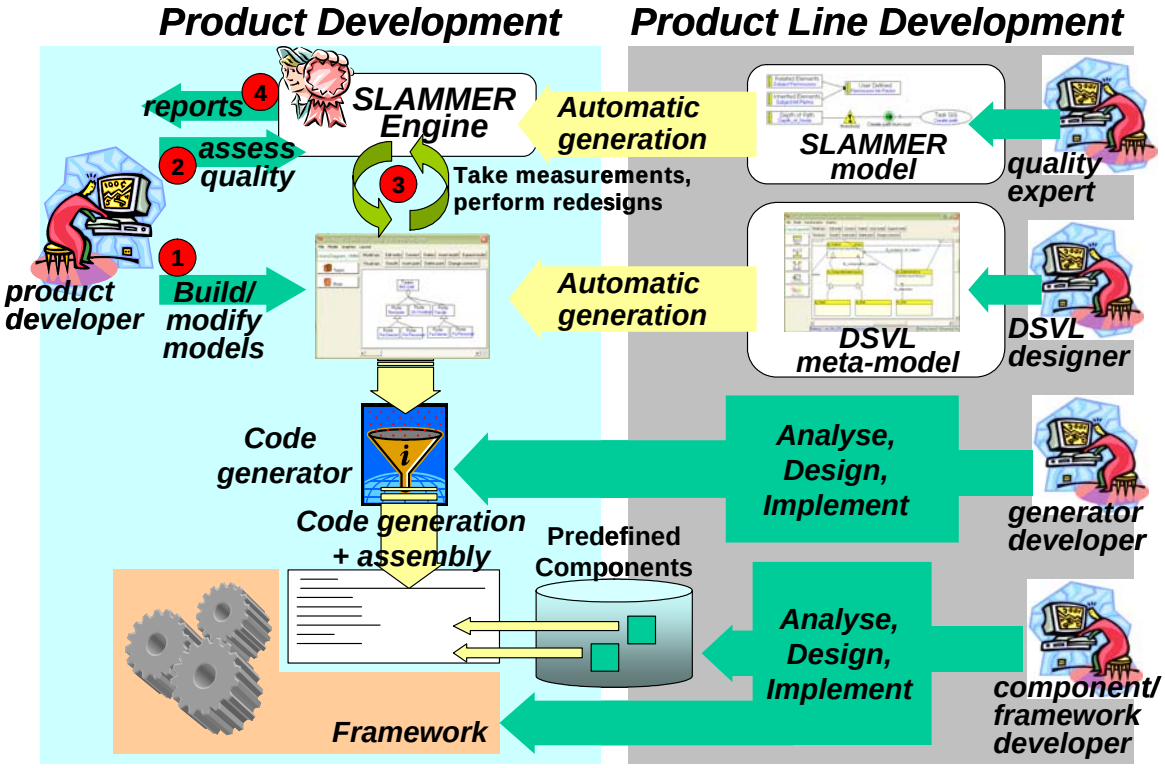

Figure 18: Integrating Quality Assessment in a MDSD Process

The left of the figure shows the process of using the artefacts generated by the development process to the right for developing specific products. In this way, the product developer can use

the DSVL in order to obtain the final application. Note that sometimes, the generated code should be completed by manually written code, but we do not show this activity in the process for simplicity. Thus, our approach introduces quality assessment at the level of product configuration, as the product developer can use the measurement tool provided by the measurement expert in order to check whether the model conforms to the quality standards or know good practices. In addition, he may have available redesigns implementing common or known structural changes to be applied on the models.

In summary, SLAMMER helps in quality assessment in two ways. First, SLAMMER models capture additional domain specific knowledge in terms of measures and redesigns. This knowledge is not only used for documentation, but in order to produce a real tool. Second, the generated tool allows developers to take advantage of the knowledge provided by experts in order to assess the quality of their models. The proposed framework is also model-driven, so code is automatically generated for the final user (vertically in the left-part of the figure) as well as for the developers working in the product development process (horizontally in the figure). That is, the DSVL and the measurement tools are generated from a meta-model and from a SLAMMER model. We believe this is the right approach, as one needs high levels of automation in order to be able to support short iterations, so common in this kind of developments.

