Proceedings of the Second Workshop on

Graphical Modeling Language Development (GMLD 2013)

In conjunction with
European Conference on Modelling Foundations and Applications (ECMFA 2013)

Montpellier, France, July 2, 2013
Website: http://www.dsmforum.org/events/GMLD13/

ACM International Conference Proceedings Series

ACM Press

Editors:
Heiko Kern, University of Leipzig
Juha-Pekka Tolvanen, MetaCase
Paolo Bottoni, University of Roma

ISBN: 978-1-4503-2044-3

The Association for Computing Machinery
2 Penn Plaza, Suite 701
New York New York 10121-0701

ACM ISBN: 978-1-4503-2044-3

TABLE OF CONTENTS

List of Authors

Program Committee

Additional reviewers

2nd Workshop on Graphical Modeling Language Development

Heiko Kern
University of Leipzig
Augustusplatz 10
D-04109 Leipzig, Germany
+49 341 97 32327
kern@informatik.uni-leipzig.de

Juha-Pekka Tolvanen
MetaCase
Ylistönmäentie 31
FI-40500 Jyväskylä, Finland
+358 14 641 000 ext 23
jpt@metacase.com

Paolo Bottoni
Sapienza University of Rome
Viale Regina Elena 295
I-00161, Roma, Italy
+39 06 .4925 5166
bottoni@di.uniroma1.it

ABSTRACT
This paper describes the 2nd Workshop on Graphical Modeling Language Development, held at ECMFA 2013.

Categories and Subject Descriptors
D.1.7 [**Programming Techniques**]: Visual Programming D.2.6 [**Software Engineering**]: Programming Environments, Graphical environments

General Terms
Languages

Keywords
Modeling Languages, Diagram, Visual Languages

1. INTRODUCTION
Modeling is a fundamental concept in software engineering. Generally models represent a system in an abstract way, improve the understanding of a system and facilitate the communication between different stakeholders. Beyond that, in modern development approaches (e.g. Model-Driven Software Development or Domain-Specific Modeling), models are increasingly used for automating software development tasks such as code generation, model-based testing, simulation and analysis. To express models in a formal way, modeling languages are used. There are a variety of modeling languages and language definition approaches. We can differentiate between general purpose languages and domain-specific languages created for a narrow application area. Regarding the concrete syntax of a modeling language, we can differentiate between graphical and textual languages or a combination of both. A further aspect is the language definition approach, for instance, there are grammar-based or metamodel-based defined languages.

GMLD'13, Jul 01 - July 01 2013, Montpellier, France
ACM 978-1-4503-2044-3/13/07.
http://dx.doi.org/10.1145/2489820.2489821

2. WORKSHOP THEME
The workshop on Graphical Modeling Language Development aims to cover all the phases of language development, including definition, testing, evaluation, and maintenance of modeling languages[1]. Particular attention is given to the principles of modeling language development, especially graphical modeling languages for domain-specific needs. It also includes papers that discuss challenges and new trends. The workshop does not focus on tools, but recognizes the need for metamodel-based tools, which significantly ease the production of modeling environments. These tools also enable experimentation with the language as it is built, and remove the burden of tool creation and maintenance from the language creator.

In response to the call for papers, 8 submissions were received. Submitted papers were formally peer-reviewed by three referees, and 6 papers were finally accepted for presentation at the workshop and publication at the proceedings

The workshop program is composed of two parts: paper presentations and group work. Selected papers focus on language design, metamodeling as well as extending tooling to support modeling work. Group work sessions aim at discussing in more detail the topics found most relevant during the paper presentations. Results of the group work will be presented at the end of the workshop.

3. PROGRAM COMMITTEE
We would like to thank the ECMFA 2013 organization for giving us the opportunity to organize this workshop. Thanks to those that submitted papers, and particularly to the contributing authors. Our gratitude also goes to the members of the GMLD 2013 Program Committee for their reviews and help in choosing and improving the selected papers.

Program committee:

Matthias Biehl, KTH Royal Institute of Technology

Jeff Gray, University of Alabama

Esther Guerra, Universidad Autonoma de Madrid

Kenji Hisazumi, Kyushu University

Emilio Insfran, Universitat Politècnica de València

Teemu Kanstren, VTT

[1] http://www.dsmforum.org/events/GMLD13/

Steven Kelly, MetaCase

Christian Kreiner, University of Graz

Stefan Kühne, University of Leipzig

Juan de Lara, Universidad Autonoma de Madrid

Ivan Lukovic, University of Novi Sad

Sietse Overbeek, University of Duisburg-Essen

Pedro Sánchez Palma, Technical University of Cartagena

Andreas Prinz, University of Agder

Mark-Oliver Reiser, Technical University of Berlin

Jonathan Sprinkle, University of Arizona

Stefan Strecker, University of Hagen

Markus Völter, Independent

Hans Vangheluwe, Univers. of Antwerp / McGill University

We hope that you will enjoy the workshop and find the information within the proceedings valuable toward your understanding of the current state-of-the-art in developing graphical modeling languages.

A Component-Based Approach for Specifying DSML's Concrete Syntax

Amine El Kouhen
CEA LIST, Laboratory of Model Driven Engineering for Embedded Systems
Gif-sur-Yvette, France
amine.elkouhen@cea.fr

Cédric Dumoulin
University of Lille, LIFL CNRS UMR 8022
Cite scientifique - Bâtiment M3
Villeneuve d'Ascq, France
cedric.dumoulin@lifl.fr

Sébastien Gérard
CEA LIST, Laboratory of Model Driven Engineering for Embedded Systems
Gif-sur-Yvette, France
sebastien.gerard@cea.fr

Pierre Boulet
University of Lille, LIFL CNRS UMR 8022
Cite scientifique - Bâtiment M3
Villeneuve d'Ascq, France
pierre.boulet@lifl.fr

ABSTRACT

Model-Driven Engineering (MDE) encourages the use of graphical modeling tools, which facilitate the development process from modeling to coding. Such tools can be designed using the MDE approach into meta-modeling environments called *metaCASE tools*.

It turned out that current metaCASE tools still require, in most cases, manual programming to build full tool support for the modeling language, especially for users' native methodologies and representational elements and propose limited possibilities in terms of reusability. In this context, we propose *MID*, a set of meta-models supporting the easy specification of modeling editors by means of reusable components and explain how representational meta-modeling is carried out with it.

Categories and Subject Descriptors

D.1.7 [**Programming Techniques**]: Visual Programming ; D.2.6 [**Software Engineering**]: [Graphical environments]
; D.2.2 [**Software Engineering**]: Design Tools and Techniques
; D.2.13 [**Software Engineering**]: Reusable Software

Keywords

MetaCASE tools, Component-based metamodeling, Visual languages, Model-Driven Development (MDD), Reusability, Concrete Syntax

GMLD '13, Montpellier, France

http://dx.doi.org/10.1145/2489820.2489822 ...$15.00.

1. INTRODUCTION

As part of its Model-Driven Architecture (MDA) initiative, the Object Management Group (OMG) [17] - an international consortium representing numerous industrial and academic institutions - has provided a comprehensive series of standardized technology recommendations in support of model-based development of both software and systems in general. These cover core facilities such as meta-modeling, model transformations, and general-purpose and domain-specific modeling languages. A key component in the latter category is UML (the Unified Modeling Language), which has emerged as the most widely used modeling language in both industry and academia. A number of tools supporting UML are available from a variety of sources. These are generally proprietary solutions whose capabilities and market availability are controlled by their respective vendors. Consequently, some industrial enterprises are seeking open-source solutions for their UML tools. To respond to this requirement, a new graphical editor called *Papyrus*, was accepted by the Eclipse Project MDT in August 2008. This graphical editing tool for UML2 is based on Eclipse and uses the Eclipse graphical modeling Framework (GMF).

Papyrus is a tool consisting of several editors, mainly graphical editors but also completed with other editors such as textual-based and tree-based editors. All these editors allow simultaneous viewing of multiple diagrams of a given UML model. However, when such diagrams were specified, we found common problems at different levels. The first common point relates to the redundant elements in all UML diagrams, such as *Comments* or *Constraints* elements that are presents in all Papyrus diagrams. The second common point relates to some specific diagrams as *Package Diagram*, which is composed of a *Class Diagram* subset (Package, import, merge ...). The other common point relates to the graphical variation of several features. For example, a *Class* in the class diagram does not have the same graphical representation as in the composite structure diagram. The same thing for the *Actor* element in the use cases diagram and other diagrams. These elements have the same semantics in

the UML model, sometimes they have the same graphical structure, but they are represented with different shapes. This statement raises a real issue of reuse when specifying diagrams.

To explain these issues, we evaluated the technologies currently used to specify Papyrus diagrams [5]. It turns out that the main reason for these gaps is the lack of reusability in this kind of technology. This results in manual copies in all diagrams, thus increasing the risk of error, problems of consistency, redundancy in the specification and the difficulty of maintenance.

At a high level of abstraction, the study of these tools, and especially GMF, allows us to identify some needs and criteria in terms of reusability, graphical completeness, model consistency and maintainability of diagrams specifications. Compliance with these criteria led us ultimately to produce an alternative meta-tool based on a set of meta-models called *MID* (**M**etamodels for user **I**nterfaces and **D**iagrams), to rapidly design, prototype and evolve graphical editors for a wide range of visual languages. We base MID's design on three overarching requirements: graphical completeness, ease of use and simplicity of (de)composition of diagrams editors for a better reusability. For that, we take advantage from MDE benefits, Component-based modeling and an inheritance mechanism to increase the reuse of editors' components. The main goal of this work is the specification and the generation of UML diagram editors of Papyrus [7], from reusable pre-configured components.

For all of this, we consider it being useful to present the basics of visual languages as well as the study we made on tools and modeling methods in section 2. This section ends with a summary of issues arising from this study and a proposal for criteria to be met by our solution. Then, section 3 describes, our approach to design graphical editors based on our proposal: MID meta-models. In section 4, we validate our approach through examples using our proposed inheritance mechanism to reuse graphical components of a diagram.

2. FOUNDATIONS AND RELATED WORKS

In this section, we present the state of the art of graphical modeling environment. We begin by presenting visual representation foundations to extract concepts that describe a diagram and then studying several existing methods and tools for specification and generation of graphical editors for diagrams. This study has identified several issues, which we expound as evaluation criteria for our approach.

2.1 Visual Languages Basics

Visual representation is one of the oldest forms of knowledge representation and predates conventional written language by almost 25,000 years [19]. Visual representations play significant roles in scientific reasoning [18].

According to [15], elementary components of a visual representation are called visual notations (visual language, diagramming notations or graphical notations) and consist of a set of graphical symbols (**visual vocabulary**), a set of compositional / structural rules (**visual grammar**) and definitions of the meaning of each symbol (**semantics**). The visual vocabulary and visual grammar form together the **concrete** (or **visual**) **syntax**.

In the literature, numerous definitions can be found for the concept of diagram. The widely accepted ones, are that

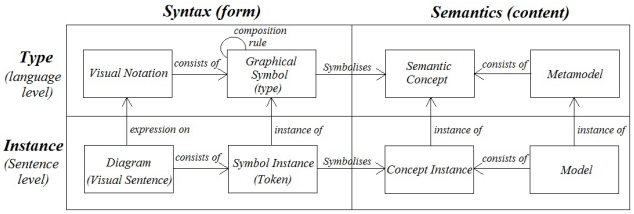


Figure 1: The nature of a visual notation [15]

of Kosslyn [10], Larkin [11] and Tversky [20]: *Diagrams are an effective medium of human thinking and problem solving. Diagrams are thus bi-dimensional, geometric, symbolic and human-oriented representations of information; they are created by humans for humans. They have little or no value for communicating with computers, whose visual processing capabilities are primitive at best* [16].

Graphical symbols are used to symbolize (perceptually represent) semantic constructs, typically defined by a metamodel [8]. The meanings of graphical symbols are defined by mapping them to the constructs they represent. A valid expression in a visual notation is called a **visual sentence** or **diagram**. Diagrams are composed of symbol instances (tokens), arranged according to the rules of the visual grammar [15]. Such distinction between the content (semantics) and the form (syntax: vocabulary and grammar), allows us to separate the different concerns of our proposition. These definitions are illustrated in fig. 1.

The seminal work in the graphical communication field is Jacques Bertin's *Semiology of Graphics* [1]. Bertin identified eight elementary visual variables, which can be used to graphically encode information (Fig. 2). These are categorized into planar variables (the two spatial dimensions *x,y*) and retinal variables (features of the retinal image).

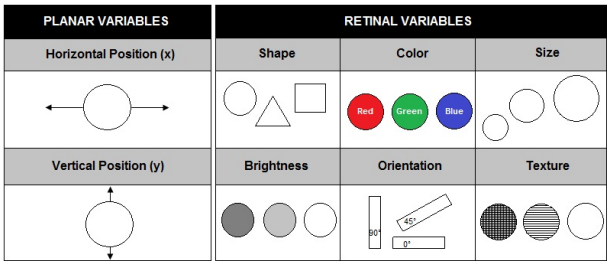


Figure 2: Visual variables [1]

The set of visual variables define a *vocabulary* for graphical communication: a set of atomic building blocks that can be used to construct any graphical representation. Different visual variables are suitable for encoding different types of information. The choice of visual variables has a major impact on cognitive effectiveness as it affects both speed and accuracy of interpretation [3, 13, 22].

A Diagram is defined as a planar graph, which has hybrid languages capabilities [2]. In the graph theory field, a pair G=(V, E) is called **graph**. The elements of V are the **vertices** of G, and those of E the **edges** of G. In literature, vertices are also called nodes or points; edges are called lines or links. These two concepts and others are the main ones in our visual grammar definition.

2.2 Evaluation Criteria

Many frameworks, meta-tool environments and toolkits have been created to support the development of visual language environments. We conducted an evaluation of the technologies currently used to specify the diagrams in Papyrus and other techniques and tools to do so in the state of the art. It turns out that the main cause of such limitations is the lack of reusability mechanism in this kind of technology. This results in manual copies in all diagrams, thus increasing the risk of error, problems of consistency, redundancy in the specification and the difficulty of maintenance. To understand these issues, we offer an overview of our tools evaluation. The diagrams specification methods have been widely discussed in [5, 2]. We summarize them in the following categories:

1. Code-based specification. As GEF, UMLet and Graphiti. This kind of tools construct graphical representations from a code description.

2. Proprietary languages-based specification. This category of tools uses meta-description languages to specify graphical editors, as MetaEdit+, which uses the GOPPRR metamodeling language [9] and GME [12], which uses a meta-description to specify the abstract and concrete syntaxes for a specific domain.

3. Specification based on graph grammar. This kind of tools is based on the graph grammar definition and allows specifying diagrams from this grammar (e.g. Visual DiaGen [14], AToM^3 [4]).

4. Other tools allow editor specifications; they are exclusively graphical drawers i.e. they care only about the graphics without giving meaning to them: semantics mapping-less (e.g, Microsoft Visio or Dia).

5. There are other methods and tools, which have recently adopted MDE approaches; we distinguish two major categories in this specification method :

   - Tools based on UML profiles as Papyrus [7], IBM RSA and MagicDraw, which propose to create visual representations for profiles' metaclasses (UML Stereotypes mechanism).
   - Tools based on DSML's. Domain-Specific Modeling Languages are more specific to domain requirements, giving users specific concepts to their occupation and their expertise. GMF tooling, Spray, TopCased Meta and Obeo Designer are examples of tools supporting this kind of specification. They describe (via models) the concrete syntax and associate it to a specific domain metamodel. This kind of technology, including GMF is used in Papyrus for several reasons (ease to manage compared to a model code, open-source technology, ease of integration with the Eclipse ecosystem ...). However, these tools still require programmatic interventions for adaptation. Many other shortcomings are detected, mainly in terms of reuse and tools rigidity [5].

This article focuses on the criterion of reusability that we consider very useful in the context of diagrams specification. Among the forms of reuse, we can cite the separation of concerns, inheritance and overriding.

Most of diagrams specification methods mix concerns. The most common form of this mixture is that of form and content (visual representations and semantics). For example, in the case of diagram specification using GME or MetaEdit+, these tools allow creating the specific concepts and their associated representations in the same repository. This weakens the required loosely coupling relationship between the semantics and the graphical aspects, which limitate the reuse of the concrete syntax. The same problem is observed with Obeo Designer, which allows graphical/semantics association in the same model.

Other form of mixing is between the visual vocabulary and visual grammar definition. Most of the tools offering the separation of the graphical part from the semantic one, as GMF tooling, TopCased-Meta and even standards like Diagram Definition (OMG, http://www.omg.org/spec/DD/), fail to separate the two graphical syntax concerns, which are visual vocabulary (shapes, colors, styles ...) and visual grammar (structure and composition of representations).

We found another problem with GMF or Obeo Designer: few opportunities (or lack) of elements specification reuse, which usually results *redundant specifications* at the **model-level** and *redundant classes* at the **code-level**.

The design of Papyrus brings up an important need in terms of diagrams definition reusability. It would be nice to describe a library of concrete syntax, independent from the semantics, to be reused (if necessary) in different diagrams, to factorize common concrete syntax in common definitions and variations in specific definitions based also on the common definitions. This will allow us to define diagrams independently from each other with the possibility of reusing and redefining (overriding) common elements.

3. METAMODELS FOR USER INTERFACES AND DIAGRAMS (MID)

The aim of our work is to design diagram editors and to allow reusing parts of such design. For that, we propose to use a Model-driven approach, to ensure the independence to technology, ease the maintenance and enable better sustainability.

To improve reusability, we propose a component-based approach. This approach aims to take advantage of encapsulation (ease of maintenance and composition) and the benefits of interfacing (interfaces naming mechanism). In addition, our approach allows the reuse through inheritance: a component can inherit from another one and it can also override some of its properties (style, structure, behaviour...).

The first advantage of this approach is the independence from the semantics, which allows assembling graphical elements differently from a diagram to another. Another advantage is the ease of the diagram specification maintainability. By avoiding redundancy and duplication at the generated code, we can facilitate the maintainability of the diagram specification, but also by providing a mechanism that allows us to reflect any change in a definition to all places using it. The component approach is promising in this context. By defining the composition (structure) of a graphical element and the encapsulation of its interfaces, it is easy to maintain and modify it.

Figure 3 shows the linkage of the meta-models involved

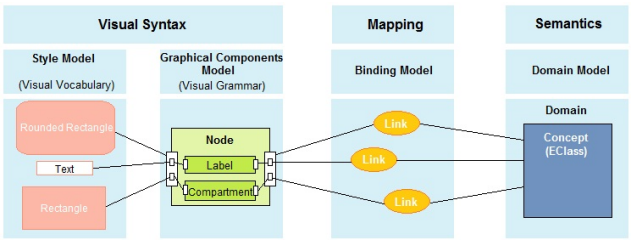


Figure 3: MID: Involved Artifacts relationship

in our proposal. First, we separate the domain content (**semantics**) and the form (**visual syntax** or concrete syntax) of a diagram at a high level of abstraction (language level). The Semantics is out of scope of our paper, it is widely treated in tools and technologies like EMF/Ecore. The form is separated into two parts : the **visual vocabulary** (different variables of shape, color, size ...) and the **visual grammar** that describes composition rules of visual representations. The link between the syntax and the semantics is also specified in a separate "binding" model. Thus, our proposal is made of several meta-models, each one used to describe one concern: a visual grammar meta-model, a visual vocabulary meta-model and a mapping meta-model.

3.1 Component Metamodel

Our goal is to specify and compose the elements of the diagram editor with an approach based on the characteristics of the components. We follow a model-based approach for modeling diagram components to solve two problems : components heterogeneity and their composition techniques.

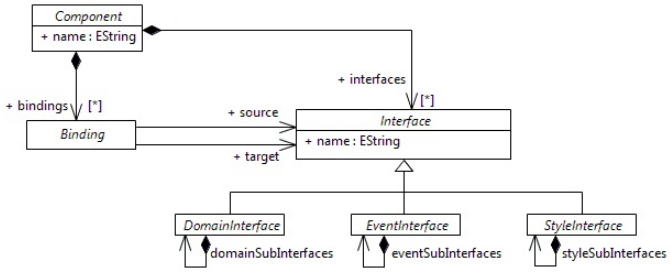


Figure 4: Component Metamodel

The component concept is the main concept of our set of meta-models. A component could have interfaces (in our context there are three types of interfaces: *domain*, *style* and *event interfaces*) and the bindings between such interfaces. Interfaces are used as an attachment point between (sub)components and the other concerns (semantics, behaviours and styles).

3.2 Visual Grammar: Diagram Composition

The visual grammar is used to describe the structure of diagrams' elements. This description is hierarchical: a root element can contain other elements. We propose two main types of elements: *Vertices* to represent complex elements of diagrams and *edges* to represent links between complex elements.

Vertex is node abstraction and shape formalization [6], it consists of main nodes (top nodes), sub-nodes (contained vertices in figure 5) and attached nodes (nodes that can be affixed to other nodes). A label is a vertex that allows access to nodes textual elements via their accessors (getters and setters). This will synchronize the data model with text value represented. A Bordered node is a node that can be affixed to other nodes. Containers (Compartments) are specific nodes that contain diagram elements. A Diagram is itself a container. An Edge is a connection between two diagram elements, this relationship could be specified semantically (in the domain metamodel) or graphically and could be more complex than a simple line (e.g, a data bus between two devices).

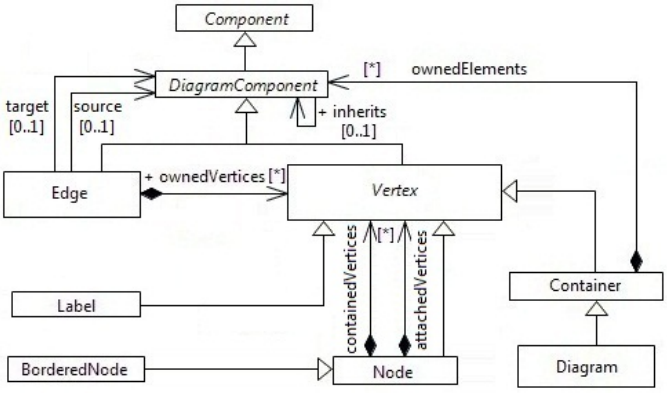


Figure 5: Diagram Elements

The meta-model in figure 5 represents diagrams main concepts. It allows designing all hybrid visual languages [2], which we use in the MDE field like UML, Petri Net, BPMN...

Edges and nodes have both the ability to inherit from each others. When a diagram component inherits from another, it recuperates all its properties (structure, style and behaviour). If the inheriting component has an element with the same name as the inherited component, this is interpreted by an overriding and then we can override the structure, style and behaviour. This feature maximizes components reuse and allows creating other derivatives components. The proposed rules of graphical inheritance are resumed in the algorithm 1.

Algorithm 1 MID: Graphical Inheritance mechanism

$\forall A, B \in DiagramComponent$, $\exists X_A, X_B$ the sets of owned-components of A and B;
if B inherits from **A then**
  $B \leftarrow X_A$;
  $\forall$**a**, **b** sub-components with $\mathbf{a} \in X_A, \mathbf{b} \in X_B$;
  if NameOf(**b**) = NameOf(**a**) **then**
    b override **a** :
    – Case 1: if we need to update **a**
    we redefine its interface in **b**;
    – Case 2: if we need to delete **a**
    we do not specify its interface in **b**;
    – Case 3: if we need to add a new element
    we add its new interface in **b**;
  end if
end if

Visual grammar elements only represent the structure and

should be associated to a visual vocabulary describing its rendering.

3.3 Visual Vocabulary: Visual Variables

Visual vocabulary allows describing the graphical symbols (visual representation) of diagrams' elements. This description is composed of different visual variables; we regroup all of them in the *Style* concept (fig. 6) representing the shape, color, size, layout...

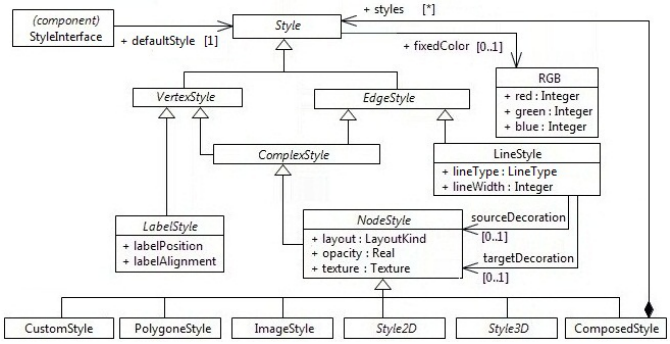


Figure 6: Visual variables(Styles) description

All diagram components are associated via their *style interfaces* to visual vocabularies represented in the meta-model by the concept of **Style**. As other characteristics of diagrams elements, this relationship can be reused and overriden through the proposed mechanism of inheritance.

The *Vertex Styles* are characterized by the layout attribute, which represent the different arrangement rules in the host figure. Vertex styles are decomposed into two main categories: node styles and label styles. Node styles represent shapes (2D and 3D figures and iconic representations like images). We propose ten default shapes in our meta-model, and we let users to create their own shapes with polygons, images and custom styles (code implementation). Label styles specify label alignment, position and label type. The *Edge Styles* could be simple lines (**LineStyle**) or more complex shape (**ComplexStyle**).

3.4 Domain Mapping

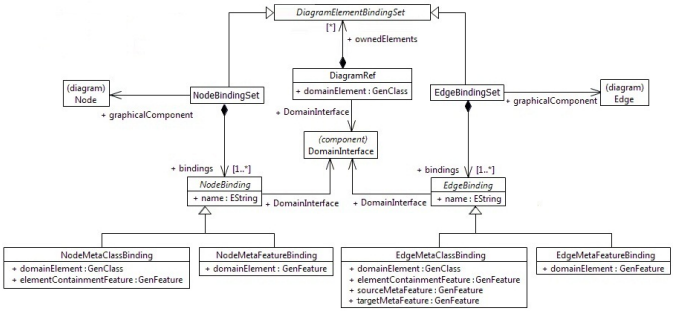


Figure 7: Semantics Mapping

The mapping meta-model (fig. 7) allows binding the different graphical elements (nodes and edges) to their corresponding semantic concepts. A graphical element can be mapped to one or more semantic elements. This is done through the concepts **NodeBindingSet** and **EdgeBindingSet**, each containing a set of associations **NodeBinding** or **EdgeBinding**.

Currently, the description of the mappings serves as entry point to the full description of the diagram. This is represented by the element **DiagramRef**, which contains all associations. This approach allows us to reuse graphical representations in different diagrams with different semantics. For example, in a UML class diagram, it is possible to use the same representation for both concepts *Class* and *Interface*. We use this to increase the reusability of graphical components.

3.5 Representation formalism

For simplicity, we propose a graphical formalism to present our concepts. This formalism allows to see graphically the diagrams specification instead of textual or tree-based form. Thus, we propose a concrete syntax for our metamodels. Note that our tools can also define this formalism with the same concepts (auto-description).