Future Trends

The presented framework can be extended by including additional metrics and action templates. It can also be interesting to study how to support other kinds of metrics, for example subjective and dynamic ones. The latter can be suitable in case of having executable models, with a precise operational semantics, for example defined through graph transformation rules. In addition, we are starting the study of mechanisms to support richer customisable template tasks. Providing further analysis tools (e.g. statistical) for studying the results, as well as more powerful visualization facilities for the results is also up to future work.

As stated in the introduction, the evolution of this 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, MDSD needs more functional tools, integrating for example quality control aspects. Some tools (e.g. OpenArchitectureWare, which however does not provide support for DSVLs) are moving towards this direction by integrating a number of additional tools helping in common MDSD 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.

Conclusions

In this chapter we have presented SLAMMER, a DSVL for the specification of measures and redesigns for other DSVLs, and its integration in a MDSD process. The work improves related approaches by decoupling the metrics meta-model and the language concepts, making the predefined metrics totally independent of the domain, and facilitating their integration with any DSVL. Our use of patterns allows a high level of abstraction and reusability, and makes easier the customisation of metrics in a graphical and declarative way. In addition, the SLAMMER meta-model includes entities modelling actions and its relation to metrics, making it more complete for software remodelling.

The framework has been implemented in the AToM^3 meta-modelling tool. In this way, when a modelling environment is generated for a DSVL, AToM^3 makes available the defined measures and redesigns to the final user. To the best of our knowledge, this feature is not available in any

other meta-CASE tool. We have shown the usefulness of this approach by defining a set of metrics and redesigns for Labyrinth, a DSVL in the web domain. However, the approach is general enough to be used with other DSVLs or even general-purpose languages such as UML, by capturing in a SLAMMER model the appropriate measures and redesigns for the notation.

We believe this is a valuable approach especially in MDSD processes, as it simplifies the customisation of metrics and definition of redesigns for DSVLs. Moreover, the implementation supports a model-driven approach for the generation of measurement and redesign tools from the SLAMMER models, allowing fast iterations and easy changes in the SLAMMER models.

References

Abrahao, S., Condori-Fernández, N., Olsina, L., & Pastor, O. (2003). *Defining and validating metrics for navigational models*. In Proceedings of 9th International Software Metrics Symposium, pp.: 200-210.

ADM: Architecture-Driven Modernization home page: http://adm.omg.org

Basili, V. R., Caldiera, G., & Rombach, H. D. (1994). *Goal Question Metric Paradigm*. Encyclopaedia of Software Engineering, pp.: 528-532. John Wiley&Sons.

Botafogo, R. A., Rivlin, E., & Shneiderman, B. (1992). *Structural analysis of hypertexts: identifying hierarchies and useful metrics*. ACM Transactions on Information Systems, Vol. 10(2). pp.: 142-180.

Czarnecki, K., & Eisenecker, E. (2000). Generative programming. Addison-Wesley Professional.

Díaz, P., Aedo, I., & Panetsos, F. (2001). *Modeling the dynamic behavior of hypermedia applications*. IEEE Transactions on Software Engineering, 27 (6), pp.: 550-572.

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

DSLTools from Microsoft, 2007: http://msdn.microsoft.com/vstudio/DSLTools/

Ehrig, H., Ehrig, K., Prange, U., & Taentzer, G. (2006). *Fundamentals of algebraic graph transformation*. Monographs in Theoretical Computer Science. Springer.

Favre, J.-M. (2004). *Towards a basic theory to model driven engineering*. Workshop on Software Model Engineering, WISME 2004, joint event with UML'2004, Lisbon.