The figure 8 shows an example of a component specification with the graphical view (top right) and its equivalent in tree-based view (top left). Both views allow the result in the bottom of the figure.

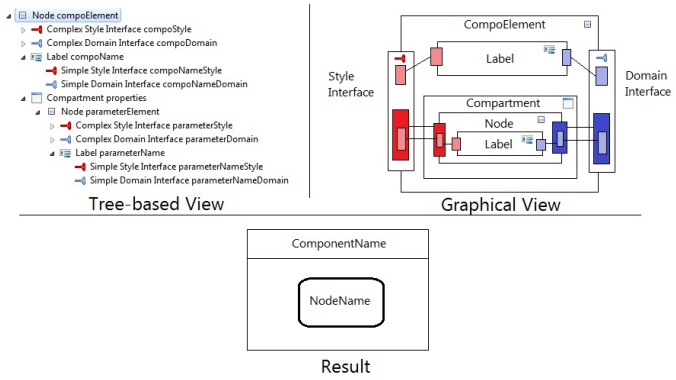


Figure 8: MID Graphical Formalism

4. VALIDATION

To validate our proposal, we have developed a chain of transformations allowing the full generation of designed editors code. Note that MID meta-models are completely independents from technological targets. In the actual implementation, we choose GMF as technological target.

We illustrate the advantages of our approach on an example specifying the UML *Classifier* element to show the reusability through inheritance. Then, we validate our approach through a case study.

4.1 Reuse by Inheritance

We chose as an example the UML concept *Classifier*. This abstract concept is the basic element of several concepts (*Class*, *Interface*, *Component*...). We want to specify the graphical appearance of the item *Classifier* and use inheritance and overriding to specialize this specification.

The basic element of *Classifier* consists graphically of a label followed by a compartment that contains properties. To specify a *Component*, we have simply to inherit from *Clas-*

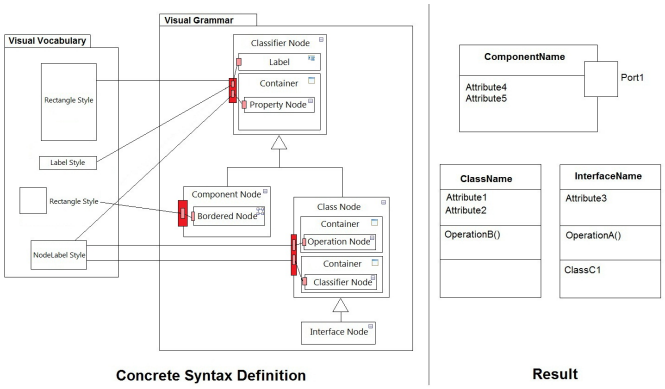


Figure 9: Reuse by Inheritance

sifier and add to its structure, a border node representing the component ports (attached on borders).

To Specify the graphical elements *Class* and *Interface*, we have simply to inherit from *Classifier* and add to its structure two other compartments, the first for operations and the other one for nested classifier. In this example, and to simplify, the Interface inherits from the graphical definition of the Class to show the graphical similarity between the two concepts. The figure 10 shows the generated result.

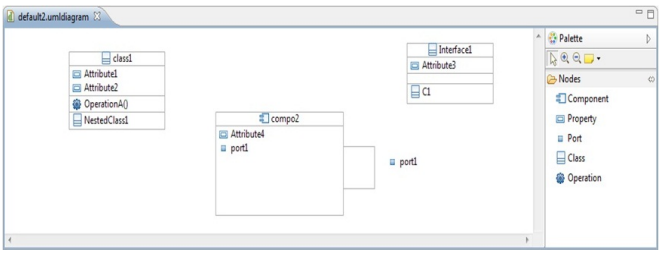


Figure 10: Generated editor for this example

The example below shows the overriding of inherited elements. The component "**Node B**" inherits from the component "**Node A**". Both have sub-components named "Node x", in this case the element *"Node x" of B* overrides the description of *"Node x" of A* (initially represented by an ellipse).

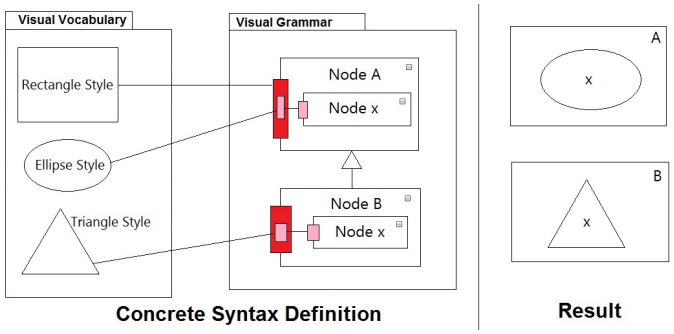


Figure 11: Example of graphical overriding

4.2 Chain of Transformation MID→GMF

To validate quickly our approach we have developed a transformation chain to generate Java code for diagram editors using GMF. The transformation chain allows us to move from a modeling workspace to another. Each workspace is at a level of abstraction and detail higher than the other. Intermediate models are described by meta-models to add technical details and technology needed to code generation. We took care to introduce technological details the latest possible. These details appear in the latest model of the chain (GMFGen Model).

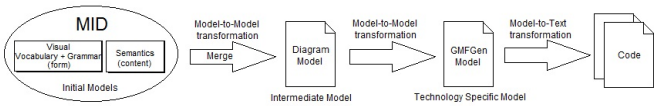


Figure 12: Chain of Transformation MID→GMF

This transformation chain allows us to generate the entire graphical editor. The tool user has only to run the application. The generated editor allows manipulating domain-specific concepts with graphical representations specified at the model level.

Currently, our approach allows reusability at the level of diagrams design. However, the chosen target technology (GMF) does not allow the reuse at the level of the generated code. It remains duplicated in the code-level, but with a single definition at our meta-models.

4.3 Case Study

We choose for the case study, to design the BPMN diagram and the UML state-machine diagram. Because of their graphical resemblance, this choice aims to show the level of components reusability in both editors.

4.3.1 BPMN Workflow Diagram

The Business Process Management Initiative (BPMI) has developed a standard Business Process Modeling Notation (BPMN). BPMN is dedicated to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes [21].

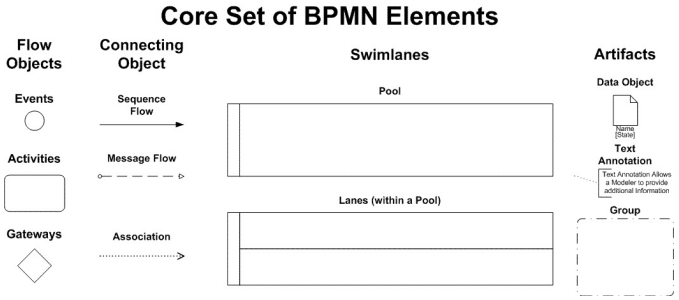


Figure 13: Graphical elements of BPMN

BPMN defines a Business Process Diagram (BPD), which is based on a flowcharting technique tailored for creating graphical models of business process operations. A Business Process Model, then, is a network of graphical objects,

which are activities (i.e., tasks) and the flow controls that define their order of performance [21]. In terms of concrete syntax elements, there are four basic categories of elements: Flow Objects, Connecting Objects, Swimlanes, and Artifacts. The symbols corresponding to them are summarized in Figure 13.

For this example, we have created a simple metamodel that includes a subset of the language concepts. This metamodel is not meant to be a realistic representation of BPMN (this is out of the scope of this study). A complete specification of the BPM notations and semantics can be found in (OMG, http://www.omg.org/spec/BPMN/2.0/).

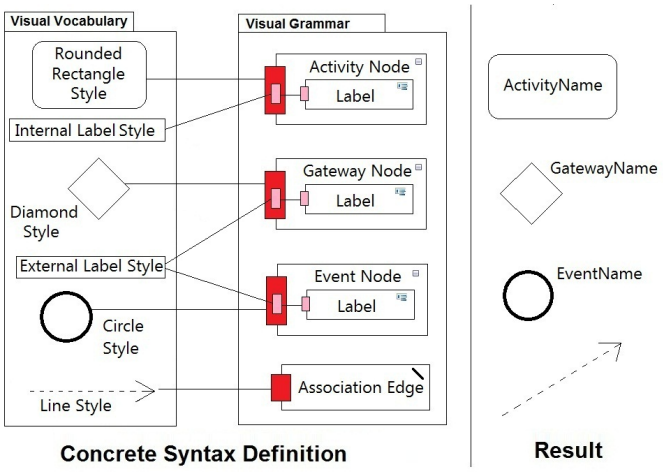


Figure 14: Specification of BPMN Diagram

We define thus BPMN diagram elements by mapping each graphical component to a style via its style interface for defining different visual variables. We find graphical similarities between events (start, intermediary, end) and gateways (OR, AND, XOR). For this reason, we use the inheritance mechanism between these elements by overriding their styles for specific needs.

4.3.2 *UML State-Machine diagram*

State machines can be used to specify behaviour of various model elements. For example, they can be used to model the behaviour of individual entities (e.g., class instances). The state machine formalism described in this sub clause is an object-based variant of Harel statecharts (OMG, http://www .omg.org/spec/UML/).

UML state machine diagrams depict the various states that an object may be in and the transitions between those states. In fact, in other modeling languages, it is common for this type of a diagram to be called a state-transition diagram or even simply a state diagram. A state represents a stage in the behaviour pattern of an object, and like UML activity diagrams it is possible to have initial states and final states. An initial state, also called a creation state, is the one that an object is in when it is first created, whereas a final state is one in which no transitions lead out of. A transition is a progression from one state to another and will be triggered by an event that is either internal or external to the object.

For this example, we have used the UML2 meta-model implementation within Eclipse (also known as the "UML2 Component"). This component has become the *de facto* standard implementation of the UML2 meta-model (note that it is also the basis for the UML2 tool suites provided by IBM).

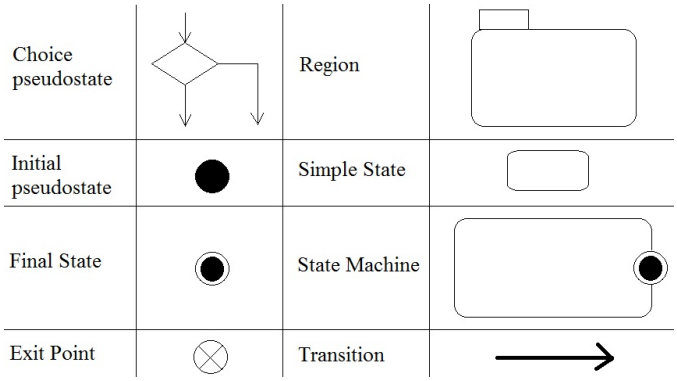


Figure 15: Graphical elements of UML SM

As BPMN Workflow Diagram, we define the UML State Machine Diagram with the same process. Because of the graphical similarity between BPMN and UML State Machine, we reuse most of components already defined in the first example (more than 80% of BPMN components). We defined the specificities of the State Machine diagram, using the inheritance mechanism and the overriding of components. This allowed us to gain a considerable time of development in the second example.

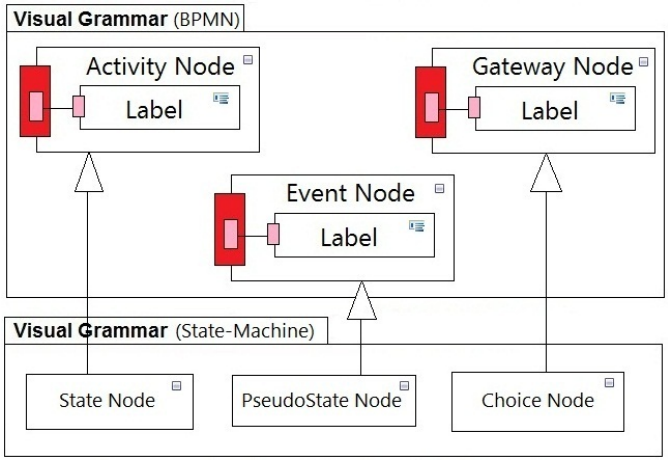


Figure 16: Specification of State-Machine Diagram

The full design and generation of BPMN and UML state machine editors, allow us to perform a first validation of our approach. We complete this validation by an evaluation according to the criteria identified in Section 2.2 of this Paper.

The specification method chosen for our approach is based on models. This approach allowed us to benefit the undeniable advantages of MDE into the editors' development cycle. These benefits are reflected in multiple aspects, such as ease of specification and technology independence, which allow a greater collaboration and flexibility in the metamodeling cycle of editors.

The following screenshot show respectively the BPMN and UML State Machine graphical editors generated from the specifications presented above.

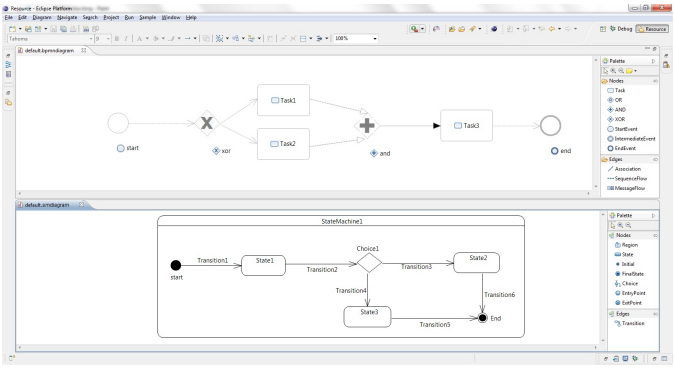


Figure 17: Generated editors: BPMN editor (top side) and State Machine editor (bottom side).

The reusability was since the beginning of our research work, the most important criterion and the most sought. This criterion motivated us to seek methods that allow more reuse of specification models. For this reason, we choose to introduce the concept of component-based metamodeling to specify graphical editors. Component-based approach ensures a better readability and better maintenance of models. It is particularly useful for teamwork and allows industrializing software development. Reusability of a component saves significant productivity gain because it reduces development time, especially as the component is reused more often.

Through the examples presented in Section 4.3, we validated our approach in terms of reusability: This approach allowed us (in State-Machine example) to **reuse more than 80%** of components created in BPMN example, which is not negligible. This approach allowed us to define more easily, editors' specificities with a model-driven approach and without any need to redefine or manually program changes, which increases the level of maintainability of editors generated with our solution.

The graphical completeness is defined by the capability to use fully the shape variable (the use of any kind of shapes : complex, composites, 2D/3D...). Unlike tools we have evaluated in [5], our metamodels have a great capacity to use the full range of these variables; we were inspired by tools based on graph grammar and their approaches to define those variables. We have proposed also representation methods used in the other tools like SVG representations, and other predefined figures : we propose around ten default shapes in our graphical metamodel, and we let the possibility to users to create their own shapes with polygons, images and the custom styles (classes). We solve some problems identified in existing tools and methods on the industry as in the literature. For example, the specification of complex diagrams editors as sequence diagram at a high level of abstraction without the need for manual programmatic intervention.

Unlike existing tools for diagrams specification, we separate the editor's aspects and concerns initially between the semantic and the graphical aspect, then we separate the two graphical aspects, which are the visual vocabulary (visual variables) and the visual grammar, which represents the composition rules of diagrams. Subsequently, it's important to create another part that would make the mapping between the different aspects, in particular between the semantic and the graphic. The separation of concerns is also carried out in the transformation chain, by introducing several intermediate level models and by delaying the introduction of technical details in the latest models of the chain, which allows a better maintainability of the transformation chain in case of a change in metamodels. A strong separation of concerns allows a better reuse and maintenance of models, it decreases development costs in terms of maintenance time in case of changes in these models and should allow designing new applications by assembling existing models. It also allows to reuse a complete diagram description with another domain model.

5. CONCLUSION

In this article, we present an approach based on MDE and components modeling, allowing the easy specification of diagram graphical editors at a high level of abstraction, in order to model, reuse, compose and generate code. In our proposal, we focus on the component concept, to describe and then assemble concepts emerging from visual languages. First, we present the definitions and theoretical foundations of visual representations and the component-based metamodeling approach while positioning our work in relation to different tools and technologies available in the industry and in the literature.

In our approach, we promote the use of reusable components (with composition, encapsulation, inheritance...) and a strong separation of concerns (domain, graphical element, styles). This increases the reusability of the editors and brings the benefits of the MDE paradigm as models verification/checking or the ability to choose target technologies through model transformation techniques.

In MID, we solve some problems identified in existing tools and methods on the industry as in the literature. For example, the specification at a high level of abstraction without the need for manual programmatic intervention, the separation of concerns, the graphical effectiveness and finally editors reusability, which was among the major problematic of our research work. To validate our approach, we have developed a transformation chain targeting the GMF technology (GMFGen), which allows in turn generation of functional editor's code. This allows us to successfully design diagrams by reusing existing components, and to generate their implementation We validated our approach on several diagrams.

Our approach presents many advantages. First, through the reuse of model: the models are theoretically more easily to understand and to manipulate by business users; which corresponds to a goal of the MDE. Secondly, this reuse saves considerable gain of productivity through ease of maintenance of components; it allows better teamwork and helps for the industrialization of software development: it is possible to build libraries of components, and then build the diagram by assembling these components.

Briefly, we can say that our approach opens a new way that shows promises for wider use of modeling tools and automatic generation of applications. Compared to the current development technologies, the promises of this approach are large through the ability to create complex applications by assembling existing simple model/components fragments, and especially the possibility for non-computer specialists, experts in their business domain, to create their own applications from a high-level description using an adapted formalism, easy to understand and manipulate for them.

In the current state of our research, many studies are still required to reach a full generation of modeling tools. First, we need to finalize the description and generation of all graphical editors of Papyrus with our approach. Finally, we need to define other meta-models that allow description of the other parts of such tools (Tree editors, tables/matrices, properties views...) following the same approach of component reuse and inheritance.

6. REFERENCES

[1] J. Bertin. *Semiology of graphics : diagrams, networks, maps*. University of Wisconsin Press, Madison, Wisconsin, 1983.

[2] P. Bottoni and A. Grau. A suite of metamodels as a basis for a classification of visual languages. In *IEEE Symp. on Visual Languages and Human Centric Computing*, pages 83 –90, sept. 2004.

[3] W. S. Cleveland and R. McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79(387):531–554, 1984.

[4] J. De Lara and H. Vangheluwe. Using atom3 as a meta-case tool. In *4th International Conference on Enterprise Information Systems (ICEIS), April 2002, Ciudad Real, Spain*, pages 642–649, 2002.

[5] A. El-Kouhen, C. Dumoulin, S. Gï¿½rard, and P. Boulet. Evaluation of modeling tools adaptation. Technical report, CNRS, 2011. `http://hal.archives-ouvertes.fr/hal-00706701`.

[6] F. Fondement. Documentation succincte sur gmf. Technical report, Universitï¿½ de Haute Alsace, 2008. `http://fondement.free.fr/lgl/courses/mde/documentation_succinte_gmf.pdf`.

[7] S. Gï¿½rard, C. Dumoulin, P. Tessier, and B. Selic. Papyrus: A uml2 tool for domain-specific language modeling. In *Model-Based Engineering of Embedded Real-Time Systems*, Lecture Notes in Computer Science. 2011.

[8] ISO/IEC. 24744: Metamodel for development methodologies, 2007.

[9] S. Kelly, K. Lyytinen, and M. Rossi. Metaedit+ a fully configurable multi-user and multi-tool case and came environment. In *Advanced Information Systems Engineering*, Lecture Notes in Computer Science, pages 1–21. 1996.

[10] S. M. Kosslyn. *Image and Mind*. Harvard University Press, 1980.

[11] J. H. Larkin and H. A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65 – 100, 1987.

[12] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing domain-specific design environments. *Computer*, 34(11):44 –51, nov 2001.

[13] G. L. Lohse. A cognitive model for understanding graphical perception. *Hum.-Comput. Interact.*, 8(4):353–388, Dec. 1993.

[14] M. Minas. Visual specification of visual editors with VisualDiaGen. In *Applications of Graph Transformation with Industrial Relevance, Proc. 2nd Intl. Workshop AGTIVE, Charlottesville, USA*, 2003.

[15] D. Moody. The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on Software Engineering*, 2009.

[16] D. Moody and J. Hillegersberg. Evaluating the visual syntax of uml. In *Software Language Engineering*, Lecture Notes in Computer Science. 2009.

[17] OMG. http://www.omg.org/.

[18] L. Perini. Visual representations and confirmation. *Philosophy of Science*, 72(5):913–926, 2005.

[19] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 1983.

[20] B. Tversky. Cognitive principles of graphic displays. In *AAAI 1997 Fall Symposium on Reasoning with Diagrammatic Representations*, November 1997.

[21] S. A. White. Introduction to bpmn. Technical report, 2009. `http://www.omg.org/bpmn/Documents/Introduction_to_BPMN.pdf`.

[22] W. Winn. Learning from maps and diagrams. *Educational Psychology Review*, 3:211–247, 1991.

A Proposal for Consolidated Intentional Modeling Language

Joshua C. Nwokeji
School of Science and Technology
Middlesex University, London
England, United Kingdom
J.Nwokeji@mdx.ac.uk

Tony Clark*
School of Science and Technology
Middlesex University, London
England, United Kingdom
T.N.Clark@mdx.ac.uk

Balbir S. Barn†
School of Science and Technology
Middlesex University, London
England, United Kingdom
B.Barn@mdx.ac.uk

ABSTRACT

Intentional modeling (IM) focuses on intentions and motivations of software systems rather than behaviours. KAOS ("Knowledge Acquisition in autOmated Systems"), and i* ("Distributed Intentionality") are the two popular IM languages used in requirement engineering. Each of these languages are defined as a collection of intentional elements, and intentional properties. However, these intentional elements are fragmented across IM languages, and thus limited in supporting detailed requirement analysis. Our proposed solution is to combine these two languages into a consolidated modeling language using a Model Based Software Engineering (MBSE) language integration technique, in EMF-Ecore, and develop a graphical tool for the new modeling language. The graphical tool is applied on a case study to show that it supports detailed requirement analysis. The rationale behind this paper is to provide the Software Engineering Community with a richer but less cumbersome intentional modeling language that can support detailed requirement analysis, this can reduce the cost associated with incomplete requirement analysis during software development.

General Terms

Intentional Modeling, KAOS, i*, Requirements Engineering

Keywords

Modeling Languages, MBSE, EMF, Requirements Analysis

1. INTRODUCTION

*http://www.eis.mdx.ac.uk/staffpages/tonyclark/
†http://www.semoris.com/
http://mdx.academia.edu/BalbirBarn

GMLD '13, Montpellier, France

http://dx.doi.org/10.1145/2489820.2489826 ...$15.00.

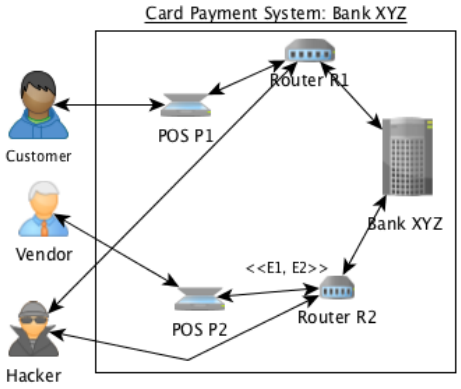


Figure 1: Card Payment System for Bank XYZ Plc.

Intentional Modeling (IM) is a modeling paradigm in software development that defines the *rationale/why* for the existence of a system [20]. According to Yu et al [20], IM makes it easier to understand stakeholders' goal, thus clarifying the drivers behind business decisions, and providing traceability for system changes. Other benefits of intentional modeling have been noted in [9] [16] [18] to include support for requirement specification, checking the completeness of a requirement specification, and providing alternatives to choose from during system design. Varieties of Intentional Modeling Languages (IMLs) have been used in requirement engineering, popular among these are i* [7] and KAOS [14]. Each of these languages are defined as a collection of intentional elements, and intentional properties. Although they offer a lot of benefits in Requirements Engineering (RE). The comparison of IM key concepts (elements) with popular IM approaches (KAOS, and i*) shown in Table 1 reveals that IM elements are fragmented across IM languages, and no single IM language has sufficient modeling elements to support detailed requirement analysis. For example, i* has no model element for *Obstacle*, thus can be difficult to support obstacle analysis. While *KAOS* does not explicitly define the concept of an intentional *Actor*. We describe a case study below and used it to explain the core model elements of intentional modelling language.

Bank XYZ provides *Point of Sale-POS* services via en-

Table 1: Comparison of IM Key Concepts with i* and KAOS

IM Key Concepts \ IM Approaches	i* [7]	KAOS [14]
Actor	√	?
Goal	√	√
HardGoal	√	√
SoftGoal	√	?
Task	√	ab
Constraint	ab	?
Agent	√	√
Strategic Dependency	√	ab
Conflict	ab	√
Obstacle	ab	√

Legend:
√-included in the Language
ab-Absent in the Language
?-Not explicitly defined in the language

crypted (but slow), and/or unencrypted (but fast) wireless *Routers* to *Vendors*, as shown in Figure 1. Each vendor sells products to *Customers*, and upload daily transactions to the *Bank*. The *Bank* should protect all transactions from *Hackers*, and limits daily transaction to *500* pounds. *Hackers* sniff around in order to steal *credit card* information from *Customers*.

The core intentional elements and properties required to model the requirements of this system are explained below. A detailed definition and descriptions of other intentional elements can be found in [14] [7] [19] [13] [2]. An **Actor** is an active entity such as human *e.g. Vendor* or machine *e.g. Router* capable of performing actions within a system [21]. A **Goal** is a statement that describes what a system is designed to achieve *e.g.*, *successful transactions*, or the intentions of a given actor in a system, *e.g.*, *steal credit card information* [4]. A *Goal* that has clear criteria for its satisfaction is known as a **HardGoal** while a **SoftGoal** is a type of goal without a clear criteria for its satisfaction [7]. A **Conflict** is a trade off between goals, a situation where the satisfaction of one goal prevents the satisfaction of another [17]. For instance, the goal *process customer request on time* is in *conflict* with another goal *secure transaction*. A **System Constraint** shows certain regulations on a system, it can be used to implement government regulations on a system [18], *e.g.*, *daily transaction* $\leq$ £500. An **Obstacle** is an undesirable condition to the satisfaction of a given goal in a system [14] [18], for example, the goal *card readable* can be obstructed by a *faulty POS terminal*. A **Strategic Dependency** defines *strategic relationship* between actors, *i.e.*, how actors depend on each other to achieve their intentions [9] [21], for instance a *Vendor* will depend on the *POS Terminal* to satisfy the goal *happy customer*.

1.1 Challenges

Our aim is to carry out a detailed requirement analysis, check the completeness, and implement the constraints for the Point of Sale (POS) system described above. Thus we need to answer the following research questions:

Research Question 1: Which IML will be used to check the completeness of the requirement specification of this system (described above) with respect to *Actor*, its *Goals*, and the *Obstacles* to the satisfaction of those *Goals*?

Research Question 2: How can we precisely express constraints such as *daily transaction* $\leq$ £500 that occur in the system (described above) and as part of the intentional model in the form of elements such as goals?