Fenton, N. E. (1996). *Software metrics: A rigorous and practical approach (2nd edition)*. International Thomson Computer Press.

Fowler, M. (1999). *Refactoring: Improving the design of existing code*". Addison Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns, elements of reusable object-oriented software*. Professional Computing Series. Addison-Wesley.

García, F., Bertoa, M. F., Calero, C., Vallecillo, A., Ruiz, F., Piattini, M., & Genero, M. (2006). *Towards a consistent terminology for software measurement*. Information and Software Technology 48, pp.: 631-644. Elsevier.

GMF, 2007: The Eclipse Graphical Modeling Framework home page: http://www.eclipse.org/gmf

Gray, J., Rossi, M., & Tolvanen, J.-P. (2004). Special issue on Domain-Specific Modeling with Visual Languages of the Journal of Visual Languages & Computing, Vol. 15 (3-4). Elsevier.

Greenfield, J., Short, K., Cook, S., Kent, S., & Crupi, J. (2004). *Software factories: assembling applications with patterns, models, frameworks, and tools*. Wiley.

Guerra, E., Díaz, P., & de Lara, J. (2006). *Visual specification of metrics for domain specific visual languages*. In Proceedings of Graph-Transformation Visual Modelling Techniques.

Guerra, E., & de Lara, J. (2006). *Model View Management with Triple Graph Transformation Systems*. Proc. ICGT'2006. Lecture Notes in Computer Science, Vol. 4178, pp.: 351-366. Springer.

Guerra, E., & de Lara, J. (2007). *Meta-modelling and graph transformation for the definition of multi-view visual languages*. Chapter of the book "Visual Languages for Interactive Computing: Definitions and Formalization", Idea Group Publishers, edited by Fernando Ferri.

Guerra, E., Sanz, D., Díaz, P., & Aedo, I. (2007). *A transformation-driven approach to the verification of security policies web designs*. In Procedings of the 7th International Conference on Web Engineering. L. Baresi, P. Fraternali, and G. J. Houben, Eds. Lecture Notes in Computer Science, Vol. 4607. Springer. pp.: 269-284.

ISO/IEC 9126 (1991). Software Engineering – Product Quality.

ISO/IEC 15939 (2002). Software Engineering – Software Measurement Process.

Karlsson, E-A. (1995). *Software Reuse: A Holistic Approach*. Wiley.

Kent, S. (2002). *Model Driven Engineering*. In Proceedings of the 3rd International Conference on Integrated Formal Methods. M. J. Butler, L. Petre, and K. Sere, Eds. Lecture Notes in Computer Science, Vol. 2335. Springer-Verlag. pp.: 286-298.

Lämmel, R. (2002). *Towards generic refactoring*. In Proceedings of the 2002 ACM SIGPLAN Workshop on Rule-Based Programming. ACM Press. pp.: 15-28.

Lanza, M., & Ducasse, S. (2002). *Beyond language independent object-oriented metrics: Model independent metrics*. In Proceedings of QAOOSE'02, pp.: 77-84.

de Lara, J., & Vangheluwe, H. (2002). *AToM3: A tool for multi-formalism modelling and meta-modelling*. In Proceedings of ETAPS/FASE'02. Lecture Notes in Computer Science, Vol. 2306, pp.: 174 - 188. Springer-Verlag. See the AToM3 home page: http://atom3.cs.mcgill.ca, and http://astreo.ii.uam.es/~jlara/doctorado.2006/_ATOM3_deploy.zip for the version described in this chapter.

Lédczi, A., Bakay, A., Marói, M., Vögyesi, P., Nordstrom, G., Sprinkle, J., & Karsai, G. (2001). *Composing domain-specific design environments*. IEEE Computer, pp.: 44-51.

Martín, M. A. & Olsina, L. (2003). *Towards an ontology for software metrics and indicators as the foundation for a cataloging Web system*. In Proceedings of LA-WEB. IEEE Computer Society.