To answer these questions we compared the key model elements (concepts) of two IMLs-*KAOS, and i** as shown in Table 1. This comparison reveals that IM elements are fragmented across IMLs, thus we cannot find a single IML that explicitly gives us a complete set of these IM element (*Actor, Obstacle, Goal, Constraints*) required to check completeness and the constraints imposed on our system, *e.g.*, *i** has no model element for *Obstacle, and Constraints*, while *KAOS* does not explicitly define an *Actor, and Constraints*. Although the completeness of a system has been defined as *assigning all goals to Actors that operationalise them* [14,17], we argue that the existence of an *Obstacle* can hinder the operationalisation of a *Goal*, thus rendering the system incomplete. From the case study, the *Goal- successful transaction* can be operationalised by assigning it to an *Actor-POS Terminal*, but if the *Obstacle- card unreadable* is not resolved, then the *Goal* remains unsatisfied and the system incomplete. Therefore, without the existence of these intentional elements (*Goal, Actor, and Obstacle*), it will be difficult to use any IML to perform a *completeness check*.

The second research question raises issues to the meaning of IML elements. For example, both KAOS and i* include goals that are essentially constraints on system configurations. However, neither of the IMLs evaluated in this paper provide a language for expressing the configurations over which the constraints can range. Therefore, it is not possible to check whether a goal is correct with respect to the elements that it refers to. Such automated checks are key to providing tool support for IML and are provided in most tools for software engineering.

Our proposition is that a new IML with richer IM elements is needed to address these limitations. The new IML will include all the elements necessary to perform *complete* IM by constructing a rationalised union of the IMLs under study. In addition, the new IML will include a language for expressing the referents of goals and thereby a route to basic tool support for IM. Therefore, our contribution is to develop a new IML that contains sufficient intentional elements. Secondly, we develop a new graphical modeling tool to support our new language and apply it to the case study above. We also show how our language can be used to support *completeness check* and *constraint analysis* using the Object Constraint Language (OCL).

This paper is organised as follows, Section 2 gives a brief description of the two popular IM approaches used for this study, followed by a review of related work in Section 3. In Section 4 we present the design of our new modeling language *CIML*, and describe the tooling process for the graphical editor in Section 5. Section 6 shows how our graphical editor is applied to model the case study. This is followed by a completeness and constraint check in Section 7, and finally a conclusion in Section 8.

2. IM APPROACHES

KAOS is a requirement engineering framework whose primary focus is to explicitly represent all system goals, the

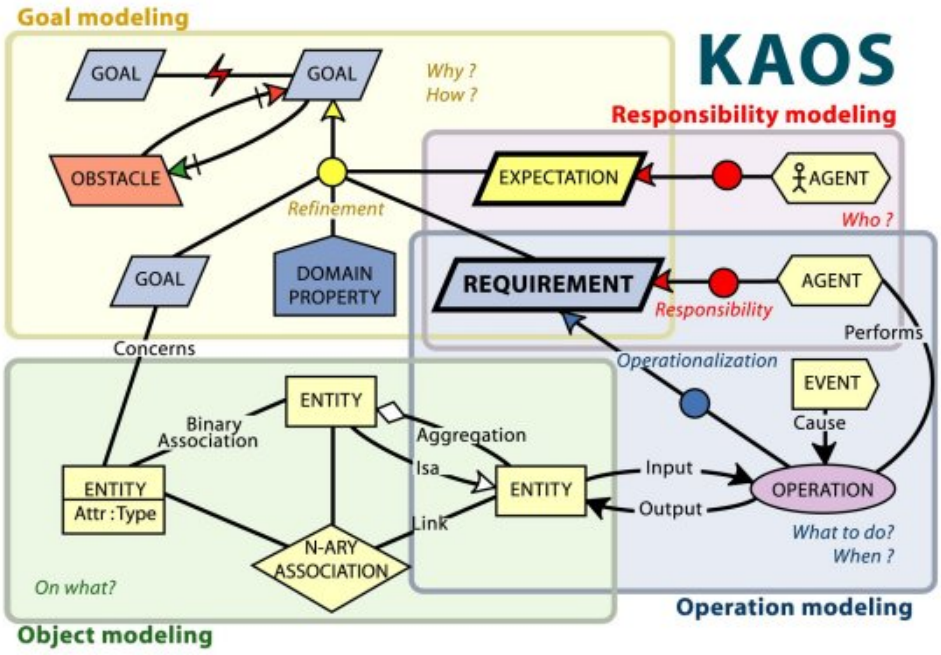


Figure 2: KAOS Meta-Model [14]

conflicts, and obstacles between these goals, the objects that are responsible for satisfying these goals, and the operations that are triggered as a result of the interaction between the goals, and objects [5] [9]. The KAOS meta-model shown in Figure 2 consists of Goal Model, Object Model, Operation Model and a Responsibility Model [5]. The goal model is a graph that represents system goals, subgoals, obstacles, and the agents responsible for satisfying the goals [19]. The object model defines the passive and active objects in a system such as agents, entities and associations [14]. The operation model describes the actions/activities that agents perform to satisfy the goals assigned to them, while the responsibility is a derived model that assigns responsibility to agents.

As described in [2, 9, 21], and shown in Figure 3, the **i*** language offers two types of models namely *strategic dependency* (SD), and *strategic rationale* (SR) models. An SD model describes the relationship where one actor called the *depender* depends on another actor called the *dependee* to satisfy a given intentional property called the *dependum*, the four types of SD corresponds to the types of dependum, e.g. if the dependum is a goal, the SD is called *Goal Dependency*, and so on [1]. The *strategic rationale* (SR) model is another graph of nodes and links that describe the intentional property of each actor, and explain the rationale behind each actor's dependencies [1, 7]. Four types of links can be used to describe actors intentions. The *means-ends* link shows that an Actor can perform a task (means) to achieve a particular goal (end); a *task decomposition link* describes the steps that can be taken to perform a given task; a task can be decomposed into *subtask*, *subgoal*, *softgoal*, and *resource* [1]. The *contribution link* describes the type of contribution offered by each intentional element to another [1].

3. RELATED WORK

Various *Model-Driven Engineering* approaches such as *model transformation and integration* [12] [13] have been used in research and practice to derive integrated IML from existing language definitions, as discussed below. However, a major set back to current approaches is that they integrate parts of IML required for their problem domains, leaving out some important ones, and failed to show how their unified models can be used to support detailed requirement analysis, such as expressing and resolving *Obstacles*, and checking *Constraints*.

3.1 Model Integration Approach

Model integration techniques have been applied in [13] to design the so called *unified goal oriented language* (UGL). This technique integrates the basic concepts of three intentional modeling languages namely KAOS, i*, and GRL by applying a comparative analysis to their language definitions, and identifying similar modeling concepts in the three languages. The similar concepts identified are then combined into a single super class, e.g. the operation, requirement, expectation model elements from KAOS, and the task model element from i* are combined into the *superAction* class in UGL. A major contribution of this approach is to define a graphical tool in EMF/GMF that allows a model defined in one language (say KAOS) to be viewed in another language (say i*).

Similarly, the meta-model for a *unified requirement modeling language* (URML) is presented in [6]. This approach integrates various aspects of system modeling such as goal modeling, feature modeling, product line modeling, and requirement modeling into the so called URML. This approach claims that such integration can encourage a "homogeneous visualization" of requirement across all the multi-disciplines involved in the software design process. Basic model elements from KAOS, i*, BMM, and TOGAF have been integrated in [4] to develop a language called *ARMOR*. This

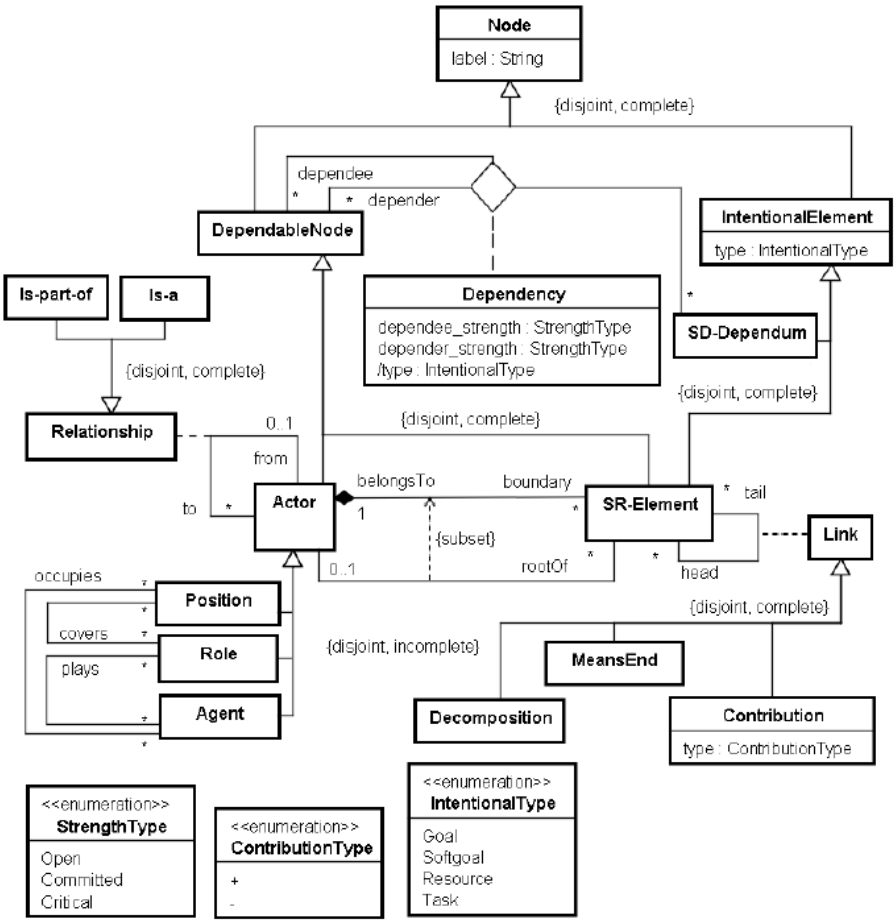


Figure 3: i* Meta-Model [2]

language is an attempt to incorporate requirement engineering into enterprise architecture.

3.2 Model Transformation Approach

A one directional transformation between i* and KAOS has been proposed in [11], and implemented in [12]. This approach defines a transformation relation between i* and KOAS in Atlas Transformation Language (ATL), and argues that such a transformation makes it easier to understand the differences between both languages, convert one model to another, and assist developers to make informed decision on the language to use in requirement analysis. A method to derive object-oriented conceptual models from i* has been proposed in [1]. This approach defines set of transformation rules that allows an i* model to be transformed into a conceptual model used for software development.

3.3 Empirical Comparison

Empirical comparison of IM languages have been reported in [3, 10, 15, 19]. Matulevicius and Heymans compared the quality of KAOS and i* in [10] by means of a *semiotic quality framework* experiment and infer that model quality is a function of user experience. [15] reports an empirical experiment to compare the suitability of KAOS, i*, and NFR (Non Functional Requirement) in modeling collaborative systems. The study reveals that the three IM languages require further enrichment and modifications in order to model collaborative systems. A set of frameworks have been used to compare i* and KAOS in [3, 19]. The aim of these comparisons is to determine the potentials, and setbacks of IM languages in other aspects of modeling such as business process modeling.

A major limitation of these approaches is that none of them show how intentional elements such as obstacles, conflicts, and domain properties can be applied to check completeness or consistency of a model, nor show how the integrated languages can support detailed requirement analysis such as checking constraints.

4. CONSOLIDATED IML (CIML)

In Figure 4, we present the *abstract syntax* of our new language: *CIML*. The *root element/container* consists of two *abstract classes*: **Node**, and **Link**. A *Node* is used to represent a model element such as *Actor, and Goal*, while a *Link* represents the relationship between model elements. For instance, *ObstructionLink* assigns *Goals* to *Obstacles*, thus showing an *obstruction* relationship. A *Node* can be any of these four *abstract classes*: **DependableNode** - a type of node that has a *dependency relationship* implies that it can allow other nodes to depend on it in which case it is called

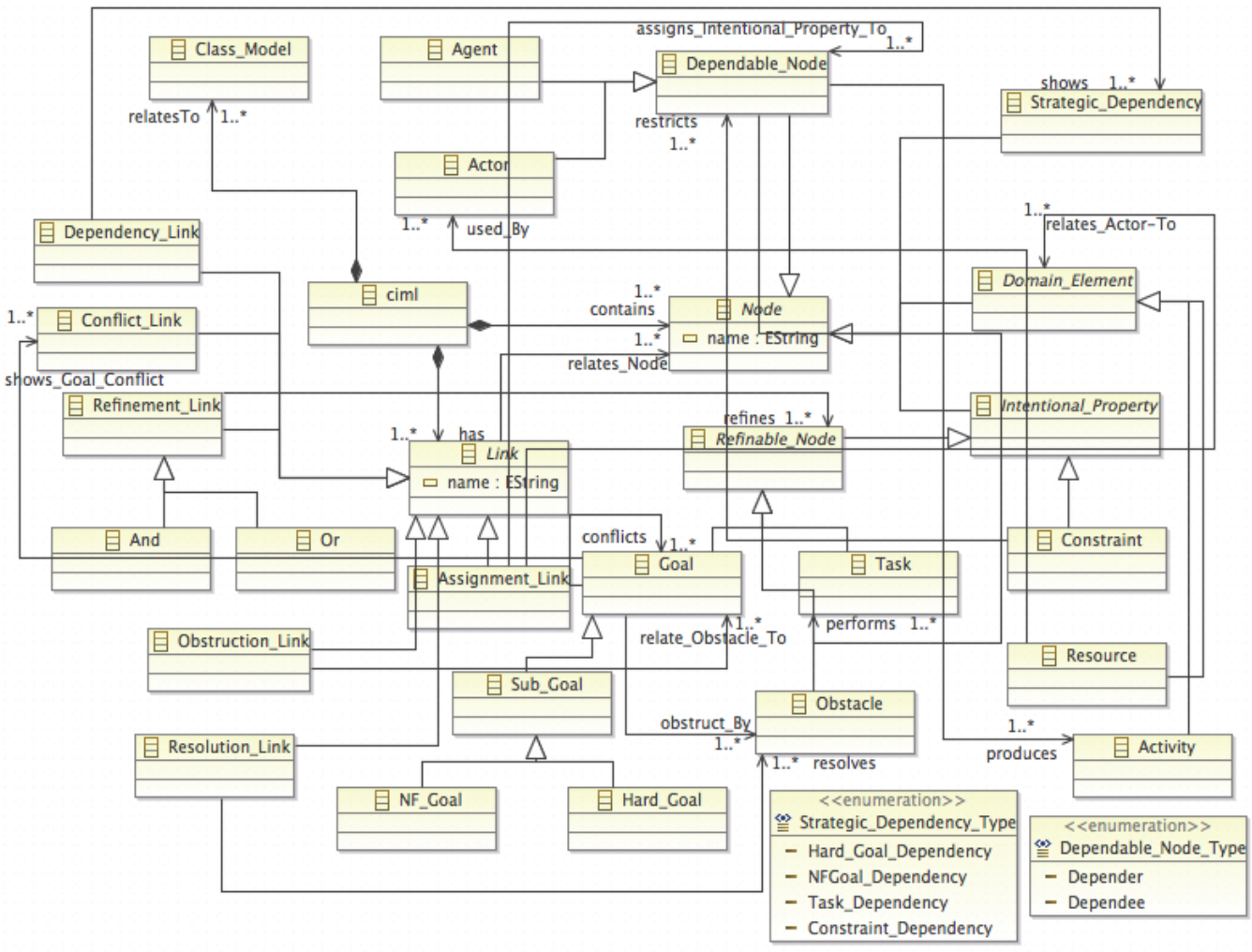


Figure 4: The CIML Abstract Syntax (Meta-Model)

Table 2: Mapping of key CIML concepts with i*, and KAOS

CIML Element/Symbol	Mapped Elements in i*	Mapped Elements in KAOS
Actor- a rectangle without an icon	Actor, Role, and Position	Environmental Agent
Agent- a rectangle with dashed border and Icon	Agent	Software agent
Goal-a rounded rectangle with no icon and thick border	Goal	Goal
Resource- a rectangle with no icon and dotted border	Resource	Entity
HardGoal- A dashed rounded rectangle without icon	HardGoal	Requirement
NFGoal-A dotted rounded rectangle without icon	Softgoal	Expectation
Task-An ellipse without an icon	Task	Events
Obstacle-Ellipse with an Icon	none	Obstacle
Conflict-same symbol as Goal	none	Conflict
Constraint-Ellipse with icon, thick and dotted border	none	Pre, and post conditions
Activity- ellipse with icon and thick border	none	Operation
RefinementLink: *And*-closed arrow with solid ellipse.*Or*-closed arrow with solid rectangle	DecompositionLink, is-a, is-part-of	RefinementLink
ObstructionLink-square	Hurt	ObstructionLink
DependencyLink-filled rhomb	DependencyLink	none
AssignmentLink-arrow	none	Operationalization Link, Responsibility Link, and Assignment Link

(Depender), and it can depend on other nodes *(Dependee)* to satisfy its intentions [7]; **Domain Element**-any model element that is part of the system's environment *e.g.*, domain property; **Intentional Property** describes the objectives of an Actor within a system *e.g.*, Goal [7]; **RefinableNode** is an abstract class that models some intentional properties that can be decomposed into sub properties e.g. a goal has subgoals. **Dependency** shows the type of dependency relationship that can exist between two actors, the various types have been listed as *Enumeration literals* in Figure 4.

A *DependableNode* can be an *Actor* which can be mapped into Actor, Position, and Roles in i* [7], and Environment-Agent in KAOS [14]; or *Agent*-the same as Agent and SoftwareAgent in i* and KAOS, respectively. Goal [7, 14], Obstacle [14], Task [7], and Constraint are types of *Intentional Property*. Unlike KAOS and i*, constraint is an intentional property in CIML, because it represents a restriction imposed on a system by either an internal stakeholder such as Actor/Agent or an external stakeholder such as Government. In any case, it represents intention of a stakeholder, thus an intentional element. Constraints are usually satisfied before or after using the system are respectively known as pre and post conditions [14]. A *Goal* can be a *Hard-Goal*, which implies a clear criteria for its satisfaction, or a *non-functional-NFGoal*, which implies lack of clear criteria for its satisfaction e.g. ensure happy customer. *DomainElement* in CIML can be a Domain Property [14], Activity, or a Resource [7], in i* resource is a type of an intentional element and defined as an entity such as data used by an actor [7]. From this definition we categorise resource as a type of domain element rather than intentional element because we argue that a resource such as *debit card* does not represent actor's intention. Activity is used to represent the actions performed by a actor or agent to satisfy its intention with the domain of the system. A (strategic) dependency relationship can exist between Actors. The dependency relationship between Actors is established when an Actor (called depender), depends on other Actor(s) (called dependee) to achieve an intention (called dependum). For instance, the Vendor depends on the POS terminal to achieve the constraint *daily transaction not greater than £500.* A dependum can be a HardGoal, a NFGoal, a task, and/or a Constraint. The four types of strategic dependency in CIML are listed as enumeration at down right side of Figure 4. Each strategic dependency corresponds to the type of the dependum. For instance, if the dependum is HardGoal, the strategic dependency is called HradGoal dependency. [7].

Refinement Link is used to decompose RefinableNodes into sub-nodes. They can be And refinement which shows a compulsory alternative, and Or refinement which shows a non-mandatory alternative [14]. DependencyLink shows the type of dependency between actors [7]. Conflictlink shows a conflict relationship between goals, resolutionlinks shows that an obstacle has been resolved, while assignmentlink assigns an actor/agent to a goal [14]. Each element in CIML is equivalent to one or more elements in KAOS, and/or i*. We demonstrate this by mapping CIML elements with elements of KAOS and i* as described in Table 2 For instance, AssignmentLink is mapped into OperationalisationLink, assignmentLink, and responsibilityLink in KAOS [14], and contribution/means-end link in i*. Similarly, an Actor in CIML is equivalent to Actor, Position, and Roles in i*, or EnvironmentAgent in KAOS. Other examples are shown in Table 2.

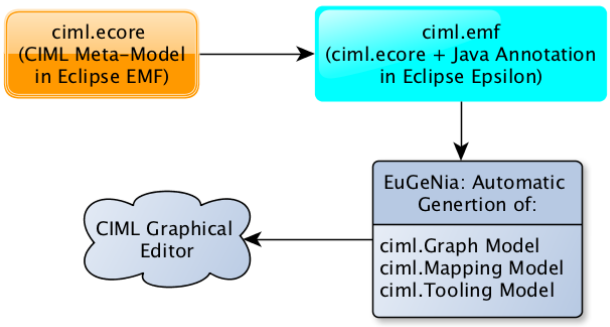


Figure 5: CIML Tooling Process

A CIML model is linked to a Class_Model. This provides a means for modelling the system configurations used as the referents of goals and other IM elements. We assume that the reader is familiar with standard class models involving classes, attributes, associations and inheritance and therefore we do not include the details in figure 4.

We assume that the modeller is free to construct an information model in the form of classes that expresses just those aspects of the system required by the intentional elements. Given that each intentional model has a class model whose instances are the referent of goals *etc.*, it is possible to use a precise language to express the goal conditions. Of course, part of the reason for constructing an IM during the early stages of system development is to be informal. However, a possible route to precision, by transforming informal constraints to formal constraints, is attractive, and is a pre-requisite for tool support. A suitable language for defining such conditions is OCL since this is defined to have instances of class models as its referent. Therefore, our proposed CIML will assume that OCL can be used to express intentional elements where appropriate, and also allow formal and informal constraints to be freely mixed.

5. THE CIML TOOLING

We developed the *CIML* tool in the Eclipse Graphical Modeling Framework (GMF) using *EuGENia* [8]. A tool for developing graphical editors. The diagram in Figure 5 describes the steps involved in *CIML* tooling. First, we design the *abstract syntax or Meta-Model* for *CIML* using Eclipse Modeling Framework (EMF) and annotate it as *ciml.ecore* as seen in Figure 5. The next step involves converting the *ciml.ecore* model to an EmfaTic file and annotating it with Java to describe the attributes of the objects/nodes and their connectors/Links. Emfatic is a development environment that allows *.ecore files to be annotated with Java. To further explain the CIML tooling process, a sample of the Java annotated *ciml.ecore* is shown in the latter part of Section 5. *Line 1* declares the *namespace* which specifies the location of *ciml.ecore*, *i.e. "http://ciml/1.0"*. *Line 4* tells *EuGENia* that the root element or container is *ciml*, and therefore should not be included in the diagram. The *gmf.node* annotations in *Line 10* are used to specify that a particular *Eclass i.e Ecore Class* is a node, and further defines the attributes such as figure, label, and icon for a particular node. Nodes are called Objects in the palette, see Figure 6. *Connectors or Links* are specified in the EmFaTic file using the *gmf.link* annotations as seen in *Line 15*, and defines attributes such as target-decorations, source, and target of the connector.

After annotating all the *Eclasses* with java, then we used EuGeNia to generate the models that describes the attributes of the CIML graphical editor, these include ciml. There are three basic types of models generated by *EuGENia*, these include the *Graph Model, Tooling Model, and Mapping Model*. The *Graph Model* describes the elements of the graphical editor such as connectors, labels, decorations; *Tooling Model* specifies the elements tools that will be available in the palette of the graphical editor; and the*Mapping Model* maps the elements in the Graph Model and creation tool with the Meta-Model defined in Ecore [8]. Once these models are generated, then the diagram plugin for CIML is created. Finally, after running a new eclipse runtime the CIML graphical editor is generated. The description of all the symbols used for each CIML model element is provided in Table 2, while the graphical convention is shown in Figure 8.

```
1  @namespace(uri="http://ciml/1.0", prefix="ciml")
2  package ciml;
3
4  @gmf.diagram(foo="bar")
5  class Ciml {
6    val Node[+] contains;
7    val Link[+] has;
8  }
9
10 @gmf.node(label="name")
11 abstract class Node {
12   attr String name;
13 }
14
15 @gmf.link(target.decoration="arrow",
16 source="froma", target="toa")
17 abstract class Link {
18   attr String name;
19   ref Node[+] froma;
20   ref Node[+] toa;
21 }
```

6. APPLYING THE CIML TOOL

We used our CIML tool to model the case study described in Section 1 as shown in Figure 6. The *Goal (G1)* of *Bank XYZ* is to provide *excellent POS Services*, this implies that all transactions must be successful, secure, and customers must be happy. Thus its refinement into three compulsory alternative *SubGoals SG1-successful transaction; and SG2-secure transaction; and SG3-Happy Customer* using the*And Refinement Link*. *SG1-successful transaction* means that all smart cards must be readable, and daily transaction must be uploaded successfully. We thus refined *SG1* into two *HardGoals: HG1-Card readable, and HG2-daily transaction uploaded*. Similarly *SG2-secure transaction* implies that the smart card information is encrypted, or that the vendor upload daily transactions using the secure encrypted option E1. This lead to refinement into non-compulsory alternatives *HardGoals:HG3-smart card information encrypted, or HG4-daily transaction uploaded with secure option*. The subgoal *SG3-happy customer* means that all customer requests are processed on time. The *Vendor* will depend on the *POS terminal* to achieve this goal. This shows an instance of strategic dependency. However, the hardgoal *HG5-process*

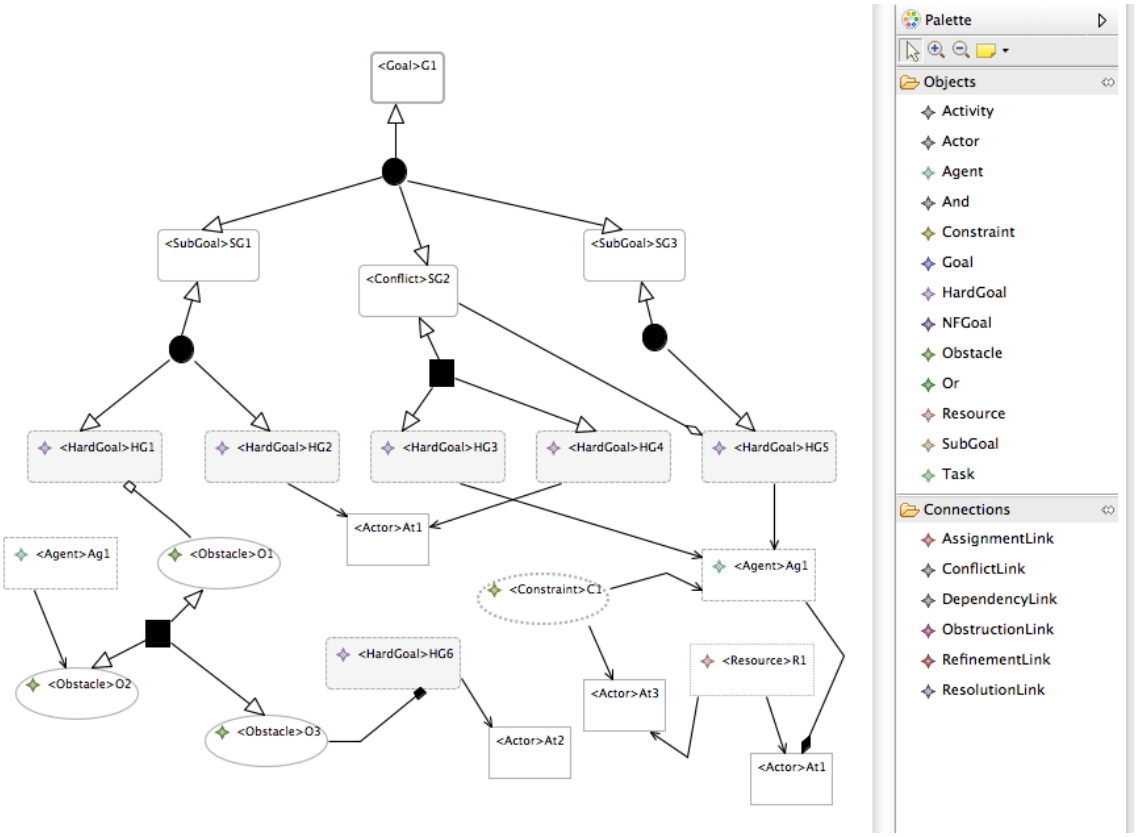


Figure 6: The Case Study Modeled with our CIML Tool

customer request on time means that a faster but less secure network will be used, this is in *Conflict* with *SG2-secure transaction*.

Although the hardgoal *HG1: card readable* can be satisfied by assigning it to the *Agent; Ag1-POS terminal*, the *Obstacle O1: card unreadable* can hinder its satisfaction. To resolve this obstacle we have to find its possible causes such as a faulty POS terminal or faulty smart card. Then, we refine the obstacle into *O2: faulty POS terminal or O3: card unreadable*, and resolve them by assigning an agent/actor to each of them. *Obstacle O3* is refined into a *hardgoal HG6 -replace card*, and assigned to the *Actor; At1-Vendor*, while Obstacle O2 is assigned to an *Actor; At2-Repair Engineer*. This provides a means of analysing and resolving obstacles in CIML using three intentional elements (Actor, Agent, Goal, and Obstacle), which is one of our contributions in this paper. Other hardgoals are assigned to *Actors, and Agents* as shown in the Figure. The *Constraint, C1-daily card transactio* $\leq$ *£500* imposed on the *Actor; At3-Customer* is enforced by assigning it to the *Agent;Ag1-POS terminal*. The *Resource;R1-smart card* can be used by the *Vendor or the Customer*.

Each CIML model is associated with a class model that describes the referents for the intentional modelling elements. We do not require any bespoke tooling for class modelling since it is ubiquitous. Figure 7 shows a class model for the case study. In overview, each shop contains a collection of POS terminals that maintain a history of their transactions. Each transaction involves a card that is owned by a customer and that may have become unreadable (perhaps the risk of a card being unreadable increase with the number of uses). Each transaction contains the time it takes for confirmation to be returned by the bank. Both the shop and the bank are external connections on a network. Paths through the network involve routers that may be encrypted or not and which have processing speeds (perhaps the level of encryption affects the speed of processing). A hacker performs a number of *sniffs* on routers that may or may not be successful (depending on the level of encryption).

7. VERIFICATION

The CIML provides a rationalised union of the intentional elements required to be complete with respect to a system. Our claim is that this new language allows us to check for completeness and this section uses OCL to specify this property of a CIML model. In addition, CIML includes system configurations expressed as class models. Therefore, OCL can also be used to express constraints over the system configurations including, where appropriate, intentional model elements.

7.1 Completeness

A requirement specification is complete with respect to *Actor, Goal, and Obstacle* if all goals have been assigned to an actor or agent, and all obstacles to the goal have been resolved. All of the assignments in a model are returned by the following query:

```
1  context ciml::assignments() =
```

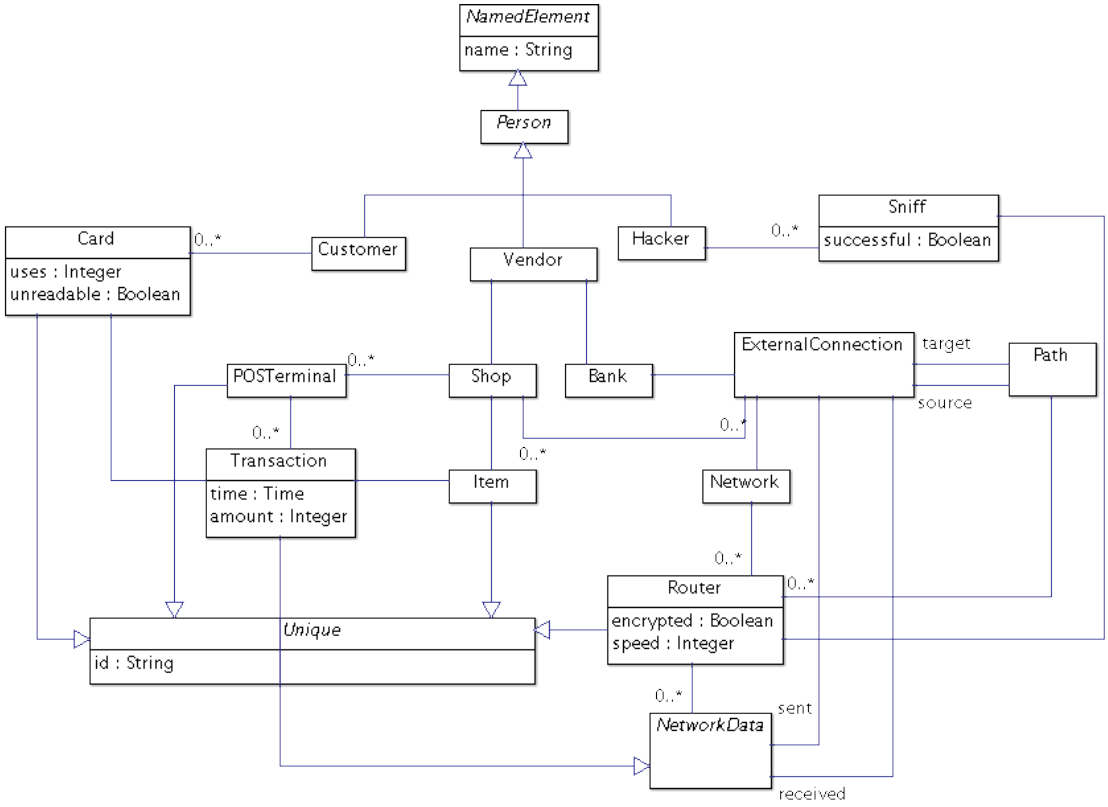


Figure 7: A Class Model for the Case Study

```
  links->select(l | l.oclIsKindOf(Assignment_Link)
```

Resolution links are returned as follows:

```
context ciml::resolutions() =
  links->select(l | l.oclIsKindOf(Resolution_Link)
```

Goals are returned by the following query:

```
context ciml::goals() =
  nodes->select(n | l.oclIsKindOf(Goal)
```

Obstacles are returned by the following query:

```
context ciml::obstacles() =
  nodes->select(n | l.oclIsKindOf(Obstacle)
```

An obstacle is resolved when there is a resolution link for it:

```
context Obstacle::resolved(model) =
  model.resolutions->exists(r |
    r.resolves->includes(self))
```

A goal is assigned in a model when the following predicate is satisfied:

```
context Goal::isAssigned(model) =
  model.assignments()->exists(a |
    a.goal = self and
    let target =
    a.assigns_intentional_property_to
    in target.oclIsKindOf(Agent) or
       target.oclIsKindOf(Actor)
  )
```

The following OCL expression can check completeness:

```
context ciml inv Completeness:
  goals()->forAll(g | g.isAssigned(self)) and
  obstacles()->forAll(o | o.resolved(self))
```

The check for completeness is therefore, precisely defined and acts as a specification for any CIML tooling. This is possible because CIML is defined as a meta-model in a standard language, as opposed to the definitions of other IMLs, such as i* and KAOS that are often defined using concrete syntax based notations.

7.2 Constraints

Expressing the system as a class model allows us to precisely encode constraints using OCL. The constraints can be *invariants* that must be true of an instance of the class model in all stable configurations, or can encode intentional model elements, such as goals, that represent desirable system configurations.

The following OCL constraint requires that there is a limit of £500 on daily transactions at any given POS terminal:

```
context Shop inv:
  terminals->forAll(t |
    transactions.groupByDay()->forAll(T |
      T.amount->sum <= 500))
```

where the query operation `groupByDay()` maps a set of transactions $T = \{t_1, t_2, t_3, \ldots, t_n\}$ to a set of disjoint sub-sets $\{T_1, T_2, \ldots, T_k\}$ such that the transactions in each T_i occur on the same day.

We can express the condition that routers process information at different speeds depending on whether they use encoding or not:

```
context Network inv:
  routers->forAll(r1 r2 |
    r1.encrypt and not(r2.encrypt)
```

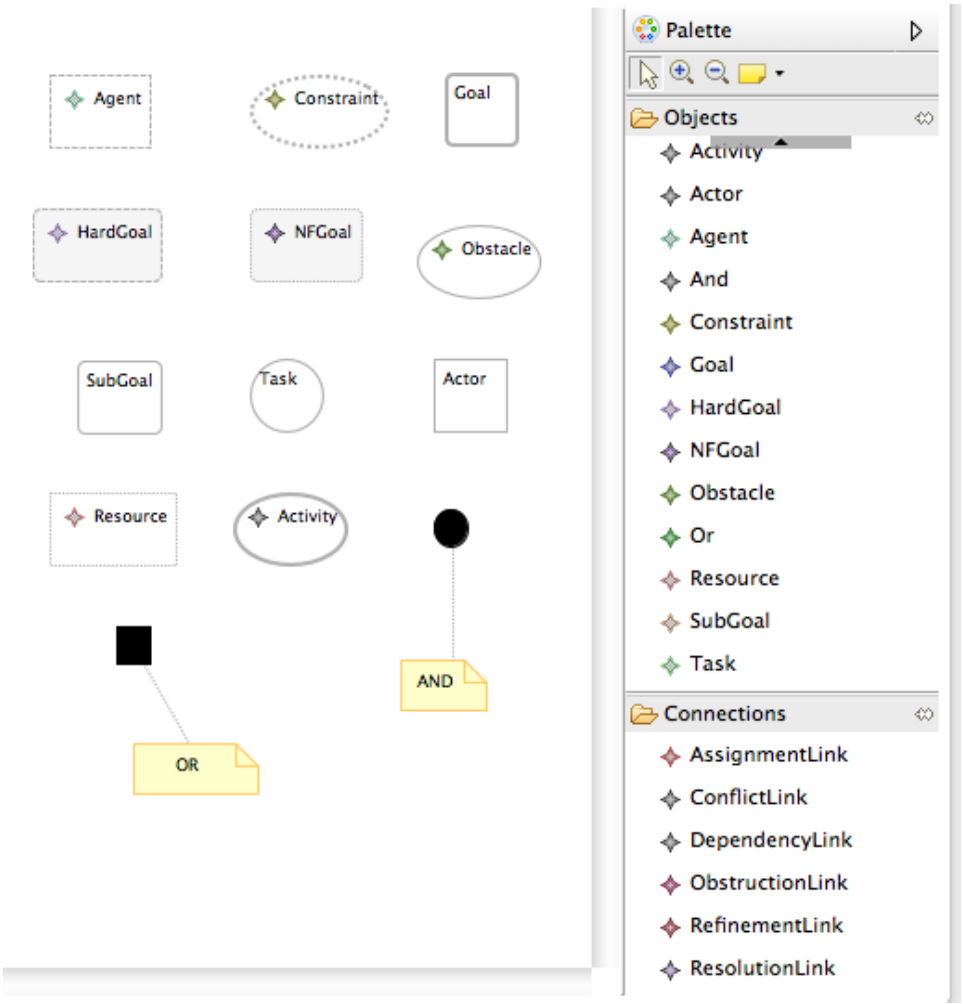


Figure 8: Graphical Convention for CIML

```
        implies r1.speed > r2.speed
    )
```

A goal, from a customer's point-of-view, is that a transaction should not take too long. Of course an acceptable duration will depend on the situation, but we assume that it is a known constant `transDur`. OCL does not allow us to express a *desirable* constraint, *i.e.*, one that might be true but which might be (reasonably) refuted by the goal of another system agent. Furthermore, goals are unlike OCL constraints in that they may be blocked through obstacles that need resolution.

Therefore, we suggest that in order to encode goals and other intentional modelling elements as expressions in OCL, we will require additional OCL features. The design of such features is left as further work. However, we propose a simple extension here that allows goals to be defined from the perspective of a stakeholder in the system. Assuming that there is a query operation called `getPathToBank()` defined for a transaction that returns a network path that can be used to send transaction data to the bank, then:

```
context Transaction goal Wait for Customer:
  getPathToBank().routers
     ->iterate(r time_taken = 0 |
          time_taken + r.speed) < transDur
```

Clearly there is more work to be done here in order to express intentional elements as precise constraints in OCL. However, we claim that the CIML is a contribution and that allowing system states to be attached to intentional models via class models is a significant step in achieving this. Further work will include using stereotypes on class model elements to attach them to specific IM elements, and extending OCL in order to support different types of IM element.

8. CONCLUSION AND FUTURE WORK

In this research we analysed two major IML used in industry and research, and discovered that intentional elements are fragmented across these languages, thus their limitation in singularly supporting certain analysis required in requirement engineering. We address this limitation by proposing a new language with richer but less cumbersome intentional elements, and design a graphical editor for our new language. We applied our language to model a case study using the graphical editor and further show how the completeness and some constraint can be checked with our model using OCL.

Our next step will be to automate these checks by encoding the completeness and constraint check into our graphical

editor using Epsilon Validation Language (EVL). Also, we intend to develop a concrete syntax for our language, and publish it over the internet as an Open Source Software. For further validation, we will apply our language in other areas of Software Engineering, such as Enterprise Architecture Alignment.

9. REFERENCES

[1] Fernanda M. Alencar, Beatriz Marin, Giovanni Giachetti, Oscar Pastor, Jaelson Castro, and Joao Henrique Pimentel. From i* requirements models to conceptual models of a model driven development process. In *PoEM*, pages 99–114, 2009.

[2] Carlos Cares, Xavier Franch, Lidia Lopez, and Jordi Marco. Definition and uses of the i* metamodel. In Jaelson Brelaz de Castro, Xavier Franch, John Mylopoulos, and Eric S. K. Yu, editors, *Proceedings of the 4th International i* Workshop, Hammamet, Tunisia, June 07-08, 2010*, volume 586 of *CEUR Workshop Proceedings*, pages 20–25. CEUR-WS.org, June 2010.

[3] A. Edirisuriya and J. Zdravkovic. Goal support towards business processes modelling. In *Innovations in Information Technology, 2008. IIT 2008. International Conference on*, pages 208 –212, dec. 2008.

[4] Wilco Engelsman and Roel Wieringa. Goal-oriented requirements engineering and enterprise architecture: Two case studies and some lessons learned. In *REFSQ*, pages 306–320, 2012.

[5] W. Heaven and A. Finkelstein. Uml profile to support requirements engineering with kaos. *Software, IEE Proceedings* -, 151(1):10 – 27, feb. 2004.

[6] J. Helming, M. Koegel, F. Schneider, M. Haeger, C. Kaminski, B. Bruegge, and B. Berenbach. Towards a unified requirements modeling language. In *Requirements Engineering Visualization (REV), 2010 Fifth International Workshop on*, pages 53 –57, sept. 2010.

[7] Jennifer Horkoff, Yu Eric, and Gemma Grau. istar quick guide, 2006.

[8] Dimitrios S Kolovos, Louis M Rose, Saad Bin Abid, Richard F Paige, Fiona AC Polack, and Goetz Botterweck. Taming emf and gmf using model transformation. In *Model Driven Engineering Languages and Systems*, pages 211–225. Springer, 2010.

[9] Alexei Lapouchnian. Goal oriented requirement engineering: An overview of the current research, 2005.

[10] Raimundas Matulevicius and Patrick Heymans. Comparing goal modelling languages: An experiment. In *REFSQ*, pages 18–32, 2007.

[11] R. Monteiro, J. Arau andjo, V. Amaral, and P. Patri andcio. Mdgore: Towards model-driven and goal-oriented requirements engineering. In *Requirements Engineering Conference (RE), 2010 18th IEEE International*, pages 405 –406, 27 2010-oct. 1 2010.

[12] R. Monteiro, J. Araujo, Vasco Amaral, M. Goulao, and P. M. B. Patricio. Model-driven development for requirements engineering: The case of goal-oriented approaches. In Ricardo Machado Joao Pascoal Faria, Alberto Silva, editor, *8th International Conference on the Quality of Information and Communications Technology (QUATIC 2012)*, number 8 in Quality of Information and Communications Technology, pages 75–84. IEEE Computer Society, 09 2012.

[13] Pedro Patricio, Vasco Amaral, Joao Araujo, and Rui Monteiro. Towards a unified goal-oriented language. In *COMPSAC*, pages 596–601, 2011.

[14] Respect-IT. A kaos tutorial, 2007.

[15] Miguel A. Teruel, Elena Navarro, Víctor López-Jaquero, Francisco Montero Simarro, and Pascual González. A comparative of goal-oriented approaches to modelling requirements for collaborative systems. In *ENASE*, pages 131–142, 2011.

[16] A. van Lamsweerde. Goal-oriented requirements engineering: a guided tour. In *Requirements Engineering, 2001. Proceedings. Fifth IEEE International Symposium on*, pages 249–262, 2001.

[17] A. van Lamsweerde, R. Darimont, and E. Letier. Managing conflicts in goal-driven requirements engineering. *Software Engineering, IEEE Transactions on*, 24(11):908 –926, nov 1998.

[18] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, 26:978–1005, 2000.

[19] Vera Maria Bejamim Werneck, Antonio de Padua Albuquerque Oliveira, and Julio Cesar Sampaio do Prado Leite. Comparing gore frameworks: i-star and kaos. In *Ibero-American Workshop of Engineering of Requirements*, Val Paraiso, Chile, July 2009.

[20] E. Yu, M. Strohmaier, and X. Deng. Exploring intentional modeling and analysis for enterprise architecture. In *Enterprise Distributed Object Computing Conference Workshops, 2006. EDOCW'06. 10th IEEE International*, pages 32–32. IEEE, 2006.

[21] Eric S.K. Yu. Social modeling and i*. In *Conceptual Modeling: Foundations and Applications*, pages 99–121, 2009.

Support for quality metrics in metamodelling

Xavier Le Pallec
LIFL - Université Lille 1
Cité Scientifique, bâtient M3
59655 Villeneuve d'Ascq Cedex, France
xavier.le-pallec@univ-lille1.fr

Sophie Dupuy-Chessa
LIG - Université Pierre Mendes France
B.P. 53
38041 Grenoble Cedex 9, France
Sophie.Dupuy-Chessa@imag.fr

ABSTRACT

The maturity of Model Driven Engineering facilitates the development of domain specific languages. Their creation relies on the definition of metamodels, but also on their corresponding visual notations. One can wonder about the quality of any new language, which can result in inunderstandable diagrams with inappropriate notations. Then our goal is to provide indicators about the quality of notations thanks to metrics. In this paper, we present functions that are necesary to calculate these metrics in a metamodeling environment. Then we introduce how metrics are integrated in a modeling environment named ModX.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

diagram quality, language quality, metrics, notations, metamodelling tool

1. INTRODUCTION

Thanks to the current outstanding technological improvements, accessing information anywhere, at anytime in a highly customizable way is becoming a reality. Unfortunately, the larger the field of possibilities becomes, the more complexity of design increases. Palen [28] has already confirmed this upward trend about design complexity for Human Computer Interaction ten years ago. Many scientific work has investigated Model Driven Engineering (MDE) this last decade in order to deal with this phenomena: 1) a model provides an efficient support for discussion, understanding a software architecture or components and 2) technological evolutions has generally less impacts on it. This has led to the definition of a lot of modelling languages for specific domains like interactive devices [5] or context of use [12].

GMLD'13, July 01 2013, Montpellier, France

http://dx.doi.org/10.1145/2489820.2489825 ...$15.00.

Unfortunately, a significant risk about those new languages is to produce useless and unclear diagrams, or to use inappropriate metamodel and/or modelling notation as a basis of a software environment. In such context, the quality of visual modelling languages and their diagrams becomes an issue. So our goal is to help authors of visual modelling languages by providing them with evaluation mechanisms that may assist the evaluation of quality of their production. In this work, we focus on quantitative metrics, that is to say, mechanisms that can be automatically computed. Despite the fact that this approach is restrictive (because evaluation of quality also includes qualitative aspects), it presents results that are useful and usable as it has been shown in the domain of code metrics.

First, we define the functions that are required to calculate metrics about visual notations. Then we explain how metrics can be defined in our metamodelling tool, named ModX. In this way, a designer may use our predefined metrics or define new ones.

Before getting any further, we can draw a list of terms uses in the remainder of this paper.

- **Abstract syntax (absStx)** refers to the set of concepts and their relations of a modelling language. It is usually specified through a metamodel. When one describes a abstract syntax, we may also use the term "abstract elements" (absElm) which refers to the components of the abstract syntax.
- **Concrete syntax (conStx)** refers to a visual notation which is associated to an abstract syntax. The latter may be linked to several concrete syntaxes. We consider here that a concrete syntax may be decomposed into concrete sub-syntaxes: UML has twelve types of diagrams and so twelve concrete sub-syntaxes. It is just a practical choice of classification. Similarly to abstract syntax elements, we also call elements of a concrete syntax or "concrete elements" (conElm), all components of its definition (shapes, images, layouts...).
- **Model (mdl)** is a set of elements that conform to an abstract syntax. Models are called "instances" of metamodels because the widespread formats MOF and EMF adopt the object paradigm.
- **Diagram (diag)** is a visual representation of model (or a part of it) that conforms to a concrete syntax.

The remainder of the paper is organized as follows. In Section 2, we present existing scientific work related to model

quality. We point out what approach we adopt and its related metrics. Section 3 describes ModX and its extensions to define metrics about models and metamodels. Section 4 reports conclusions and presents directions for future work.

2. CONTEXT

We first investigate existing work concerning quality of visual modelling languages and diagrams. We use them as a basis to propose an approach which is focused on quantitative metrics. Those ones rely on elementary functions which determine elements that have to be considered to compute automatic evaluations.

2.1 Related work

The quality of visual notations and diagrams is a scientific issue that has benefitted from the increasing interest in MDE. However quality frameworks have been defined long before the emergence of MDE:

- Hierarchical frameworks that propose a tree-view of quality. The root concept is the global quality which is further decomposed into different aspects which can be decomposed themselves. Each leaf is generally associated to a metric. One of the most famous framework related to quality is the ISO9126 standard (ISO 2001).

- Formal frameworks that propose methodology to construct quality metrics through mathematical properties that metrics have to respect. They aim to propose means to improve validity of new metrics while they do not provide any definition of quality.

- Causality-oriented frameworks [34, 20] which, unlike hierarchical ones, focus on the mutual influence between properties and intend to measure related correlations.

- Semiotic frameworks which are based on sign theory and have been initiated by Lindland [19]. They have been designed to evaluate any kind of diagrams. In [19], quality is detailed as syntactic, semantic and pragmatic. Syntactic quality evaluates the model according to the modelling language structure. Semantic quality deals with the correspondence between a model and its associated domain (or experts knowledge about it). Finally, pragmatic quality is about how the audience interprets the model. Even if [25] has demonstrated the benefits of Lindland's framework, the pragmatic quality has been extended with organizational and technical qualities [17] and MDE considerations have been integrated in the global framework [32] [23].

One of the main interests of those frameworks lies in their definition of quality and how they structure the way to deal with it. However, we think that their usability is reduced when they do not provide automatic evaluation mechanisms. For this reason, we promote an approach similar to hierarchical frameworks which lead to metrics-based evaluations. This metrics-based approach has already been applied to metamodels by [30] to calculate the theoretical conceptual complexity. Here we will propose to define metrics related to visual notations. Of course this approach can be combined with complementary proposals based on users' feedback.

2.2 Approach

If our approach does not consider the whole complexity related to quality, using metrics has the benefit of providing useful and usable results to designers. In the same way of tools like JDepend which dynamically compute metrics during coding, our tool aims to allow designers to dynamically compute metrics about the quality of their abstract and concrete syntaxes and relative diagrams. In that way, designers may avoid errors that may be revealed far later.

Our (meta-)modelling environment ModX proposes a set of already implemented metrics, that are proposed to designers of languages and diagrams. One of these metrics deals with informational density: it indicates if a concrete syntax contains too many elements. It is based on the magic number of Georges A. Miller (7 ± 2) which should not be exceeded. This is also relevant for diagrams. For example, an UML sequence diagram that presents more than nine lifelines may be considered as too complex (this is confirmed by the UML practitioners' feedback about diagram complexity [4]). So our tool can calculate metrics about diagrams quality as well as language quality.

Our approach also allows designers or experts in model quality to define themselves new metrics, like in [21]:

- Quality experts define metrics related to the general evaluation of abstract and concrete syntaxes and of any type of diagrams.

- Designers of a modelling language use the previous metrics to define efficient modelling languages and may further establish their own metrics for their diagrams but they may also modify existing ones.

Since Green's work on Cognitive Dimensions [10], visual aspects in Software Modelling have been considered as an important issue. Unfortunately, to the best of our knowledge, there are still few work and much less tools about their related quality [15]. If our approach intends to be global, for the moment, we focus on graphical concrete syntaxes, i.e. visual notations. We assume that the quality of diagrams is composed of two parts: the quality of its underlying concrete syntax and the quality related to its components (ex: number of elements, layout). We are mainly concerned by the first part (concrete syntax). We use the Physics Of Notations (PON) has defined by Moody in [24] as a basis of our work. If there is still no tool with metrics compliant with PON, an increasing number of work adopt it in order to evaluate visual notations ([9, 3, 35]). In this perspective, we propose here a software basis to such kind of study: we draw a list of elementary functions that PON criteria require and that each metamodelling tool should provide in order to define novel metrics.

2.3 Elementary functions to evaluate quality of concrete syntax

As we previously mentioned, specifying quality indicators about language quality remains a scientific issue. However, it is possible to draw a first list of properties which form elementary data to calculate metrics.