Mens, T. & Lanza, M. (2002). *A Graph-Based Metamodel for Object-Oriented Software Metrics*. Electronic Notes in Theoretical Computer Science, Vol. 72(2)

Mens, T. (2006). *On the use of graph transformations for model refactoring*. In Proceedings of Generative and Transformational Techniques in Software Engineering, pp.: 219-257

Misic, V. B. & Moser, S. (1997). *From Formal Metamodels to Metrics: An Object-Oriented Approach*. In Proceedings of 24th International Conference on Technology of Object-Oriented Languages and Systems, pp.: 330-339.

Munro, M., J. (2005). *Product metrics for automatic identification of "bad smell" design problems in Java source-code*. In Proceedings of 11th International Software Metrics Symposium, IEEE Computer Society.

Pohjonen, R., & Tolvanen, J-P. (2002). *Automated production of family members: Lessons learned*. In Proceedings of International Workshop on Product Line Engineering The Early Steps: Planning, Modeling, and Managing, pp.: 49-57.

Roberts, D., Brant, J., & Johnson, R. (1997). *A refactoring tool for Smalltalk*. Theory and Practice of Object Systems, Vol. 3, pp.: 253-263.

Schürr, A. (1994). *Specification of graph translators with Triple Graph Grammars*. In Lecture Notes in Computer Science, Vol. 903, pp.: 151-163. Springer.

SDMetric home page: http://www.sdmetrics.com

Stahl, T., & Völter, M. (2006). *Model-driven software development*. Wiley.

Simon, F., Löffler, S., & Lewerentz, C. (1999). *Distance based cohesion measuring*. In Proceedings of 2nd European Software Measurement Conference, pp.: 69-83.

SPQR/20. (1995). *User Manual*. Software Productivity Research Inc.

Together Technologies home page: http://www.borland.com/us/products/together

Tourwé, T., & Mens, T. (2003). *Identifying refactoring opportunities using logic meta programming*. In Proceedings of 7th European Conference on Software Maintenance and Reengineering, pp.: 91-100.

Tsalidis, C., Christodoulakis, D., & Maritsas, D. (1992). ATHENA: a software measurement and metrics environment. *Journal of Software Maintenance* 4, 2. pp.: 61-81.

UML 2.0 specification at the OMG home page (2006). http://www.omg.org/UML

Warmer, J., & Kleppe, A. (2003). *The object constraint language: Getting your models ready for MDA*, 2nd Edition. Pearson Education. Boston, MA.

Whitmire, S. A. (1997). *Object oriented design measurement*. John Wiley & Sons, Inc.

Additional Reading

Graph Transformation, applications to Refactoring

Rozenberg, G. (ed). (1997). Handbook of Graph Grammars and Computing by Graph Transformations. Volume 1: Foundations. World Scientific.

This book presents the foundations of all the basic approaches to graph transformation.

Ehrig, H., Engels, G., Kreowski, H.-J., U., & Rozenberg, G. (ed). (1999). Handbook of Graph Grammars and Computing by Graph Transformations. Volume 2: Applications, Languages and Tools. World Scientific.

It includes applications of graph transformation to different domains, such as functional languages, visual and object-oriented languages, software engineering or mechanical engineering.

Ehrig, H., Kreowski, H.-J., Montanari, U., & Rozenberg, G. (ed). (1999). Handbook of Graph Grammars and Computing by Graph Transformations. Volume 3: Concurrency, Parallelism and Distribution. World Scientific.

The third book of the series presents the main results on concurrency, parallelism and distribution of graph grammars. An interesting field of application is the coordination of concurrent of systems.

Mens, T., Demeyer, S., & Janssens, D. (2002). *Formalizing behaviour preserving program transformations*, In Proceedings of International Conference on Graph Transformation, Lecture Notes in Computer Science, Vol. 2505, pp.: 286-301, Springer.