Metrics about the abstract syntax require an access to construction properties of language elements. MOF or EMF provide reflexive interfaces which are enough in this perspective. Similar interfaces are also needed concerning concrete syntaxes and unfortunately, they are not yet defined. We

	Semiotic Clarity	Perceptual Discriminability	Semantic Transparency	Complexity Managment	Cognitive Integration	Visual Expressiveness	Dual Coding	Graphic Economy	Cognitive Fit
visualRepresentation	X							X	
visualRepresentations	X								
semanticConstruction	X								
position		X				X		X	
size		X				X		X	
value		X				X		X	
grain		X				X		X	
color		X				X		X	
orientation		X				X		X	
shape		X				X		X	
complexityMechanisms				X					
contextRepresentation					X				
textualAnnotation							X		
preferredDrawingSupport									X

Table 1: Criteria and access functions

intend to propose this kind of interfaces by studying the PON criteria. For each of them, we indicate what elementary access functions it may require. Table 1 shows the correspondence between the PON criteria and the elementary functions. We use an informal formalism to specify functions: the star refers to a collection of values while braces permit to define objects whose properties are separated by comma.

Now let's study the PON criteria.

Semiotic Clarity. It is a bijective relation between the set of abstract elements and the set of concrete elements. To avoid redundancy, overload, excess or deficit of symbols, it is recommended to associate one and only one concrete element to each abstract one. This concrete element should not be linked to several abstract elements. So we need a function that permits to get the concrete element(s) for a given abstract one (`concreteElement ( absStx, conStx, absElm ) = conElm*`). We also need a function which returns the list of concrete elements for a given abstract syntax (`concreteElements ( conStx ) = conElm*`).

Perceptual Discriminability. It refers to the level of visual discriminability between two concrete elements. The higher the level is, the quicker the perception of diagrams will be and the lower cognitive effort will be. Visual distance is the main function of this criteria. However, there is still no well-established associated formula. Coming from cartography area, visual distance refers to several visual variables [1]: location (x,y), size, value (clear versus dark), grain (scale of texture pattern), color, orientation and shape. Visual distance between two elements is related to the number of visual variables on which both elements are different. Access to these variables is highly required when dealing with metrics about perceptual discriminability.

```
location ( conElm ) = { x, y }
size ( conElm ) = { width, height }
value ( conElm ) = brightness
grain ( conElm ) = { texturePattern, scale}
color ( conElm ) = { r, g, b }
orientation ( conElm) = angle
shape ( conElm ) = description
```

Semantic transparency. When a person reads a diagram, she/he has to associate a meaning to each kind of visual representation. For author(s) of the concrete syntax, this meaning corresponds to its original semantic. Unfortunately, an inadequate visual representation may lead non-expert readers to associate a wrong meaning. Semantic transparency refers to the distance between the perceived meaning by reader(s) for a particular visual representation and the meaning as it has been defined by language author(s). It is a criteria which is difficult to automatically evaluate because it relies on many parameters: reader profile, business context, related practices about visual symbols in corresponding business area... Evaluation needs a comparative study from different corpus rather than a formula that one has just to apply. So we do not detect here a required function.

Complexity Management. Visual representation of complex systems is a cross-disciplinary issue in science. We face a similar issue when it comes to represent complex diagrams, where complex means many elements and relations. Managing this complexity is necessary and implies that concrete syntaxes has dedicated mechanisms for that. Graphical inclusion and diagrams partitioning are examples of such mechanisms. So an interesting elementary function for this criterion should return the associated mechanism for a given *container-contained element* relation because such type of relation is the most likely to be used for complexity management.

```
complexityMechanism (absStx, conStx,
 container-contained element relation)
   = graphicalMechanism*
```

Cognitive Integration. When a reader browses diagrams thanks to complexity management mechanisms, she/he needs visual artifacts which help her/him to not forget for

which reason she/he came to this diagram or in which context the diagram takes place. In other words, it is important to integrate the part of the model, which is currently read, into the mental model of the reader. For example, a good practice in the Web is to display the path of the current page. A possible elementary function to evaluate this criteria may return visual elements for a given type of a complexity management mechanism.

```
contextRepresentation (absStx, conStx,
   container-containedElement relation)
        = conElm*
```

Visual Expressiveness. The more a concrete syntax exploits efficiently visual variables, the more it is visually expressive. Visual expressiveness positively and significantly impacts on cognitive effectiveness. This criteria uses the same properties as perceptual discriminability.

Dual coding. It is not recommended to only use textual annotations to visually represent an abstract element. But reinforcing visual artifacts with such annotations is a good practice. To evaluate this criteria, we need to know what textual annotations are used for a given concrete element.

```
textualAnnotation ( conElm ) = textualAnnotation*
```

Graphic Economy. A concrete syntax should not have a visual vocabulary that is too large/rich, that is to say with (too many values of) too many visual variables. Otherwise, mental activity dedicated to representation-meaning association will be too important and will negatively impact reading associated diagrams. This criteria implies for example to partition a model in different kinds of diagrams which are visually different (previously mentioned as concrete sub-syntaxes). This is not totally compliant with the semiotic clarity because it may imply symbol deficit. This criteria requires a function to get the number of associated concrete element(s) for a given abstract one. This function has already been mentioned above.

Cognitive fit. Abilities to fully exploit visual variables are relative to the support that is used to draw diagrams. For example, using complex images for concrete elements is not a good choice if related diagrams are planned to be drawn on paper. In a similar way, one also has to know if future readers (for a new modelling language) are not used to handling software engineering diagrams. If not, it is not recommended to use advanced mechanisms (like those previously mentioned) because they may require important mental efforts. A function to know what drawing support is planned to be preferred may be interesting because any language author has to indicate it and so to be aware of it. To deal with skills of future readers, we may refer to comparative studies as for semantic transparency. This is out of the scope here.

```
preferredDrawingSupport ( absStx, conStx )
      = support
```

Evaluating quality of concrete syntaxes with metrics implies to access to all listed primitives. We have implemented them in ModX in order to propose a tool able to calculate metrics about concrete syntaxes.

3. PROPOSITION

We have implemented the previous elementary functions in ModX [29]. These functions return values that are, of course, specific to ModX, that is to say, that they are limited to ModX 's abilities to represent concrete syntaxes. To report this implementation, we first describe ModX and the proviously mentioned abilities. We further show how it integrates the principle of metrics. Finally, we illustrate how to use elementary metrics in order to write a metric related to visual distance.

3.1 Presentation of ModX

ModX is a tool for modelling and metamodelling. It has been created in 2004 at Lille and is based of MOF v1.4 (Meta-Object Facility) [11]. Initiated in the context of the Kaleidoscope network of excellence [14], this editor aims to manipulate graphically any kind of models in Software Engineering (e-Learning [2], User Interface [31]). Figure 1 shows how ModX allows designers to create metamodels (abstract syntaxes), to associate them with visual notations (concrete syntaxes) and to edit derived instances i.e. models throw diagrams. There is no compilation or generation phase: syntaxes and models can be modified at any time and side effects are instantaneously visible. However, ModX's intercession rules remain simple: for example, if the type of an attribute is modified in a metamodel, all the associated properties in the models (present in ModX) will be reset. The interested reader could refer to ModX web site for more information (http://www.lifl.fr/modx).

Based on a simplified version of UML Use Cases, Figure 1 gives an overview of mechanisms which are proposed to designers to author their own concrete syntaxes. For a metamodel class, one can choose between one of the given shapes or an image, set the colors (background, border, text) and the size. One can also indicate if instances may embed other elements. For an association, one can choose between a line (and set its properties like color or style), an graphical inclusion relation (isNested or isNestedInText) or if linked elements will be on periphery (isJuxtaposed) like the dashed rectangle for UML template parameters. Concerning the edition of the concrete syntax, we have chosen simplicity (like TopCased [36]) rather than power (like Obeo Designer [26]): it is "easy" to define/parameterize a concrete syntax but possibilities are limited.

3.2 Metrics in ModX

One of ModX's main goal is to study how to allow a person, who has no skill in Computer Science, to design or to parameterize a software system through graphical diagrams. We have first proposed and implemented a method with which metamodel authors can define and associate modelling methodology to their metamodels (Incremental Modeling Process [18]). We plan now to inform them about the quality of their concrete syntaxes. So we propose:

- A programming interface (in Javascript) to access to abstract/concrete syntaxes defined in ModX and also to their models/diagrams;
- Two areas that are dedicated to the Javascript implementation of metrics related to abstract/concrete syntaxes and diagrams.

Two simple metrics samples are proposed for each previous area: the informational density for syntaxes metrics and the

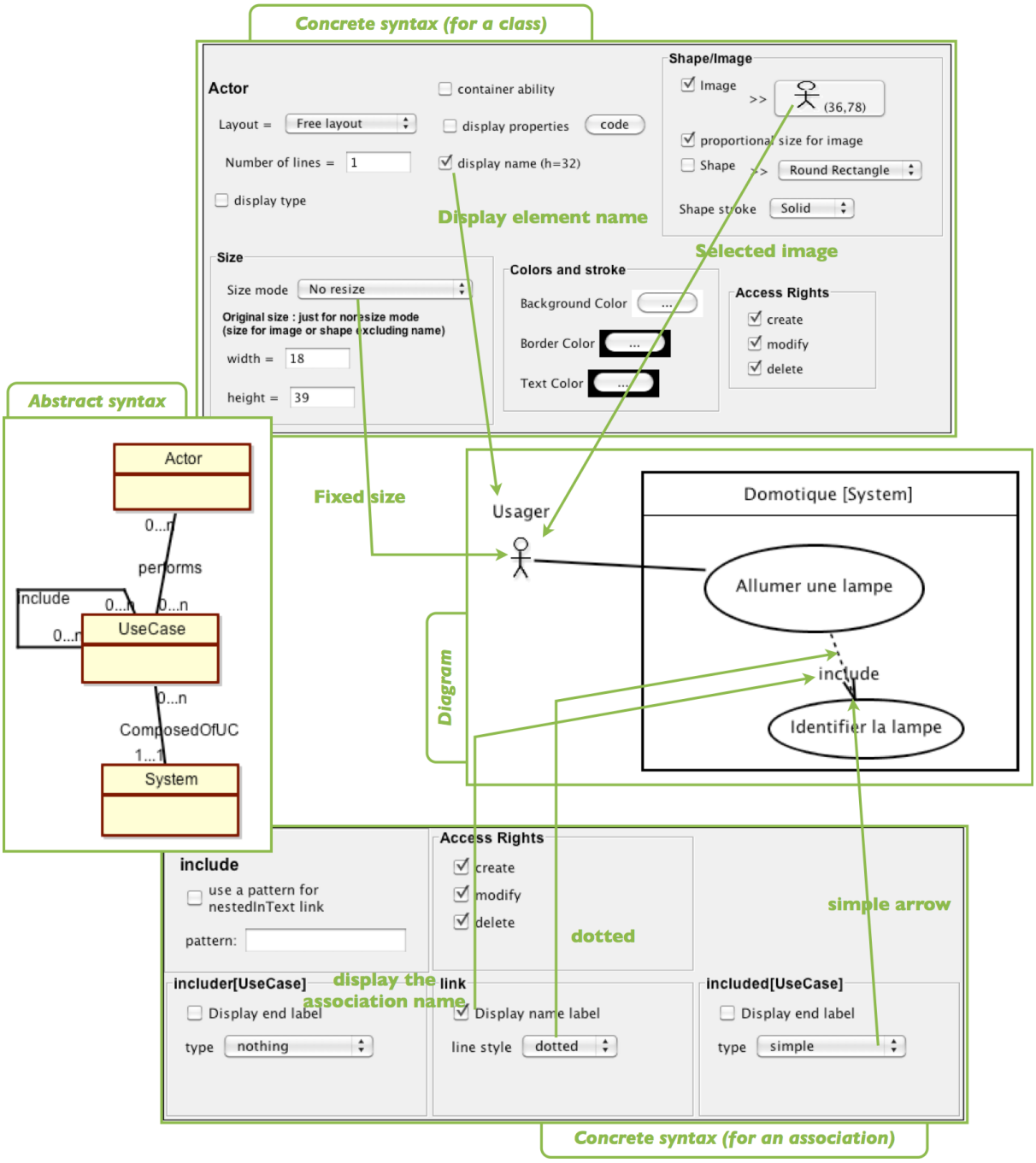


Figure 1: Abstract and Concrete Syntaxes and Diagrams in ModX

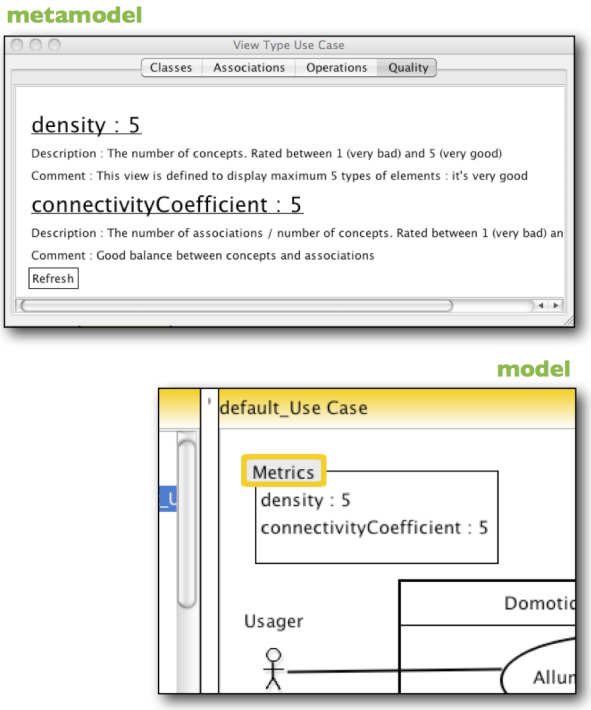


Figure 2: Display of Quality Metrics in ModX

connectivity coefficient for diagram metrics. If the algorithm is the same regarding to syntaxes and diagrams, the types of handled elements are different. The connectivity coefficient is related to the ratio between the number of relations and the number of elements.

Figure 2 illustrates metrics display. We only focus on display related to language because we are mainly interested by concrete syntaxes in this paper. The remainder of this section concerns these syntaxes and their properties. Metrics display is the execution result of scripts which are accessible at any time through areas we have previously mentioned.

Such scripts can start as follows:

```
var conStx_classes=concreteSyntax.
    __classes;
var conStx_associations=concreteSyntax.
    __associations;
for (var idx=0;idx<conStx_classes.length;
    idx++) {
  ...
}
addMetric ("my indicator", itsValue , "one
     comment", "one description");
```

3.3 Elementary functions related to syntaxes

The metrics implemented in ModX rely on the elementary functions that we have previously presented in section 2.3. We now draw a list of correspondence between these functions and the properties and functions that are present in MoX to show how these elementary functions can be implemented here. This list is presented in Table 3.3 with one column for classes and one for associations. Each property which is present in the ModX columns is an object property that is related to the visual representation of classes or associations.

The concrete syntax can be reached thanks to variable `concreteSyntax`. It gives access to an array of the concrete elements corresponding to classes (`__classes`) and associations (`__associations`). ModX uses the properties of these elements to draw the diagrams. For example, the `shape` property of the concrete element associated to UseCase class is set to `ELLIPSE`. This element also has a property `source` which refers to the UseClass class that has the properties `__name`, `__contents` (for attributes and references) or `__container`.

The visual variables (color, value, grain) are applied to the different parts of the concrete element. For example, the shape variable can refer to the geometric shape associated to a class or the shape that is used as a pattern to draw a border (or as a pattern of the border of a pattern). For concrete elements in ModX, there is no such recursion, and visual variables are limited to those that are proposed by ModX.

The listing 1 aims to determine if the visual distance is bigger enough between each pair of concrete elements. Note that the visual distance depends on its context of use, and for now, no scientific work has presented concrete empirical materials in order to define a formula for visual distance in Software Engineering. For this reason, the script is a simplified version used as a simple example. For each pair, the `computeAll` function will calculate the corresponding visual distance. If it is more than 1 - it means that there are at least two visual variables where elements have the same value - then the distance is considered as big enough. For classes, the distance may be calculated only if both elements use geometric shapes. If at least one element uses an image, then the distance is set to 2, meaning that the distance is also big enough. Otherwise, the function looks for differences concerning geometric shape, color, border, size, etc. For associations, the distance may be calculated only if both elements use lines. In this case, the functions looks for differences between the end line and the grain.

3.4 In other meta-case tools

Other meta-case tools (MCT) do not integrate metrics facilities for concrete syntaxes. But the amount of work to implement them vary according tools. First, a lot of meta-case tools only focus on textual notations or form-based ones like MPS [13], Spoofax [37], Whole platform[33] or Rascal[16]. So, we may assert that they are not candidate for concrete syntax metrics. Second, writing codes or rules to automatically calculate metrics requires an API to access to specifications of visual specifications. Such an API is rarely directly proposed. Eclipse-based tools like Obeo Designer [27] or Eugenia[8] do not propose it. However, as specification of concrete syntaxes are EMF models, it is possible to use Eclipse plug-ins like EOL[7] or QVTo[6] to programmatically navigate within them. But this requires to install additional plug-ins and the connection between them and previous models is generally not automatic. Besides, some functions like visualRepresentation or textualAnnotation will be not present and will require implementing higher-level function. MCT like MetaEdit+[22] or xOWL[38] provide a direct access to concrete syntax specifications. Third, it is better to propose a complete integration of metrics: a place to write script of rules about metrics that are expected and also a place where result of metrics calculation will be displayed,

	Properties in ModX	
Elementary functions	**Classes**	**Associations**
visualRepresentation	`concreteSyntax`	
visualRepresentations	untreated	
location	untreated	
size	width (`width`) height (`height`)	untreated
value	background (`backgroundBrightness`) border (`borderBrightness`) text (`textBrightness`)	untreated
grain	border (`borderType`)	stroke (`stroke`)
color	background (`backgroundColor`) border (`borderColor`) text (`textColor`)	
orientation	untreated	
shape	geometric shape (`shape`) image (`image`)	left/right ends (`left/rightEndStyle`)
complexityManagement	ability to contain (`containerAbility`)	type of link (`style`)
contextRepresentation	untreated	untreated
textualAnnotation	display name (`displayName`) display class name (`displayType`)	display association name (`displayType`)
preferredDrawingSupport	untreated	

Table 2: Elementary functions / ModX Properties

```
visualDistance = {
  between2classes : function ( element1, element2) {
    var difference = 0;
    if (element1.formMode==element2.formMode &&
      element1.formMode==concreteSyntax.SHAPE_MODE) {
    difference+=(element1.shape!=element2.shape ? 1 : 0 );
    difference+=(element1.backgroundColor!=element2.backgroundColor
        ? 1 : 0 );
    difference+=(element1.borderType!=element2.borderType ? 1 : 0 )
        ;
    if (element1.resizeMode==element2.resizeMode &&
      element1.resizeMode==concreteSyntax.NO_RESIZE)
      difference+=(element1.width!=element2.width ||
        element1.height!=element2.height ? 1 : 0 );
    } else difference = 2;
    return difference;
  },
  between2associations : function (element1, element2) {
    var difference=0;
    if (element1.style==element2.style && element1.style==
        concreteSyntax.LINK_MODE) {
      difference+=(element1.leftEndStyle!=element2.leftEndStyle ? 1
          : 0 );
      difference+=(element1.rightEndStyle!=element2.rightEndStyle ?
          1 : 0 );
      difference+=(element1.stroke!=element2.stroke ? 1 : 0 );
    } else difference = 2;
    return difference;
  },
  computeForAll : function (concreteSyntax) {
    // uses function addMetric (title, value, comments)
    // to display pair of elements which are visually too close
  }
}
```

Listing 1: Simplified function for visual distance

place well-know and easily access, in order to guide the design of concrete syntaxes. At the best of our knowledge, we did not see such integration in studied MCT. However, tools based on meta-programming like MetaEdit+ provides a better support for such integration rather that compiled MCT (like xOWL).

4. CONCLUSION AND FUTURE WORKS

In this paper, we present a metric-based approach to assess quality of modelling languages. The metrics are integrated in (meta-)modelling environment in order to adopt and to test them. We illustrate our approach with some quality indicators about concrete syntax, because few work focus on it. The originality is twofold: first, we propose an extensible environment to automatically calculate metrics on modelling languages; second, the metrics address an innovative domain i.e. the quality of concrete syntaxes.

If we show the feasibility of our approach, it remains to prove or to confirm its relevance with empirical studies. First, we plan to determine a more realistic measure of visual distance between elements of visual notation through significative experiments. We aim to define a formula that permits to grade the perceptual discriminability of visual notations. We will be able then to start experiments with the authors of visual notations and diagrams to assess if the presence of metrics is really useful.

5. REFERENCES

[1] J. Bertin. *Semiology of Graphics. Diagrams, Networks and Maps*. University of Wisconsin Press, 1983.

[2] P.-A. Caron, A. Derycke, and X. L. Pallec. Bricolage and model driven approach to design distant course. In G. Richards, editor, *Proceedings of World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education 2005*, pages 2856–2863, E-Learn 2005–World Conference on E-Learning in Corporate, Government, Healthcare, and Higher Education, October 2005. AACE.

[3] M. Cortes-Cornax, S. Dupuy-Chessa, D. Rieu, and M. Dumas. Evaluating choreographies in bpmn 2.0 using an extended quality framework. In *Proceedings o the 3rd International Workshop on the Business Process Model and Notation, BPMN 2011*, LNBIP. Springer-Verlag, 2011.

[4] B. Dobing and J. Parsons. How uml is used. *Commun. ACM*, 49(5):109–113, May 2006.

[5] E. Dubois, P. Gray, and L. Nigay. Asur++: A design notation for mobile mixed system. In *In: Mobile HCI '02: Proceedings of the 4th International Symposium on Mobile Human-Computer Interaction. ISBN: 3-540-44189-1*, pages 123–139, 2002.

[6] E. Foundation. Qvto : Qvt operational. *http://projects.eclipse.org/projects/modeling.mmt.qvt-oml*, 2008.

[7] E. Foundation. Eol : Epsilon object language. *http://www.eclipse.org/epsilon/doc/eol/*, 2012.

[8] E. Foundation. Eugenia : Gmf for mortals. *http://www.eclipse.org/epsilon/doc/eugenia/*, 2013.

[9] N. Genon, P. Heymans, and D. Amyot. Analysing the cognitive effectiveness of the bpmn 2.0 visual notation. In B. A. Malloy, S. Staab, and M. van den Brand, editors, *Software Language Eengineering, SLE'2010*, volume 6563 of *Lecture Notes in Computer Science*, pages 377–396. Springer, 2010.

[10] T. R. G. Green. Cognitive dimensions of notations. In *Proceedings of the fifth conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and computers V*, pages 443–460, New York, NY, USA, 1989. Cambridge University Press.

[11] O. M. Group. Meta-object facility. *http://www.omg.org/mof/*, 2012.

[12] K. Henricksen and J. Indulska. Modelling and using imperfect context information. pages 33–37, 2004.

[13] JetBrains. Meta programming system. *http://www.jetbrains.com/mps/*, 2013.

[14] Kaleidoscope. Network of excellence. *http://www.noe-kaleidoscope.org/*, 2012.

[15] S. Kelly and R. Pohjonen. Worst practices for domain-specific modeling. *Software, IEEE*, 26(4):22–29, 2009.

[16] P. Klint. Rascal : "rascal - meta programming language ". *http://www.rascal-mpl.org/*, 2010.

[17] J. Krogstie. Integrating the understanding of quality in requirements specification and conceptual modeling. *SIGSOFT Softw. Eng. Notes*, 23(1):86–91, Jan. 1998.

[18] X. Le Pallec, C. Filho, R. Marvie, M. Nebut, and J.-C. Tarby. Supporting generic methodologies to assist ims-ld modeling. In *Advanced Learning Technologies, 2006. Sixth International Conference on*, pages 923 –927, july 2006.

[19] O. Lindland, G. Sindre, and A. Solvberg. Understanding quality in conceptual modeling. *Software, IEEE*, 11(2):42 –49, march 1994.

[20] A. Maes and G. Poels. Evaluating quality of conceptual models based on user perceptions. In D. Embley, A. Olive, and S. Ram, editors, *Conceptual Modeling - ER 2006*, volume 4215 of *Lecture Notes in Computer Science*, pages 54–67. Springer Berlin / Heidelberg, 2006.

[21] K. Mehmood, S. Si-Said Cherfi, and I. Comyn-Wattiau. Data quality through model quality: a quality model for measuring and improving the understandability of conceptual models. In *Proceedings of the first international workshop on Model driven service engineering and data quality and security*, MoSE+DQS '09, pages 29–32, New York, NY, USA, 2009. ACM.

[22] MetaCase. Metaedit+ : Domain-specific modeling (dsm) environment. *http://www.metacase.com/products.html*, 2013.

[23] P. Mohagheghi and V. Dehlen. An overview of quality frameworks in model-driven engineering and observations on transformation quality, workshop on quality in modelling at modelsâĂŹ2007, nashville, usa, 2007, pp. 3-17, 2007.

[24] D. Moody. The physics of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Trans. Softw. Eng.*, 35(6):756–779, Nov. 2009.

[25] D. Moody, G. Sindre, T. Brasethvik, and A. Solvberg. Evaluating the quality of information models:

empirical testing of a conceptual model quality framework. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 295 – 305, may 2003.

[26] Obeo. Obeo designer. *http://www.obeo.fr/pages/obeo-designer/en*, 2013.

[27] Obeo. Obeo designer : Methods and tools for architects. *http://www.obeo.fr/pages/obeo-designer/en*, 2013.

[28] L. Palen and M. Salzman. Beyond the handset: designing for wireless communications usability. *ACM Trans. Comput.-Hum. Interact.*, 9(2):125–151, June 2002.

[29] X. L. Pallec, E. Renaux, and C. O. Moura. Modx - a graphical tool for mof metamodels. In *ECMDA-FA'2005 Tools Exhibition ECMDA-FA Open Source and Academic Tools*, 2005.

[30] M. Rossi and S. Brinkkemper. Complexity metrics for systems development methods and techniques. *Information Systems*, 21(2):209–227, 1996.

[31] J. Rouillard, J.-C. Tarby, X. Le Pallec, and R. Marvie. From Meta-modeling to Automatic Generation of Multimodal Interfaces for Ambient Computing. *International Journal On Advances in Software*, 3(3 & 4):318–332, 2010.

[32] N. T. Solheim I. Model quality in the context of model-driven develpment. *2nd Int. Workshop on Model-Driven Entreprise Information Systems (MDEIS 06)*, pages 27–35, 2006.

[33] R. Solmi. Whole platform. *http://whole.sourceforge.net/*, 2012.

[34] Someswar and Kesh. Evaluating the quality of entity relationship models. *Information and Software Technology*, 37(12):681 – 689, 1995.

[35] K. Sousa, J. Vanderdonckt, B. Henderson-Sellers, and C. Gonzalez-Perez. Evaluating a graphical notation for modelling software development methodologies. *Journal of Visual Languages & Computing*, 23(4):195 – 212, 2012.

[36] TopCased. The open-source toolkit for critical systems. *http://www.topcased.org/*, 2012.

[37] E. Visser. Spoofax: The spoofax language workbench. *http://strategoxt.org/Spoofax*, 2013.

[38] L. Wouters. xowl : xowl infrastucture. *http://xowl.codeplex.com/*, 2012.

Harmonizing Textual and Graphical Visualizations of Domain Specific Models

Colin Atkinson
University of Mannheim
Mannheim, Germany
atkinson@informatik.uni-mannheim.de

Ralph Gerbig
University of Mannheim
Mannheim, Germany
gerbig@informatik.uni-mannheim.de

ABSTRACT

Domain-specific models, and the modeling languages that support them, have played a central role in the success of model-driven development and its ability to bridge the abstraction gap between software implementation technologies and human developers. However, the current generation of domain-specific modeling tools is firmly split into two camps — those that support textual domain-specific languages and those that support graphical domain-specific languages. Both camps have enjoyed significant success, but at the time of writing no mainstream tool supports both at the same time. This stops developers from swapping freely between textual and graphical visualizations of a given subject of interest, even if each form has clear advantages and/or disadvantages for different stakeholders. In this paper we present an environment which offers a solution to this problem by strictly separating concerns for notation (i.e. concrete syntax) and concerns for concepts (i.e. abstract syntax) and allowing them to be connected and mixed dynamically as required.

Categories and Subject Descriptors

D.1.7 [**Programming Techniques**]: Visual Programming; D2.2 [**Design Tools and Techniques**]: Computer-aided software engineering (CASE); D.2.6 [**Software Engineering**]: Programming Environments—*Graphical environments*

General Terms

Languages

Keywords

Multi-level modeling, domain-specific languages, orthogonal classification architecture, symbiotic languages, linguistic classification, ontological classification, modeling languages, diagram, visual languages

GMLD '13, Montpellier, France

http://dx.doi.org/10.1145/2489820.2489821.

1. INTRODUCTION

Domain-specific modeling has grown in importance over the last few years, and domain-specific modeling tools are now routinely used in industry. However, the market for domain-specific modeling tools has long been split into two groups — one group supporting graphical domain-specific modeling and one group supporting textual domain-specific modeling. Well known examples from the first group are MetaEdit+ [30], Graphical Modeling Environment (GMF) [14], Generic Modeling Environment (GME) [22], Poseidon for DSLs [13], Microsoft Visual Studio for DSLs [23] and ATOM3 [21], while well known examples of the second group are EMFText [15], XText [16], Spoofax [18], MetaDepth [20] and JetBrains MPS [17]. To our knowledge the intersection of these two groups is currently empty. In other words, there is no mainstream domain-specific modeling tool available today which supports both graphical and textual domain-specific modeling out-of-the-box.

For small, focused problems where the goal of the domain-specific modeling activity is to meet a very specific need of a single type of stakeholder this is usually not a problem. However, for larger, less-well-defined projects with multiple types of stakeholders this mutual-exclusion of textual and graphical visualizations of information can become very restrictive. The problem of mixing textual and graphical visualizations in a single editor has consequently received a lot of attention in recent years ([10], [12], [25], [27], [28]). When using state-of-the art commercial tools to develop mixed textual and graphical editors developers have to resort to workarounds that provide some way of transcending the different technology spaces and map graphically-viewable information into textually-viewable information. In the best case, the text-oriented and graphical-oriented tools are based on the same underlying infrastructure (e.g. XText and GMF are both based on the Eclipse Modeling Framework (EMF) [29] and the Eclipse Platform) and it is sufficient to develop glue code allowing the two editors to be used in tandem. In the worst case, when there is no common infrastructure behind the tools, a full suite of "model-to-text" and "text-to-model" transformations has to be defined (and subsequently maintained) to provide a full and coherent mapping of all the concepts in the languages driving the different tools. In all cases, supporting interoperability between the two kinds of visualization approaches is a tedious and often ad hoc task that invariably introduces accidental complexity in software engineering projects.

There are three fundamental roots to this problem. The first is the reliance of the majority of textual domain-specific

modeling tools on traditional parsing technology, based on context free grammar definitions, to define the structure of statements in textual languages, and the convention of entwining the definition of the abstract syntax with the definition of the concrete syntax. Thus, the majority of tools supporting textual domain-specific modeling languages work on the assumption that a dedicated abstract syntax is created to support the required concrete syntax, and if a different concrete syntax is required, a new dedicated abstract syntax is created for it. It is because this abstract syntax is concrete-syntax-oriented, and not problem domain oriented, that the underlying representation models for graphics-oriented tools and text-oriented tools are incompatible. Exceptions are tools like EMFText which try to solve the problem by annotating meta-models with information to generate parsers and textual editors. This means that languages must be built in such a way that they can be supported by the class of parsers supported by the tool with the consequence that the abstract syntax is influenced by the concrete syntax.

The second problem arises when using graphical and textual editors simultaneously while editing a model. Textual model editors based on parsers always create a new abstract syntax tree based on the currently parsed text. This model must then be merged with the model on which the graphical editor works. The merging problem is not trivial and cannot be solved in all cases, for example when identification information between the textual and graphical representation gets lost in the textual representation of a model. Such a loss leads to glitches when using graphical modeling editors and parser based textual modeling editors side-by-side.

The third problem is the reliance of all existing domain-specific modeling tools on a two-level physical modeling architecture to support classification. In other words, all editors in all the tools, no matter whether textual or graphical, basically support the creation and manipulation of instances of one fixed set of types. In other words, the types are hardwired into the code driving the editor while the instances of the types are "soft" (i.e. data). The consequence is that when the current generation of domain-specific modeling tools wants to "deploy" an additional layer in a domain-specific modeling language whose concepts are initially "soft" (and thus uninstantiable) they have to perform a major compilation and deployment operation to effectively create a new editor in which these formerly soft concepts are then hardwired. Again this introduces a lot of unnecessary accidental complexity and rigidity into the modeling process when employing more than one pair of classification levels in a domain model.

A fundamental solution to the problem of harmonizing textual and graphical visualizations of domain specific models must address all three of these problems. In this paper we introduce an approach for achieving this goal which addresses the first problem by introducing the notion of visualizers to drive all visualization processes and addresses the second problem by employing textual model editors based on projection technology of the kind supported by JetBrains MPS. This frees the tool from having to worry about the underlying parser generator's restrictions when defining the internal representation format. Furthermore, synchronization with a graphical editor is not based on any form of model merging because models are directly edited without the need for a generated parser. In other words, it makes this possible for text editors to utilize the same underlying model as graphics-oriented tools. However, the only mainstream tool currently supporting projectional editing for text, JetBrains MPS, only focuses on textual notations. The third problem is addressed by using a multi-level model environment to make all model elements and visualizers, at whatever level of classification, "soft" and thus amenable to change and manipulation at any time. By taking such an approach, the presented environment provides a uniform mechanism for supporting all possible forms of visualization, including tabular and wiki oriented forms of visualization, as well as textual and graphical forms. An additional benefit of the use of a multi-level modeling approach is that the DSLs can have a symbiotic relationship with the general-purpose notation used in the tool. This means that any of the domain-specific visualizations of a model-element is interchangeable with the general-purpose visualization at any time, at the touch of a button [2].

The remainder of this paper is structured as follows: in Section 2 multi-level modeling is introduced followed by a description of the domain-specific capabilities of multi-level modeling in the area of graphical and textual domain-specific languages (DSL) and symbiotic language support in section 3. Then, section 4 applies the approach to a small case study for modeling organizational structures. This language allows an organizational structure to be modeled in both a graphical and textual domain-specific language. The work closes with related work in section 5 and conclusions in Section 6.

2. MULTI-LEVEL MODELING

Multi-level modeling allows modelers to model a domain spanning an unlimited number of classification levels without resorting to awkward workarounds (e.g. UML Power Types or Profiles). This is made possible by the orthogonal classification architecture [4] which strictly separates ontological and linguistic classification into two orthogonal dimensions. The linguistic classification dimension represents the traditional model stack from a tool's point of view and describes how domain model-elements (L_1) representing domain concepts (L_0) are classified by the abstract syntax constructs of a single, ontological-level-spanning modeling language (L_2). The ontological classification dimension is a stack of ontological levels (a.k.a models) contained within, and orthogonal to, L_1. The sum of all ontological levels is called an ontology. Additionally, the concept of deep characterization is applied by giving each model element a linguistic attribute (a.k.a trait) called potency. This states how many levels the instantiation tree of a model-element can span and thus a model-element can influence. The potency of a model-element is either a non negative value or "*". Instances of a model-element have a potency one lower than the potency of their type. If the type has "*" potency, the instances can have an arbitrary value as their potency including "*". Model-elements with a potency of "0" cannot have instances. A potency is also attached to attributes (a.k.a durability) and their values (a.k.a mutability). The durability describes over how many instantiation steps an attribute can be handed over to instances and mutability over how many instantiation steps the value can be changed. In a modeling architecture featuring multiple levels it can happen that a model-element residing at the middle ontological levels is at the same time an instance of a type and a type for an instance. The term clabject is therefore introduced

to reflect the type (class) / instance (object) duality of an ontological model-element.

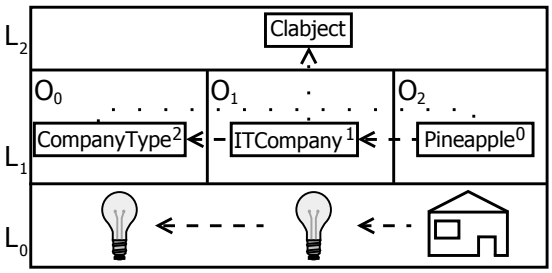


Figure 1: The orthogonal classification architecture.

Figure 1 shows an example of the orthogonal classification architecture and the concept of deep characterization. Vertically, a fixed number of three levels (L_2 - L_0) exist describing the multi-level modeling language at L_2, the ontology at L_1 and the real world at L_0. The ontological levels horizontally contained in L_1 show an example in the domain of an organization modeling language. The example spans only three ontological levels (O_0 - O_2) for space reasons. All three ontological model-elements (CompanyType, ITCompany, Pineapple) are instance of the linguistic metamodel-element Clabject (dotted vertical arrows) stating that these model-elements can be types and instances at the same time. In this example only ITCompany is simultaneously a type and an instance because it is the only model-element on a level that is both classified by a level and classifies another level. Ontological instantiation is indicated through horizontal dashed arrows. Each model-element has a potency as superscript next to its name. CompanyType for example has potency 2. The instances of CompanyType, here ITCompany, have a potency that is one lower than the potency of their type, CompanyType, resulting in a potency of one. At the lowest level an ITCompany named Pineapple resides with a potency of zero because it is created from CompanyType through two instantiation steps each lowering the potency by one. The rendering shown here uses the Level-agnostic modeling language (LML), the general-purpose language for rendering multi-level models. A modeler, however, can also define domain-specific renderings for model-elements as described in the following section.

3. SYSTEMATICALLY SEPARATING ABSTRACT AND CONCRETE SYNTAX

At the present time our multi-level modeling environment, MelanEE [1], supports graphical and textual visualizations of domain-specific languages but tabular and wiki-oriented visualizations are also planned in the future. All visualization forms are realized in a uniform way using the concept of visualizers. These allow the representation of model-elements to be defined and updated in a highly flexible way, allowing the rapid prototyping and changing of domain-specific languages. In this section we introduce the concepts of visualizers and explain how they are realized in MelanEE.

3.1 The Visualizer Concept

As its name implies, a visualizer determines how a model-element is visually represented. Two general kinds of visualizers are currently supported — Level-agnostic Modeling Language (LML) visualizers and DSL Visualizers. LML visualizers define the general-purpose visualization of clabjects and their instances / subclasses in the LML. Their attributes store all the values needed to define how the visualization symbol should appear, such as how each trait is displayed. The default visualization, as defined by the LML, is indicated by the value "default". For example, by default the potency of an attribute (a.k.a. durability) is not shown if it has the same value as that of its clabject. This default behavior can be overridden by changing the value of the attribute to "noshow", "show" or "tvs". The first of these values, "noshow", indicates that the value should never be shown no matter in which circumstances. This value is often used to hide the level a clabject is located at, because this is unimportant information in most cases. The second value "show" specifies that a trait should always be displayed. The durability and mutability of an attribute are often set to "show" in educational models to make their values explicit. The last value "tvs" is used to display the value in a special location under a clabject's designator (containing besides other things the name of a clabject), the trait value specification. In addition to these attributes, a visualizer contains information about what should be displayed in a clabject's designator and whether a domain-specific rendering is desired. The designator trait of a visualizer indicates what to display in the header compartment of the clabject's general-purpose rendering — the LML —, including whether to display information about its heritage, classification and location. More information on the LML can be found in [5].

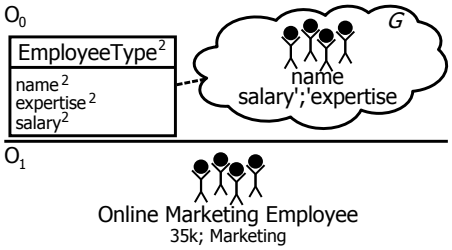


Figure 2: The definition of a graphical domain-specific rendering.

Domain-specific visualizers can be used to configure domain-specific visualizations of clabjects and their instances / subclasses. More specifically in case of a graphical visualization, they allow various kinds of graphical symbols to be associated with clabjects based on the concepts of layouts, geometric shapes and labels — an approach to graphical modeling language definition that has been widely implemented (e.g. MetaEdit+, GMF-Tooling, OMG Diagram Definition [24]). The basic idea behind this approach is shown in Figure 2 which shows a small excerpt of a more complete DSL definition shown in Figure 7. The example focuses on the definition of a graphical notation for one model-element, EmployeeType, which has three attributes, name, expertise and salary. In the domain-specific visualization these three attributes should be shown underneath a graphical symbol resembling the notion of EmployeeType. In the figure, the attachment of a symbol to a clabject is represented by a cloud, but in a tool such as MelanEE this is achieved by creating a visualizer model-element. Here, a scalable vector graphics (SVG) figure is placed in the first row of a one column table layout. The label mappings for the attributes

to be displayed are placed in the second row of the table layout.

Since there can be several visualizers attached to a clabject, a user can decide on-the-fly whether to work with the general-purpose or one of the domain-specific notations of a model-element. This can be extremely useful to domain novices in cases where different metaphors in a modeling language are quite similar to each other. We refer to such a relationship between a domain-specific and a general-purpose language (i.e. where the symbols defined by one can be intermingled on a clabject-by-clabject basis with the different symbols defined by another) as a symbiotic relationship.

Textual domain-specific languages are defined using the concept of visualizers, too. In our multi-level modeling tool, MelanEE, only a simple language is need for this purpose. This is possible because most of the structural information such as multiplicities etc. is defined in the model itself and not the concrete syntax definition. On the contrary, in EBNF-focused textual domain-specific language tools such as XText the concrete and abstract syntax definitions are mixed together. MelanEE implements textual domain specific language definitions in a similar way to EMFText which annotates a meta-model with information for textual rendering. The concept of visualizers is similar to the concept of annotating an Ecore model with textual rendering information. For each model-element a visualizer containing the definition of the text snippet for rendering is attached. From this information an editor for editing the model in a textual language is generated.

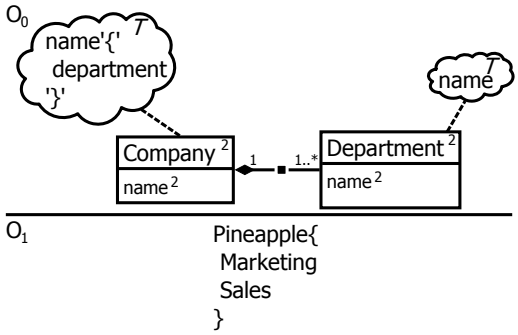


Figure 3: The definition of a textual domain-specific visualization.

The language used for defining textual syntax revolves around two concepts and has thus a very low complexity. One is the definition of parts of the textual language which cannot be edited, the other part is a mapping of text to attributes and references in the model which are editable in the resulting editor. An example of a textual language definition is displayed in Figure 3 which again is an excerpt of the organization modeling language in full detail shown in Figure 7. This language definition excerpt defines two model-elements, Company and Department. A textual visualizer represented as a cloud is attached to each one of these concepts. The visualizer of Company states that first the name of a company is displayed followed by its Departments enclosed in curly brackets, while the visualizer of Department states that a department is visualized by the value of its name attribute. O_1 shows an instance of the modeling language represented by the instances' textual notation. A company called Pineapple has been instantiated containing two departments called Marketing and Sales.

3.2 The Visualization Search Algorithm

The idea of annotating model-elements with rendering information used in an editor, and applying a search algorithm to find what symbols to render is not new to meta-modeling. For instance in [11] Ecore meta-models are annotated with information for textual visualization and a search algorithm is applied to find the most appropriate visualization information for a model-element. That algorithm, however, does not cover multi-level modeling scenarios. A multi-level visualization search algorithm such as that described in [4] (and steadily refined since then) searches up a clabject's complete stack of classifying levels as well as its own level. In general the visualization search algorithm first searches in the inheritance hierarchy of a clabject for rendering information. If no visualization information is found the next level in the classification hierarchy is searched and so on until all the classification levels above that of the clabject to be visualized are searched through. This is done recursively over all classifying levels until a result is found. If the algorithm ends without finding visualization information at the ontological levels, the visualization of the model-element's linguistic type, which is the LML rendering, is used. In fact, because of MelanEE's support for symbiotic languages (i.e. toggling the rendering of a clabject between DSL or GPL notation at will) a modeler can switch off this visualization search algorithm at any time, for any individual model-element, and immediately use the GPL visualization. The visualization search algorithm is not limited to graphical visualizers, but applies to any kind of visualizer including textual and tabular visualizers. For illustration reasons the application of the algorithm is only shown for graphical visualizations in Figure 4. However,it works in the same way for textual and tabular visualizations etc.

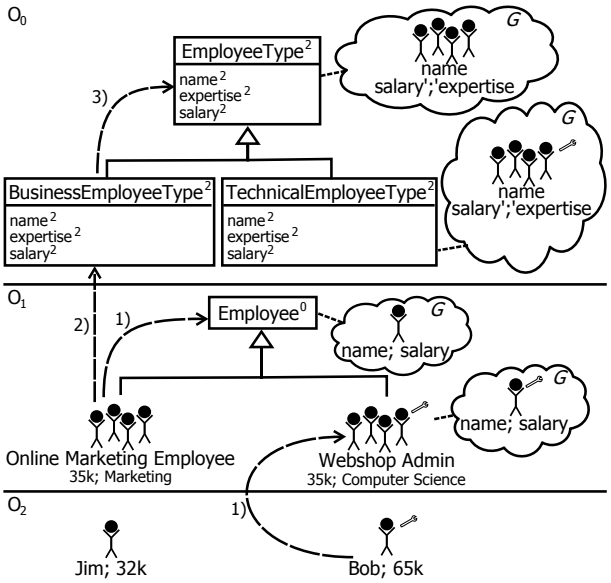


Figure 4: The visualizer search algorithm. Dashed lines indicate search order of algorithm.

Figure 4 shows the application of the visualization search algorithm in the context of the company modeling language introduced previously. EmployeeType, residing on level O_0,

is modeled with a default graphical visualization representing a group of employees. This default visualization is overridden by TechnicalEmployeeType which shows a wrench in addition to the group of employees. BusinessEmployeeType and its instances on the contrary are represented by the visualization defined for EmployeeType. On level O_1 the EmployeeTypes are instantiated by different kinds of employees. An abstract base class, Employee, which has a default employee metaphor attached, is introduced as root model-element. To prevent its subtypes to use this visualization information instead of their type's it is marked as to instances applicable only. This not indicated in the figure for overview reasons. The attached visualizer accesses two attributes which are not present in the Employee model-element. This is allowed by the visualization algorithm as non existing mappings are simply ignored during run-time offering a high degree of flexibility when defining visualizers within an inheritance hierarchy. Online Marketing Employee is instantiated as instance of BusinessEmployeeType and Webshop Admin is instantiated from TechnicalEmployeeType. The visualization search algorithm is applied to find a suitable visualization for each model-element at O_1. Employee does neither have supertypes nor ontological types, but has a graphical visualizer attached to itself. Thus its own visualizer which is found by the search algorithm can be used for visualization. In the example, however, it was decided to use the GPL visualization for the model-element and not to apply the result of the visualization search algorithm. Online Marketing Employee and Webshop Admin are both rendered using their domain-specific notation. Both do have a common supertype, Employee, which stores a graphical visualizer. The visualizer is marked as only applicable to instances which is omitted in the figure. Thus no visualization information is found in the inheritance hierarchy. Hence, the visualizer search algorithm continues searching at the next ontological type level. In case of Webshop Admin a graphical DSL visualizer is found at its ontological type, TechnicalEmployeeType. A visualizer for Online Marketing Employee cannot be found at its ontological type, BusinessEmployeeType. Hence, the inheritance hierarchy of BusinessEmployeeType is searched which results in a graphical visualizer found at EmployeeType. Level O_2 contains two employees, Jim an Online Marketing Employee and Bob a Webshop Admin. For both the visualizer search algorithm is applied in the same way as described previously for the Online Marketing Employee and Webshop Admin which results in a visualization for Bob gained through its ontological type Webshop Admin and a visual representation for Jim gained through the supertype of its ontological type Online Marketing Employee namely Employee.

3.3 Graphical DSL Editing

Graphical domain-specific modeling visualizations are supported in the same model editor as general-purpose visualizations without the need for any recompilations or redeployment. A modeler can change between a clabject's general-purpose or domain-specific rendering at the click of a mouse. All types available for instantiation in a domain-specific language are displayed alongside the general-purpose language elements in a tool palette on the right hand side of the editor. The properties view at the bottom displays the attributes to be edited of the currently selected model-element. A modeler can choose to filter the information displayed while using a domain-specific language by for example hiding all ontological levels except the one he is working with and hiding all the general-purpose language modeling tools from the palette. The properties view also allows a domain-specific view on model-elements which only displays ontological attributes and hides linguistic attributes which are irrelevant when using a domain-specific language. Through these mechanisms a modeler can customize the degree of domain-specificity of features offered when using the tool. A user more confident within a domain can work fully with the domain-specific features, whereas a language novice can choose to include general-purpose features as well.

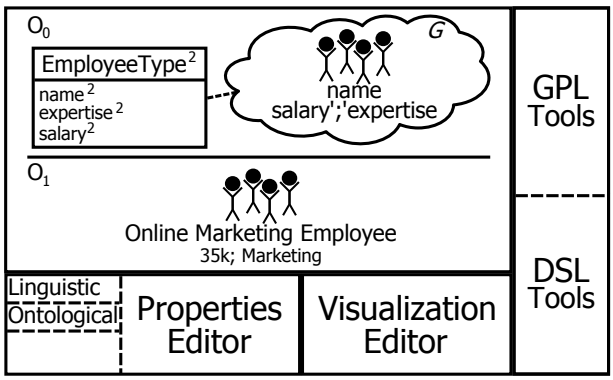


Figure 5: Mock-up of the graphical DSL editor.

A mockup illustration of the MelanEE editor is shown in Figure 5. The main modeling space is positioned at the central, left part of the screen consuming the most space. The editing area shows two levels, O_0 and O_1. A user can, however, decide which levels to work with as mentioned earlier. At the left bottom the properties view, with two tabs on its left hand side, is located. This view is used to change the values of linguistic traits and ontological attributes. The first tab contains the Linguistic properties page which displays linguistic traits of the currently selected model-element, such as the potency, and can be used to change these values. The second tab contains the Ontological properties page which displays the ontological attributes of the currently selected model-element and is thus most frequently used in the DSL view. It is needed because in most DSLs a user cannot select an ontological attribute within a model-element's symbol and change its linguistic value trait. To do so all ontological attributes of the currently selected model-element and their value traits are made available for editing in this view. Next to the properties view, on its right side, a view for editing the visualization information of the currently selected model-element is placed. This view does support the definition for graphical, textual etc. editors based on the concept of modeling visualizers which are available for these different view kinds. The right side of the environment shows the tool palette which is divided into two parts, the GPL Tools and DSL Tools part. The GPL Tools palette defines the general LML types available for instantiation, while the DSL Tools palette offers the ontological model-elements available for instantiation. This palette responds in a context sensitive way. After a user has selected the ontological level / container into which one wishes to place a model-element, the DSL Tools palette offers all the ontological types that can be instantiated at this place.

3.4 Textual DSL Editing

Two basic ways of editing models using a textual concrete syntax exist — one based on parsers and the other based on projection. The difference between the two is how they build up and edit the model representation (i.e. abstract syntax tree) of the edited text. The most wide spread approach, used by technologies like XText and EMFText, uses a parser driven by some kind of context free grammar specification (e.g. defined in an EBNF dialect). This approach, however, has a few disadvantages. A general problem, not related to the combination with model-driven technologies, is that a parser generated from a grammar is always limited to the parser class supported by the parser generators used in the employed language workbench. This requires a language engineer to know which parser class is required to parse a certain language before choosing a tool to implement a textual domain-specific language. In addition, when designing a domain-specific language, an engineer has to keep in mind the capabilities of the underlying parser generator. In some cases it can happen that constructs in a grammar need to be expressed in a certain way to work with a generated parser while other ways of expressing the same language construct do not work.

Also problems arise when editing models through text which are used in more than one editor at the same time (e.g. in a graphical and textual model editor). Graphical editors for example often store meta-information about the layout of a model which needs to be preserved while editing a model. Here problems occur when saving the content generated by a parser. Parsers cause problems because they always create a new model (i.e. abstract syntax tree) out of unstructured data (i.e. plain text). This model then needs to be combined with an existing model in order to keep meta-information like IDs, layout information etc. defined in the edited model.

Merging algorithms are often applied to merge models generated by a parser with the existing edited model to preserve in it defined meta-information, such as layout information. Such merging algorithms rely on identification information which must be present in both the textual model representation and the edited model itself. In cases in which this information gets corrupted, the merging algorithm fails to merge the models. This failure is compensated for by removing the model elements from the edited model which have no matching identification information in the edited model and the textual representation. Afterwards, a new model element is created for those model elements present in the textual syntax which do not have a counterpart in the edited model anymore. In this way unmatched model elements in the edited model are replaced with new model elements containing the same information as the previously deleted ones. This behavior leads to a loss of meta-information (e.g. for layout) about model elements because for the newly created model elements this information is not present. Losing layout meta-information in a graphical editor often leads to glitches like model elements getting repositioned at an arbitrary position after transferring a change from the textual model editor to the edited model. A user then needs to manually layout the corrupted layout of the model. An additional problem when editing a model through text arises with statements which do not exactly conform to a language's textual syntax definition. In these cases it can happen that a textual model editor either prohibits the saving of a textual model or starts deleting the elements (and all of their children) that do not conform to the concrete syntax. This again can cause a loss of information in an edited model. Strictly speaking, when a user changes the keyword "class" to "clas" in java, for example, the whole class including its content should be deleted from the saved model. The approach of prohibiting a save in cases of invalid concrete syntax is for example used by the Eclipse Project's OCLInEcore editor [9], which allows Ecore models to be edited using a textual editor. The textual editor prevents the user from saving the diagram until the textual representation of the model is in full conformance with the textual syntax definition. This is, however, not very practical because it prevents a user from saving his / her work while editing a model. Moreover after each save the parsed text and the model are merged. If an ID gets corrupted while editing the model's textual representation it can happen that no model-element in the edited Ecore model can be found for the text snippet currently edited. This causes the old model-element be deleted and a new one to be created. As mentioned earlier, this is not a problem as long as this model is not edited with a second editor that stores meta-information for a model-element such as a graphical model editor. Otherwise the previously described glitches when editing the model in the graphical and textual model editor side-by-side can occur in this case.