This paper introduces a graph representation of those aspects preserved by a code refactoring, and uses graph rewriting rules in order to formalize the refactoring transformations.

Mens, T., Taentzer, G., & Runge, O. (2007). *Analysing refactoring dependencies using graph transformation.* Software and Systems Modeling Journal, Springer.

In this paper, refactorings are formalized by means of graph transformation rules, so that implicit dependencies between refactorings can be studied by using critical pair analysis. The obtained results can help developers to choose which refactoring is more appropriate in a given context.

Additional Meta-Modelling and MDSD Tools

AndroMDA web page at: http://www.andromda.org/

GEMS (Generic Eclipse Modeling System) web page at: http://sourceforge.net/projects/gems

GME web page at: http://www.isis.vanderbilt.edu/projects/gme/

GMT web page at: http://www.eclipse.org/gmt/

MetaEdit+ web page at: http://www.metacase.com/

OpenArchitectureWare web page at: http://www.openarchitectureware.org/

OpenMDX web page at: http://www.openmdx.org/index.html

OptimalJ web page at: http://www.compuware.com/products/optimalj/default.htm

TIGER Project web page: http://tfs.cs.tu-berlin.de/~tigerprj/

UMT web page at: http://umt-qvt.sourceforge.net/

Model-Driven Software Development

Frankel, D. (2003). *Model driven architecture – Applying MDA to enterprise computing*. Wiley.

The Model Driven Architecture (MDA) is the OMG's proposal for Model Driven Development. This book explains this methodology and demonstrates how it can work with different technologies.

Kleppe, A., Warmer, J., & Bast, W. (2003). *MDA explained. The model driven architecture: Practice and promise*. Addison Wesley.

This is a useful second reference for researchers interested in MDA.

Proceedings of the Model Driven Engineering Languages and Systems (MoDELS) series of conferences: http://www.umlconference.org/, edited by Springer Lecture Notes.

Software Measurement and Refactoring

ISO/IEC 25000:2005 Software Engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Guide to SQuaRE. Available at the web page of ISO: http://www.iso.org

Set of standards, including those for software measurement.

Kerievsky, J. (2004). *Refactoring to patterns*. Addison-Wesley.

The book is about improving system designs through the execution of sequences of low-level design transformations (refactorings) towards well-known design patterns. It provides useful examples.

Lindvall, M., Donzelli, P., Asgari, S. & Basili V. (2005). *Towards Reusable Measurement Patterns*. Proceedings of the 11th IEEE International Software Metrics Symposium, pp.: 21-28.

The paper identifies a catalogue of measurement patterns that can be reused in different software measurement programs. The objective is to reduce the time and cost to develop new measurement tools, without starting their implementation from scratch.

Mens, T., & Tourwé, T. (2004). *A survey of software refactoring*. IEEE Transactions on Software Engineering, Volume 30, Number 2, pp.: 126-139.

This paper provides an overview of existing research in the field of software refactoring: supported activities and techniques, target artefacts, tool support, and integration on the software development process.

Pretschner, A., & Prenninger, W. (2007). *Computing refactorings of state machines*. Software and Systems Modeling Journal, Springer.

In this paper, refactorings are formalized as logical predicates and applied to the computation of semantically equivalent models.

Visual Languages

Luoma, J., Kelly, S., & Tolvanen, J.-P. (2004). *Defining domain-specific modeling languages: Collected experience*. Object-Oriented Programming Systems, Languages and Applications (OOPSLA) Workshop on Domain Specific Languages.

This paper explores several approaches to the identification and creation of modelling constructs when defining domain specific languages.

Marriot, K., & Meyer, B. (1998). *Visual language theory*. Springer-Verlag.

This book provides a broad survey concerning the definition, specification, structural analysis and theoretical foundations of visual languages. It is oriented to researchers intrested in formal language theory, HCI, artificial intelligence and computational linguistics.