The second way to implement a textual model editor is to use projection. An example of a textual textual language workbench based on projection is JetBrains MPS. Projection provides a continuous connection between the model and the text representing the model. Text is no longer parsed but changes are directly written into the model. This direct editing of the abstract syntax tree makes parsers obsolete and frees languages from the constraints of the parser classes that can be generated by a certain textual language engineering tool. Also the model merging step is not required and thus, no meta-information (e.g. layout) is lost in cases where re-parsing would cause a loss of such information. The drawback of such an approach is that it cannot deal with unstructured data "out of the box". In contrast to a parser based editor one cannot paste text into a projectional editor because it is not capable of transforming this text into a model. This weakness could be reduced by combing a projectional editor with a generated parser that allows existing text to be imported. Such an import mechanism is, however, again limited to the power of the parser generator. The textual multi-level model editor in MelanEE is based on a projectional approach.

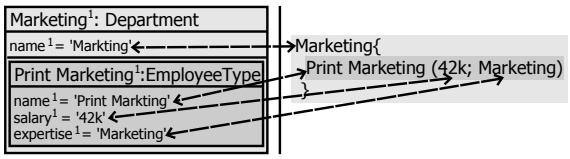


Figure 6: Textual model editor and weaving links pointing to abstract syntax elements.

Figure 6 shows how projectional editing is realized in our Multi-level modeling tool, MelanEE. The left hand side shows the general-purpose graphical visualization of a model-element in LML and the right hand side shows a textual representation of the same information. Strictly speaking the left

side is a representation of the model's abstract syntax in memory which could also be displayed as a graphical model side-by-side with the textual editor as indicated in the figure. The dashed arrows between the text and the model indicate the connections that are established and maintained by the projectional text editor. The connections are realized by a bi-directional weaving model that connects the model and its textual representation. This weaving model looks very much like the concrete syntax model of the text displayed in the text editor with one exception. This exception is that the concrete syntax model is expanded with weaving links to the edited model. If a user changes the text in the text editor, the weaving model (the concrete syntax model) is immediately changed. This model is monitored for changes by the model stored in memory which, on the event of a change, updates its model-elements which are connected to the changed concrete syntax model elements. The same mechanism also works in the other direction. As soon as a modeler changes the model in the memory (e.g. through a graphical editor) the concrete syntax model of the textual representation is updated which in turn updates the text which is represented by this model. Through this mechanism it is always guaranteed that the multi-level model and its textual representation are synchronized.

The model editor ensures that a user can only make changes which correspond to the textual concrete syntax definition of the model. This prevents situations in which a user can not save a model as this would cause an inconsistency between concrete syntax and edited text and would thus cause an accidental loss of information during a save operation. To achieve the target of only allowing textual representations which conform to the textual syntax definition of a model the idea of syntax directed editing is used to realize all aspects of the model editor. Keywords are marked with a grey background when selected and cannot be changed. Only delete operations are allowed on keywords. If they are optional they can be deleted without any effect on the model. In cases where they are not optional the user is asked whether to delete the whole model element and all model elements it contains. Literals which only consist of white-spaces can be extended by white-space or parts of them can be deleted for formatting reasons. New model elements can only be created by invoking the context sensitive and language driven features of the editor. Attributes can be edited like usual text. Validation errors introduced through editing are displayed as editor annotations in the textual editor. These errors cannot only be fixed in the textual editor but also in others, such as graphical editors, which then fix the error in the textual view by remaining synchronized with it.

4. MULTI DOMAIN-SPECIFIC VISUALIZATION MODELING

In the previous section the visualization and editing mechanisms for textual and graphical domain-specific multi-level model editing have been presented in isolation. The unique characteristic of MelanEE's domain-specific language support is that the defined editors can also be used together in an integrated manner to facilitate the viewing and editing of a model in multiple visualization forms (e.g. graphical and textual) at the same time. To do so neither separate tools using different technologies nor glue code that connects two modeling technologies within one environment needs to be developed. A modeler can use different visualization forms side-by-side by simply defining visualizers for the visualization form to be used and open the model in different editors at the same time. Figure 7 shows the editing of a model using two different domain-specific language visualization forms in the context of the earlier partly introduced organization structure modeling DSL. This example shows one model being edited using two separate editors for the same model. It is also planned to support the mixing of visualization forms within editors. A user can for example host a textual or tree based editor within a graphical editor to improve the overall user experience. An example where hosting a textual editor within a graphical one is a significant advantage is found in the domain of UML Class diagrams. These diagrams can use graphics to display classes and a textual editor hosted within the graphical editor to offer convenient attribute editing with features like coding assistance.

The top of Figure 7, O_0, shows the language's meta-model with the definition of the textual and graphical concrete syntax defined side-by-side. To indicate the definition of a DSL visualizer the notation of a cloud attached to the model-element for which the concrete syntax is defined is used. The graphical definition is indicated by a *G* and the textual definition by a *T* in the upper right of the cloud representing the visualizer. The language defined at the top provides the possibility to model Companies consisting of Departments which again consist of different EmployeeTypes. A Company is represented by the visual metaphor of a house containing Departments represented by dashed rectangles containing EmployeeTypes rendered through a group of stick men. From a textual view point a Company is rendered by first printing its name, followed by its Departments in curly brackets which are rendered by their name and contain their different types of employees in curly brackets. These different EmployeeTypes are visualized by stating their name followed by their salary and expertise separated by a semi-colon and enclosed in round brackets. An instantiation of this organization modeling DSL with a fictive IT company called Pineapple is shown at the bottom level of Figure 7. The company consists of two departments, a Marketing and Sales department. These departments employ different types of employees, with different salaries and different skills, e.g. Webshop Admins with a salary of 45k and skills in Computer Science. The model is shown in a graphical DSL (left side) and textual DSL (right side) side-by-side in the figure. A language user, however, has different options when editing this model — a) only showing the graphical visualization of the DSL, b) only showing the textual visualization of the DSL and c) showing both graphical and textual forms side-by-side. The option shown in the figure is the concurrent editing of the model in its textual and graphical notation. A user can create the content using the projectional, graphical layout preserving, textual DSL editor as this is assumed to be faster than using a graphical editor. On the other hand the user can at the same time manipulate the model from a graphical point of view, for example changing the layout of the graphical diagram, without the need to switch modeling tools. This enables a user to exploit the strengths of multiple domain-specific visualization forms, here textual and graphical, at same time. In addition, one could also get offered a third view on the same model such as a tree based model editor focusing on the containment hierarchy of the model. Such a view again integrates seamlessly into

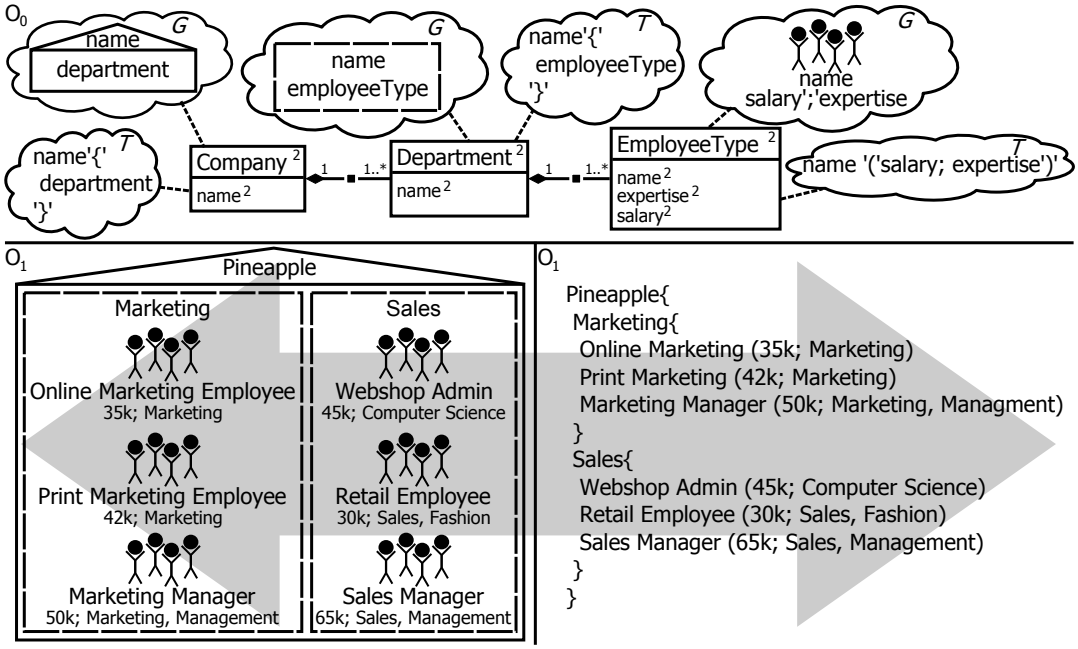


Figure 7: Simultaneous graphical and textual editing of a model.

the modeling language editing experience. One could even imagine a third level which can contain instances of the types modeling a company in details (e.g. having different Marketing departments with different employees in different roles such as Marketing Manager). It would also be possible to define a second textual interchange syntax on the same model which can be used for exchanging content modeled using the graphical and textual multi-level DSL with other tools.

5. RELATED WORK

A lot of work as been done in the area of combining textual and graphical languages with each other. The work discussed here mostly focuses on combining two tools with each other. Some focus on integrating XText with GMF based model editors by writing glue code between the two by hand. These scenarios support viewing a whole model in code and graphical notation side-by-side or embedding textual editors within a graphical one gluing together two different technical stacks (e.g. XText and GMF in [12]) and synchronizing the textual and graphical representations on save operations. There are also approaches that work on two separate models and apply a merge algorithm at run-time independent of load/save mechanisms and synchronize views based on these separate models [28]. Other work describes a framework for augmenting a graphical editor with context menus for editing parts of a model using a textual view [27]. All of these techniques however involve the application of multiple technologies to define a single modeling environment. No uniform technology and modeling concepts are used to create both a graphical and textual editor. Furthermore, the first two technologies require complex glue code to be written in order to connect a graphical and textual editor. The integration of textual and graphical languages has also been implemented in some tools e.g. UML Activity Modeling [10] or in the context of the KIELER tool [28]. In [25] Atom^3 is extended to include textual views on an underlying model. The textual views are defined on a metamodel and grammar based parsers are generated to parse the textual model instance and write it back to the underlying model of the whole system. The authors, however, make no statement about the editing experience when working on the same part of a model using a graphical and textual notation at the same time. They do mention that the approach currently has some shortcomings such as the fact that the only direction that is currently supported is text-to-model.

In the area of multi-level modeling not much work has been done on offering graphical and textual editors side by side on a model. MetaDepth [8], a multi-level modeling tool based on textual syntax, makes it possible to view a model in a read-only graphical view. This view does not support domain-specific notations which can be defined by a language engineer. Domain-specific textual languages can be created by using this tool [20].

The work here is not related to view-based software modeling methods which differ in the sense that a whole system is modeled using views. These views often do not overlap each other or even describe the same aspects of the underlying system using two or more different visual notations. Examples of such view based modeling approaches are enterprise modeling frameworks like Archimate [19] or ARIS [26] and their supporting tools. View based modeling is also applied in software engineering for the purpose of managing the complexity of refactorings [7] or developing component based software systems [6]. In contrast the work presented here focuses on the definition and usage of multiple concrete syntax (e.g. graphical and textual) in an integrated and uniform way for one single model and one single purpose. Hence, the technology here would be more suitable to define views of a view based tool than realizing the tool with all its transformations etc. itself. In [3] MelanEE's domain-specific modeling capabilities are used to realized the views of such a view based modeling tool.

6. CONCLUSIONS

In this paper we identified the current fragmentation of domain-specific modeling tools into two groups — one focusing on the textual visualization of models and the other on the graphical visualization of models, and have explained why this is suboptimal. We then explained the underlying reasons for this problem and presented a fundamental strategy for overcoming them. Finally we described how our prototype modeling environment, MelanEE, applies these strategies to unify and harmonize the way model content is visualized. The key idea is to use visualizers to fully disconnect concrete syntax concerns from domain information representation concerns so that the former has absolutely no influence on the latter. This allows multiple kinds of representation formats (corresponding to the different kinds of visualizers) to be applied to the same model context at the same time, and even mixed so that it is possible to incorporate a textual rendering of a part of a model within a graphical rendering of another part of a model. Tabular and Wiki oriented visualizations can also be supported using this approach by adding the necessary visualizers. In fact, in the long run even the GUI of a system (i.e. the GUI widgets) could potentially be supported using this approach.

A natural and profound consequence of this technology is that the notion of a language should be completely decoupled from the notion of its concrete visualization format. Thus, a language should no longer be characterized as being textual, graphical or tabular, since these are not properties of the language per se but of visualizations of a language. In other words, we propose that a language is characterized by the domain concepts it provides support for (i.e. a business process modeling language provides support for task and decision points, while a state modeling language provides the notions of states and transitions etc.) rather than by a particular concrete visualization that may have been associated with it "out of the box". It follows that using this technology a given domain-specific modeling language can have several visualization formats, some of them textual, some of them graphical.

This terminology is consistent with an ongoing trend for general-purpose modeling languages where the essence of what the "language" is has moved away from concrete syntax towards the meta-modeling concepts (c.f. abstract syntax). Such a trend has been very pronounced with the UML. However, the terminological consequences of this trend with the UML have not been carried through to its logical conclusion, so that many people still associate the Unified Modeling Language with the original concrete symbols rather than with its meta-model's concepts. Ideally, the concrete visualization should be regarded as one possible visualization of the UML (albeit the default one). Other textual visualizations have existed for a long time (e.g. XMI, HUTN).

To our knowledge MelanEE is the only prototype tool that supports textual and graphical visualizations of DSLs side-by-side in a uniform way by combining the principles of multi-level modeling with projectional editing (in the case of textual visualizations). However, we hope that when the benefits become apparent other prototypes will follow. Our group plans to continue this research by adding examples of other classes of visualizers in due course, such as tabular and wiki visualizers. This enriches the number of editors in which a modeler can view parts or the whole model in a unified way without dealing with multiple domain-specific language engineering tools. We also plan to expand the range of DSLs available in MelanEE.

7. REFERENCES

[1] C. Atkinson and R. Gerbig. Melanie: multi-level modeling and ontology engineering environment. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering: Modeling Wizards*, MW '12, pages 7:1–7:2, New York, NY, USA, 2012. ACM.

[2] C. Atkinson, R. Gerbig, and B. Kennel. Symbiotic general-purpose and domain-specific languages. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 1269–1272, Piscataway, NJ, USA, 2012. IEEE Press.

[3] C. Atkinson, R. Gerbig, and C. Tunjic. A multi-level modeling environment for sum-based software engineering. In *1st Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO) 2013*, 2013.

[4] C. Atkinson, M. Gutheil, and B. Kennel. A Flexible Infrastructure for Multilevel Language Engineering. *IEEE Transactions on Software Engineering*, 2009.

[5] C. Atkinson, B. Kennel, and B. Goß. The level-agnostic modeling language. In B. Malloy, S. Staab, and M. Brand, editors, *Software Language Engineering*, volume 6563 of *Lecture Notes in Computer Science*, pages 266–275. Springer Berlin Heidelberg, 2011.

[6] C. Atkinson, D. Stoll, and C. Tunjic. Orthographic service modeling. In *EDOCW*, pages 67–70, 2011.

[7] M. Breu, R. Breu, and S. Löw. Moveing forward: Towards an architecture and processes for a living models infrastructure. In *International Journal On Advances in Life Sciences, IARIA*, volume 3, pages 12–22, 2011.

[8] J. de Lara and E. Guerra. Deep meta-modelling with metadepth. In *Proceedings of the 48th international conference on Objects, models, components, patterns*, TOOLS'10, pages 1–20, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] Eclipse Foundation. OCLInEcore. `http://wiki.eclipse.org/MDT/OCLinEcore`, download June 2013.

[10] L. Engelen and M. van den Brand. Integrating textual and graphical modelling languages. *Electron. Notes Theor. Comput. Sci.*, 253(7):105–120, Sept. 2010.

[11] J. Espinazo-Pagán, M. Menárguez, and J. García-Molina. Metamodel syntactic sheets: An approach for defining textual concrete syntaxes. In *Proceedings of the 4th European conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA '08, pages 185–199, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] M. Eysholdt. Converging Textual and Graphical Editors. Eclipse Modeling Days 2009, 2009.

[13] Gentleware. Poseidon For DSLs. `http://www.gentleware.com/poseidon-for-dsls.html`, download April 2013.

[14] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit.* Addison-Wesley Professional, 1 edition, 2009.

[15] F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and refinement of textual syntax for models. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 114–129, Berlin, Heidelberg, 2009. Springer-Verlag.

[16] Itemis. XText. `http://www.xtext.org`, download April 2013.

[17] Jet Brains. Jet Brains MPS. `http://www.jetbrains.com/mps/`, download April 2013.

[18] L. C. Kats and E. Visser. The spoofax language workbench: rules for declarative specification of languages and ides. *SIGPLAN Not.*, 45(10):444–463, 2010.

[19] M. Lankhorst, H. Proper, and H. Jonkers. The architecture of the archimate language. In T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, and R. Ukor, editors, *Enterprise, Business-Process and Information Systems Modeling*, Lecture Notes in Business Information Processing, pages 367–380. Springer Berlin Heidelberg, 2009.

[20] J. Lara and E. Guerra. Domain-specific textual meta-modelling languages for model driven engineering. In A. Vallecillo, J.-P. Tolvanen, E. Kindler, H. Störrle, and D. Kolovos, editors, *Modelling Foundations and Applications*, volume 7349 of *Lecture Notes in Computer Science*, pages 259–274. Springer Berlin Heidelberg, 2012.

[21] J. Lara and H. Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2002.

[22] A. Ledeczi, M. Maroti, A. Bakay, and G. Karsai. The generic modeling environment. In *WISP'2001*, 2001.

[23] Microsoft. Visual Studio Visualization and Modeling SDK. `http://archive.msdn.microsoft.com/vsvmsdk`, download April 2013.

[24] Object Management Group (OMG). Diagram Definition Version 1.0. `http://www.omg.org/spec/DD/`, 2012.

[25] F. Pérez Andrés, J. Lara, and E. Guerra. Domain specific languages with graphical and textual views. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Applications of Graph Transformations with Industrial Relevance*, pages 82–97. Springer-Verlag, Berlin, Heidelberg, 2008.

[26] A.-W. W. Scheer. *Aris-Business Process Frameworks*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 1998.

[27] M. Scheidgen. Textual modelling embedded into graphical modelling. In I. Schieferdecker and A. Hartman, editors, *Model Driven Architecture âĂŞ Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 153–168. Springer Berlin Heidelberg, 2008.

[28] C. Schneider. On Integrating Graphical and Textual Modeling. Real-Time and Embedded Systems Group, Christian-Albrechts-Universität zu Kiel, 2011.

[29] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[30] J.-P. Tolvanen and M. Rossi. Metaedit+: defining and using domain-specific modeling languages and code generators. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '03, pages 92–93. ACM, 2003.

Implementing Semantic Feedback in a Diagram Editor

Niklas Fors
Department of Computer Science
Lund University, Lund, Sweden
niklas@cs.lth.se

Görel Hedin
Department of Computer Science
Lund University, Lund, Sweden
gorel@cs.lth.se

ABSTRACT

In editors for visual languages it is often useful to provide interactive feedback that depends on the static semantics of the edited program. In this paper we demonstrate how such feedback can be implemented using reference attribute grammars. Because the implementation is declarative, it is easy to modularize compiler and editor computations, reusing the compiler's program model in the editor. Furthermore, the declarative approach makes it easy to keep the program model and view consistent during editing. The approach is illustrated using a function block diagram language, with visual feedback on, for example, type checking and cyclic data flow.

Categories and Subject Descriptors

D.1.7 [**Programming Techniques**]: Visual Programming; D.2.6 [**Software Engineering**]: Programming Environments—*Graphical environments*; D.3.4 [**Programming Languages**]: Processors

General Terms

Languages

Keywords

Visual languages, diagram, reference attribute grammars, jastadd

1. INTRODUCTION

In implementing language tooling, there is often the need for providing several related tools: textual editor, visual editor, compiler, program analyzers, etc. Developing such language tooling is typically very costly [22, 18], and there are many current efforts on providing meta-tooling to reduce these costs. Examples include both work based on metamodeling, like the Eclipse Modeling Framework (EMF), and work based on grammars, e.g., Spoofax [13].

GMLD '13, July 01 2013, Montpellier, France

http://dx.doi.org/10.1145/2489820.2489827

While visual editors typically support structured editing, it is often useful to additionally provide *semantic feedback*, i.e., visualization and interaction that depends on static-semantic properties of the program. Examples include both visual display of static-semantic information, like displaying of type-checking errors, as well as interactive cues during editing gestures. An example could be to show if dragging a visual item to another location would change the semantics or not, or would introduce an error. If the underlying static-semantic analyses are done by a compiler, it is desirable to let the editor reuse this analysis, both in order to keep down development costs, and also to keep the tools consistent with each other.

We are currently exploring how reference attribute grammars (RAGs) [11] can be used in this context. RAGs are based on abstract syntax trees (ASTs), but provide reference attributes that link together AST nodes to form a graph. RAGs can also be integrated with EMF as shown by Bürger et al. [6], where containment relations correspond to the AST, and non-containment relations are mapped to reference attributes. RAGs have previously been shown useful for building extensible compilers for textual languages like Java [8] and Modelica [5].

We are applying RAGs for building tooling for a function block diagram language for control systems. The language, PicoDiagram, is based on existing languages in industrial use, and exhibits fairly advanced static-semantics, like diagram types, and data-flow based execution order. We have designed both a textual and visual syntax for PicoDiagram, and implemented both a compiler and a visual editor for it. The compiler is implemented in RAGs, using the metacompilation system JastAdd [12], and the editor is implemented using GEF, the Graphical Editing Framework in Eclipse.

In this paper, we investigate different opportunities for reusing static-semantics computations in the PicoDiagram compiler in order to provide semantic feedback during visual editing. In some cases, the compiler computations can be directly reused by the editor, and in other cases, they are modularly extended to provide specific feedback. This paper provides more details on the use of RAGs, compared to previous work [10].

The rest of this paper is organized as follows: Section 2 describes the PicoDiagram language, and Section 3 gives some background on RAGs. Sections 4, 5, and 6 describe different aspects of static-semantic analysis of PicoDiagram: name analysis, type checking, cyclic types, and cyclic connections. We explain how these aspects are implemented using RAGs, and how the implementation can be reused or

extended by the editor to provide semantic feedback. Section 7 describes related work, Section 8 provides a discussion and Section 9 concludes the paper.

2. PICODIAGRAM

PicoDiagram is an experimental language for control systems. It is inspired by a product from ABB, which in turn builds on the IEC-61131 standard for programmable logic controllers, including function block diagrams.

A PicoDiagram diagram consists of blocks and connections that model data flow. The blocks are instances of diagram types that may be defined either by other PicoDiagram diagrams, or by external C functions. We have designed PicoDiagram to have both a visual and a textual syntax. The textual syntax is a high-level model of a diagram, in the sense that visual properties such as shapes and colors are not part of the textual syntax. There are several benefits of having a textual syntax, for example, when generating diagrams or in concurrent development to allow the use of existing text-based merging tools.

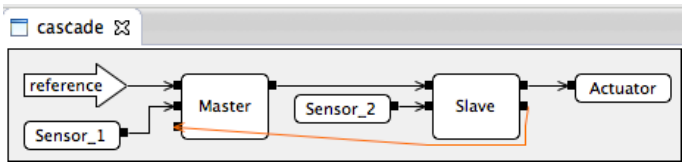
Figure 1 shows an example PicoDiagram program in both the visual and textual syntax. The program is a cascade regulator that controls an actuator using two controllers: a master controller uses input from one sensor to control the setpoint of the slave controller. The slave controller reads another sensor and sets the actuator value. There is also feedback information that goes from the slave to the master (the orange connection).

A diagram is executed periodically, reading/setting sensors and actuators with a given frequency. In each period, the blocks are executed in a sequence. In the case of a loop, like in the example, the compiler automatically breaks the loop by considering a certain connection as a *backwards connection*. In the visual editor, backward connections are colored orange. This is an example of giving the user semantic feedback. For values coming via backward connections, the old value from the previous period will be used.

The algorithm for computing the backwards connections makes use of a positional order between the blocks, see Section 6. For the visual syntax, the positional order corresponds to the distance to the origin, i.e., the upper left corner of the diagram. For the textual syntax, it corresponds to the declaration order. The execution order follows the forward data flow connections, and uses the positional order to define an unambiguous total order. In a previous paper, we defined the execution order declaratively on diagrams with acyclic data flow, and described how to provide semantic feedback on how blocks in a diagram can be moved without affecting the execution order [10]. In this paper, we extend this result by describing an algorithm to remove data flow cycles in diagrams using the positional order and give a more thorough description on the use of RAGs.

A diagram type can be instantiated several times. For example, a larger system may include several instances of the `Cascade` diagram.

We have implemented two different tools for PicoDiagram: a visual editor where the user can edit programs using the visual syntax, and a compiler that translates the text representation to C code. The visual editor stores the programs in the text representation, and it is also possible to use a normal text editor to edit the programs. Both tools reuse the same parser and core RAG specification, but they can also extend the RAG specification with their own attributes.




```
diagramtype Cascade(Int reference) {
  Sensor Sensor_1;
  Master Master;
  Sensor Sensor_2;
  Slave Slave;
  Actuator Actuator;
  connect(Sensor_1.value, Master.pv);
  connect(reference, Master.sv);
  connect(Master.u, Slave.sv);
  connect(Sensor_2.value, Slave.pv);
  connect(Slave.u, Actuator.value);
  connect(Slave.feedback, Master.slaveFeedback);
}
```

Figure 1: A cascade regulator in PicoDiagram. Visual and textual syntax.

For example, the editor can add attributes that aids the implementation of semantic feedback, and the compiler adds attributes for code generation. This way the tools are kept consistent with each other, yet can be tailored as desired.

3. REFERENCE ATTRIBUTE GRAMMARS

We use the semantic formalism *Reference Attribute Grammars* (RAGs) [11] to define the meaning of a diagram in the PicoDiagram language. A diagram is represented as an attributed *abstract syntax tree* (AST), typically created by a parser or by an interactive visual editor. The attributes are derived values computed given a set of equations and the AST. Equations are directed, with an attribute on the left hand side, and an expression over other attributes on the right hand side. The attributes are *declarative* in the sense that each attribute will have a value that is equal to the right-hand side of its defining equation, and the expressions inside an equation are forbidden to have any externally observable side effects. An *attribute evaluation engine* automatically evaluates the attributes, according to their dependencies.

RAGs extends Knuth's Attribute Grammars [14] with *reference* attributes, i.e., attributes whose values are references to other nodes in the AST. This makes is possible to superimpose graphs on top of the AST.

For implementation, we use the JastAdd metacompilation tool [12] which supports RAGs, object-oriented ASTs, and aspect-oriented modularization of RAG specifications. JastAdd also supports circular attributes [16], i.e., attributes that may transitively depend on themselves, and for which the evaluation engine uses fixed-point iteration to compute the value.

JastAdd attributes are accessed by calling them, like a method. In fact, the attributes are translated to Java methods by the JastAdd tool. For efficiency, the JastAdd attribute evaluator does not evaluate an attribute until its value is needed (i.e., until it is called), and evaluated attributes are cached for fast future access. The attribute caches are flushed whenever the AST is modified, for example after an edit action.

3.1 A Simple RAG Example

To illustrate how RAGs work, consider the following simple abstract grammar (written in JastAdd syntax):

```
A ::= B1:B B2:B;
B ::= <ID:String>;
C : B;
```

This abstract grammar corresponds to a Java class hierarchy, and defines that:

- `A`, `B`, and `C` are classes
- `A` has two children: `B1` and `B2`, both of type `B`
- `B` has a token: `ID` of type `String`
- `C` is a subclass of `B`

The JastAdd RAG specification below defines attributes over this abstract grammar. Line 1 declares an attribute `B.otherB` of type `B`. I.e., `otherB` is a reference attribute in `B` nodes, and that will point to another `B` node. Lines 2 and 3 show two different equations defining `otherB`: the first one holds for the `B1` child of an `A` node, and the other one for the `B2` child. The right-hand side of the equation is in the context of `A`. The attribute `otherB` is *inherited* (in the attribute grammar sense), meaning that the `B` node does not know how the value is defined, but delegates this responsibility to the parent node, i.e., to the `A` node in this case.[1]

```
1   inh B B.otherB();
2   eq A.getB1().otherB() = getB2();
3   eq A.getB2().otherB() = getB1();
```

The result of this specification is that the two `B` children refer to each other, thus forming a cyclic data structure, as shown for the example AST in Figure 2. This AST corresponds to the expression `new A(new B("B"), new C("C"))`. Note that one of the `B` nodes is actually of the subclass `C`, illustrating that attributes are inherited (in the usual object-oriented sense) by subclasses.

Because RAGs are declarative, the order of specification of individual attributes and equations is irrelevant, and they can be split up into different modules based on what is practical to reuse in different tools. The following small RAG module illustrates the modular addition of an attribute `B.name` of type `String`. This is a *synthesized* attribute, meaning that there must be an equation defining its value in the same node. Line 5 gives a definition of the attribute value for `B` nodes. Line 6 illustrates how this definition is overridden for `C` nodes, relying on the usual object-oriented features of inheritance and overriding. The equation on line 6 also illustrates how information in distant nodes can be accessed: the `ID` of the other `B` node is accessed via the reference attribute `otherB`.

```
4   syn String B.name();
5   eq B.name() = getID();
6   eq C.name() = otherB().getID();
```

[1]JastAdd also supports inheritance in the object-oriented sense, and if there is a risk on confusion, we will comment on what kind of inheritance we mean. Both uses of the term originate from the late 60s when both OO and AGs were invented.

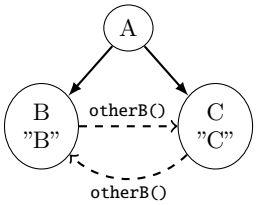


Figure 2: Example AST with reference attributes

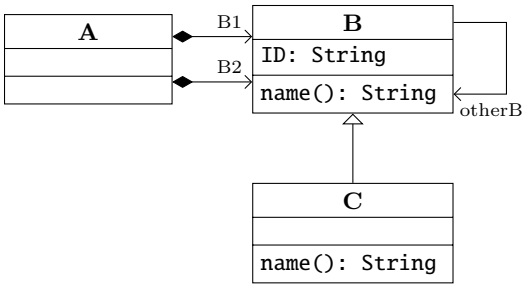


Figure 3: UML diagram for the RAG example

The RAG corresponds to the UML diagram depicted in Figure 3. The class hierarchy, the parent-child relations (i.e., containment relations), and UML attributes like `ID`, are specified in the abstract grammar. Non-containment relations like `otherB`, and methods like `name`, are specified as RAG attributes, and their values are computed automatically by the underlying RAG evaluation engine.

Additional JastAdd/RAG mechanisms used in this paper include parameterized attributes and circular attributes, and are explained as part of our discussion of the PicoDiagram compiler and editor.

4. NAME AND TYPE ANALYSIS

The name analaysis is concerned with binding name uses to name declarations. For PicoDiagram, we have two separate namespaces, one for types and one for variables. Type uses and type declarations are represented by the classes `TypeUse` and `TypeDecl`, respectively, and `VarUse` and `VarDecl` represent variables. The bindings are implemented by a reference attribute `decl` in `TypeUse` and `VarUse`, referring to the appropriate declaration node.

An example of an attributed AST is shown in Figure 4. The structure of this AST is defined by the core abstract grammar of PicoDiagram, which is shown in Figure 5.[2] In this grammar, we can see that a `Program` consists of a list of `DiagramType`s, which in turn consists of a name, `Parameter`s, `Block`s and `Connection`s. The classes `Parameter` and `Block` are subclasses to `VarDecl` and have a `TypeUse` each. The `Connection` class has at least one `VarUse`.[3]

Figure 6 illustrates a simple diagram type `S`, using the textual syntax. It has one input parameter `p`, a block `t` of type `T` and a connection between the local parameter `p` and the parameter `p2` in type `T`. There are three type uses in this example: the parameters `p` and `p2` use the primitive type `Int` and the block `t` uses the diagram type `T`. The variable uses

[2]The full PicoDiagram language contains additional constructs like structs, floats, booleans, etc.

[3]The source of a connection can be an integer constant.

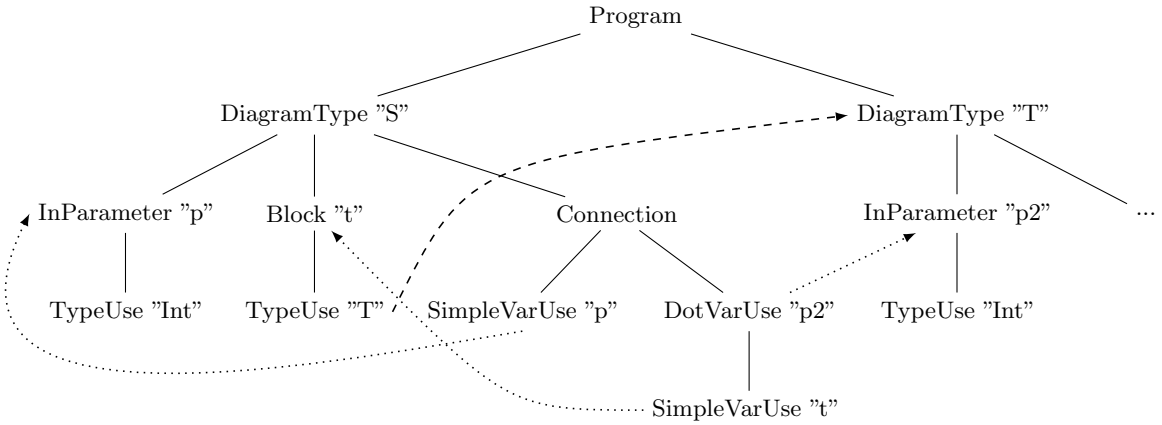


Figure 4: Example AST. Dashed lines are references from type uses to type declarations. Dotted lines are references from variable uses to variable declarations. The type `Int` and references to it have been omitted.

```
Program ::= DiagramType*;

abstract TypeDecl;
DiagramType : TypeDecl ::=
  <ID> Parameter* Block* Connection*;
IntType : TypeDecl;
BoolType : TypeDecl;
UnknownType : TypeDecl;

abstract VarDecl;
abstract Parameter : VarDecl ::= TypeUse <ID>;
InParameter : Parameter;
OutParameter : Parameter;
Block : VarDecl ::= TypeUse <ID>;

Connection ::= Source:Expr Target:VarUse;

abstract Expr;
IntConst : Expr ::= <Value:int>;
abstract VarUse : Expr;
SimpleVarUse : VarUse ::= <ID>;
DotVarUse : VarUse ::= VarUse <ID>;

TypeUse ::= <ID>;
```

Figure 5: Core abstract grammar for PicoDiagram. Abstract classes (like in Java) use the keyword `abstract`. Kleene star means a list of children.

```
diagramtype S(Int p =>) {
  T t;
  connect(p, t.p2);
}
diagramtype T(Int p2 =>) { ... }
```

Figure 6: Simple example for name analysis

```
syn TypeDecl TypeUse.decl()
  = lookupType(getID());

inh TypeDecl TypeUse.lookupType(String name);
eq Program.getDiagramType()
    .lookupType(String name) {
  TypeDecl typeDecl = lookupPredefType(name);
  if (typeDecl == null)
    typeDecl = lookupDiagramType(name);
  return typeDecl;
}
```

Figure 7: Name binding for types

are found in the connection: `p` and `t.p2`. Note that the binding of `p2` depends on the type of `t`. The corresponding AST of this example and the name bindings are shown in Figure 4.

As described earlier, both `TypeUse` and `VarUse` have a reference attribute `decl` that refers to the corresponding declaration. The RAGs implementation follows a typical pattern for name analysis where the `decl` attribute delegates the searching for the declaration to an inherited attribute `lookup` that is defined by the context in the AST [7]. The `lookup` attributes are examples of *parameterized* attributes, that can be thought of as functions, but where the results are cached (i.e., memoized).

Figure 7 shows the name binding for types, where the `decl` attribute uses the inherited attribute `lookupType` which in turn is defined by the `Program` node for its diagram types. The attribute first searches among the predeclared types (integer and boolean) using the attribute `lookupPredefType` and then continues among the user defined diagram types using the attribute `lookupDiagramType`. See the appendix on how these helper attributes are defined.

The implementation of name bindings for variables is shown in Figure 8. For `SimpleVarUse`, the attribute `decl` uses the `lookup` attribute directly. The enclosing diagram type defines the `lookup` attribute for all its children. It first searches among its parameters and then continues with searching among its blocks. The definitions of the helper attributes can be seen in the appendix. For `DotVarUse`, the attribute

```
syn VarDecl VarUse.decl();

eq SimpleVarUse.decl()
  = lookup(getID());

inh VarDecl VarUse.lookup(String name);
eq DiagramType.getChild().lookup(String name)
{
  VarDecl decl = localParameterLookup(name);
  if (decl == null)
    decl = localBlockLookup(name);
  return decl;
}


eq DotVarUse.decl()
  = getVarUse().decl() == null
  ? null
  : getVarUse().decl().findMember(getID());

syn VarDecl VarDecl.findMember(String name)
  = null;
eq Block.findMember(String name)
  = type().localParameterLookup(name);
```

Figure 8: Name binding for variables

```
syn TypeDecl TypeUse.type()
  = decl() != null
  ? decl()
  : program().unknownType();

syn TypeDecl VarDecl.type();
eq Parameter.type() = getTypeUse().type();
eq Block.type() = getTypeUse().type();

syn TypeDecl Expr.type();
eq VarUse.type()
  = decl() != null
  ? decl().type()
  : program().unknownType();
eq IntConst.type() = program().intType();
```

Figure 9: Type analysis

`decl` is defined by asking the declaration of its child if it has a member using the attribute `findMember`. For example, in the expression `t.p2`, we use the declaration of `t` to define `p2`. In this example, the declaration of `t` is a block, and `Block` defines `findMember` by searching among the parameters of its type, that is, `p2`.

Types of expressions and declarations are represented by `type` attributes that refer to `TypeDecl` AST nodes. They are straightforward to define with the help of the `decl` attributes, as shown in Figure 9. Predeclared types, like `intType`, are represented using attributes in the program root. Semantic feedback on errors, for example, connections that connect incompatible types, can be shown in the visual editor using, for instance, colors or icons and associated error messages.

5. CYCLIC DIAGRAM TYPES

If a diagram type contains a block of its own type (directly or transitively), this constitutes a compile-time error

```
diagramtype T() {
  S s;
}
diagramtype S() {
  T t;
}
```

Figure 10: Cyclic diagram types (a compile-time error)

```
syn Set<TypeDecl> TypeDecl.reachable()
  circular[new HashSet<TypeDecl>()];
eq TypeDecl.reachable()
  = new HashSet<TypeDecl>();
eq DiagramType.reachable() {
  Set<TypeDecl> set = new HashSet<TypeDecl>();
  for (Block b: getBlocks()) {
    if (b.type().isDiagramType()) {
      set.add(b.type());
      set.addAll(b.type().reachable());
    }
  }
  return set;
}

syn boolean DiagramType.isCircular()
  = reachable().contains(this);
```

Figure 11: Type reachability

since it would lead to endless unfolding of blocks. Figure 10 illustrates this error.

We use a circular attribute [16] to identify such cycles. The value of a circular attribute is computed using a fixed-point iteration. We define the circular attribute `reachable` that is the set of all reachable diagram types from a diagram type, as can be seen in Figure 11. If the diagram type is in this set, the diagram type is circular, otherwise it is not. The attribute is defined on `TypeDecl`, to avoid explicit type casts. The attribute returns an empty set for all other types than diagram types. For diagram types, it returns the types of its blocks and the reachable types from these types. The attribute `isCircular` on `DiagramType` tells whether the diagram type is circular or not, by using the attribute `reachable`.

5.1 Semantic Feedback

We can now provide semantic feedback for cyclic diagram types by coloring blocks red that are in a type cycle. For example, in viewing the diagram type `T` from the example in Figure 10, the component block `s` would be colored red as follows:

To aid the implementation of this visual semantic feedback, an attribute `isCircular` is defined for `Block` as follows:

```
syn boolean Block.isCircular() =
  type().reachable().contains(diagramType());
```

The attribute checks if the enclosing diagram type of the block is reachable from the type of the block. If this true, then the block is in a type cycle. The enclosing diagram type is accessed by the inherited attribute `diagramType`.

6. CYCLIC CONNECTIONS

The connections define data flow between blocks, transferring values from source blocks to target blocks. Normally, a connection's source block should therefore execute before the target block. However, this becomes problematic when the connections form a cycle, since we do not know which block to execute first. We solve this by constructing a graph where edges corresponding to certain connections are removed, so that the resulting graph becomes acyclic. All blocks are executed periodically, so for a connection corresponding to a removed edge, we use the source block value from the *previous period* and feed it into the target block.

The idea behind the algorithm for removing edges is to only remove those corresponding to connections that visually go backwards, meaning that the source block is further away from the origin than the target block. If there are several such connections in a cycle, we choose the one whose target is closest to the origin. This is simple to understand for a user of the visual editor, and fits with the way control diagrams are usually layed out.

We model the diagram as a directed graph. In the diagram, the connections go between specific *ports* on the blocks, representing different variables. For the constructed graph, we abstract this to edges between the blocks, since we are only interested in ordering complete blocks. So if there are more than two connections between two specific blocks, they will be represented by a single edge in the graph.

The algorithm for breaking cycles is shown in Figure 12 and we use strongly connected components (SCCs) to detect cycles. An SCC is the maximal set of vertices where there is a path from all vertices in the SCC to all other vertices in the SCC, and can be computed in linear time using Tarjan's algorithm [21]. The cycle-breaking algorithm is iterative and removes edges until the graph is acyclic. For each iteration, all SCCs are computed. For each SCC consisting of more than one vertex, we select the vertex v_1 that is closest to the origin in the SCC. We then remove all edges from all vertices in the same SCC to v_1. The function $po(v)$ maps a block to an integer and preserves the positional order.

An example of a cyclic graph is shown in Figure 13. Running the algorithm on this graph, two SCCs are computed during the first iteration: $\{\alpha\}$ and $\{a, b, c, d, e\}$. We are only interested in SCCs with more than one vertex, that is, the latter SCC. In this SCC, vertex a is closest to the origin. We thus remove all edges from vertices in the same SCC to a, that is, the edge (c, a). During the second iteration, we compute the SCC $\{c, e\}$ and therefore remove the edge (e, c). The graph is now acyclic and we are finished.

The algorithm is implemented as a method `breakCycles` on `DiagramType` and uses attributes to obtain the information needed to construct the original cyclic graph. The method returns a wrapper object consisting of all predecessor and successor sets for the corresponding acyclic graph. The attribute `DiagramType.predSucc` is defined by calling `breakCycles`, as can be seen in Figure 14. This is fine since although `breakCycles` is implemented imperatively, it does not result in any externally observable side-effects. Note that `breakCycles` is only computed once for each diagram

remove all edges (v, v)
while graph has cycles **do**
  compute all SSCs
  for all SCC where $|SCC| > 1$ **do**
    let $v_1 \in SCC$, where $\forall v \in SCC : po(v_1) \leq po(v)$
    remove all edges (v, v_1), where $v \in SCC$
  end for
end while

Figure 12: Algorithm for breaking cycles

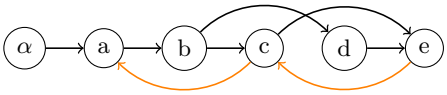


Figure 13: Cyclic graph. Orange edges are removed.

type, since JastAdd caches attribute values. The wrapper object consists of two maps: `pred` maps a `Block` to its predecessors and `succ` maps it to its successors. We use these two maps to define the attributes `pred()` and `succ()` on `Block`.

6.1 Semantic Feedback

Connections corresponding to a removed edge will have a different meaning than normal edges, since their values will be delayed one period. To help the user in understanding this, we provide semantic feedback by coloring these connections orange, as can be seen in Figure 1. To aid this implementation, an attribute `isBroken` is defined on `Connection`. A connection is defined as broken if both its source and targets are blocks, and if the target block is not a successor to the source block in the acyclic graph, see Figure 15.

We also provide interactive semantic feedback while the user moves a block, since this could affect how cycles are broken, see Figure 16. Here, the user is dragging the `Slave` block, and the green box shows its current position. The

```
syn PredSucc DiagramType.predSucc()
  = breakCycles();

inh Set<Block> Block.pred();
eq DiagramType.getBlock(int i).pred()
  = predSucc().pred.get(getBlock(i));

inh Set<Block> Block.succ();
eq DiagramType.getBlock(int i).succ()
  = predSucc().succ.get(getBlock(i));
```

Figure 14: Attributes `predSucc()`, `pred()` and `succ()`

```
syn boolean Connection.isBroken() {
  if (getSource().isBlock()
      && getTarget().isBlock()) {
    Block src = getSource().block();
    return !src.succ().contains(getTarget());
  } else {
    return false;
  }
}
```

Figure 15: Attribute `isBroken`

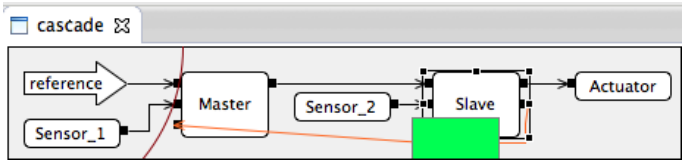


Figure 16: Semantic feedback for the cascade regulator. Moving the `Slave` block to the left of the red circle segment will change how the cycle is broken.

block stays green as long as releasing the block would not change the cycle breaking. However, if the user drags the block to the far left, across the red circle segment, i.e., closer to the origin than the `Master` block, the box will turn yellow, indicating that releasing it at that point may change the way cycles are broken.

In the general case, there could be both an upper and a lower bound for how a block could be moved without changing the cycle breaking. For example, how can we move the block c in Figure 13 without changing how the cycles are broken? By looking at the algorithm in Figure 12, we can see that only edges between vertices in the same SCC are removed, meaning that c is only constrained by vertices in the same SCCs. Block c is in two SCCs: $\{a, b, c, d, e\}$ for iteration 1 and $\{c, e\}$ for iteration 2. For each SCC, the only way to change which edges that are removed is to change the first vertex v_1 to another vertex. For block c, we can do this by moving c before a (first SCC) or after e (second SCC). Hence, c is constrained by a as the lower bound and e as the upper bound. The general constraint for all SCCs is $\forall v \in SCC : po(v_1) \leq po(v)$.

To implement the move feedback, we changed the implementation of the method `breakCycles` to also return information about what vertices a block is constrained by. We then use this information in the visual editor to provide the semantic feedback.

7. RELATED WORK

The focus of this paper is on reusing static semantics between different tools and providing semantic feedback in the visual editor.

In the Eclipse world, there are several ways to create a visual editor, such as using the Graphical Editing Framework (GEF) [1], the Graphical Modeling Framework (GMF) [2] or Graphiti [2]. The visual editor for PicoDiagram has been created using GEF. GMF is built on top of GEF and requires an EMF model to define a visual editor. These projects focus on lowering the burden for creating visual editors, and they focus less on semantic analysis and semantic feedback. One interesting work by Bürger et al. [6] combines RAGs and EMF, to define the static semantics of EMF models using RAGs. By building on their work, one possible future direction would be to use GMF instead of GEF to create the visual editor for PicoDiagram, in order to decrease the effort spent on the visual editor.

The blocks in PicoDiagram are instantiations of diagram types, and the appearance of these blocks depends on their types. A diagram type defines how many parameters it has and the visual representation of a block of this type reflects this. The language workbench MetaEdit+ [3] supports, since version 5.0, something they call dynamic ports, which they have created to support situations like this.

Another tool is xText [9] that is used for creating textual domain specific languages. It supports name binding for languages with simple name rules, otherwise the user needs to provide a Java implementation of the name binding. Also in the Spoofax language workbench, which has been used for graphical languages [23], a DSL is developed for name binding [15]. While such specific support can be useful in many cases, it is limited, both in what domain is supported (for example, name analysis), and in the generality of how that domain is supported (for example, limited to certain known kinds of scope rules). In contrast, RAGs is a general declarative formalism for defining *any* static-semantic analysis, and has been successfully used to implement complex name and type analyses for languages like Java [8] and Modelica [5].

Many metamodelling-based tools, including EMF, MetaEdit+, and xText, use constraint languages like OCL (Object Constraint Language) for specifying static-semantic constraints on the model. RAGs serve a similar purpose, but in addition supports *building* derived parts of the model itself, through the use of the reference attributes. Some of these derived parts, e.g., name bindings, may correspond to explicitly defined non-containment relations in EMF, whereas others go beyond EMF+OCL, like the representation of the broken cycles in section 6.

Schmidt et. al. [19] have used attribute grammars to implement visual editors. They use something called visual patterns that are predefined reusable implementations of common visual representations, such as lists, graphs, tables and line connections. These patterns have been implemented using attribute grammars. The user can refine these patterns and also create new visual patterns. The focus of their work is on using attribute grammars to define visual languages. A difference is that we use reference attributes and they do not. It would, however, be interesting to combine visual patterns with our approach.

In this paper, we handle cyclic data flows by removing connections based on the layout. In Simulink, a diagram can contain loops, which are called algebraic loops, that Simulink tries to solve mathematically [17]. If a loop contains a block with a discrete value as output, then Simulink cannot solve that loop. When Simulink cannot solve an algebraic loop, the user can add a so-called delay block to explicitly break the loop. In PicoDiagram, when a connection is removed due to a cycle, a delay is added implicitly, which corresponds to a delay block in Simulink.

Modelica is a language for modeling and simulating physical systems [4]. It has also a textual and a visual syntax, like PicoDiagram. However, in contrast to PicoDiagram, its semantics is not layout-dependent.

8. DISCUSSION

The declarative nature of attribute grammars allows attributes to be reused by different tools, for example, by the visual editor to provide semantic feedback. The attributes are declarative in the sense that they always compute the same value and no externally visible side-effects are allowed. This means that we can reuse the attributes in the visual editor without thinking about in what order they are computed. By reusing attributes between different tools, it is easier to keep the tools consistent with each other and the semantics is only required to be specified once.

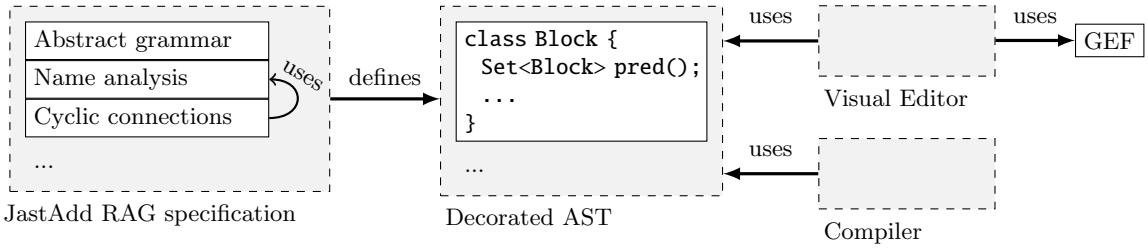


Figure 17: The visual editor and the compiler reuse the same decorated AST.

Figure 17 shows the overall architecture of how the RAG specification defines the attributed AST which is reused by both the compiler and the editor. In addition to this, the compiler and the visual editor can modularly add new attributes.

Attributes are computed values given a set of equations and an AST. When the user changes something in the visual editor, for example, adding a connection or a block, attribute values may become invalid, since the AST has changed. When the AST changes, we currently flush all attribute values, and they are recomputed again when they are needed (due to on-demand evaluation). In industry, large problems are modularized to many smaller diagrams, often limited to at most 50 blocks, since larger diagrams are difficult to understand and to fit on screen or paper. For PicoDiagram, there are no performance problems with individual diagrams: feedback is immediate even when interacting with diagrams with 500 blocks and 500 connections. However, further studies are needed to see how our approach scales to large libraries of diagrams. If performance needs to be improved, a possibility is to apply incremental evaluation for RAGs, where flushing is limited to attributes that depend on the AST change. Such incremental evaluation is currently investigated by Söderberg et. al. [20].

The base structure in EMF is a graph, whereas for RAGs it is a tree (the AST). With reference attributes, a graph can be super-imposed on the tree. We think having a high-level textual syntax also for a visual language has huge benefits. It makes the stored programs independent of tooling, and allows all existing text-level tools to be used. At the same time, the trees created by the parser are automatically decorated with attributes representing graphs, making the representation suitable for visual editors. However, the visual editor cannot change the reference attributes directly, but needs to transform visual editing operations to corresponding tree changes.

9. CONCLUSION

In this paper we have shown how to implement semantic feedback in a visual editor by reusing the static semantic specification. To only specify the static semantics once, and reuse the attributed AST in different tools, makes it is easy to keep the tools consistent with each other. As a case study, we have created a language called PicoDiagram that is similar to function block diagrams. The language has both a textual and a visual syntax, and we define its semantics using reference attribute grammars (RAGs), which is a declarative, but executable, formalism. For PicoDiagram, we have created a compiler and a visual editor that both reuse the same semantic specification including name analysis, type analysis, detection of cyclic diagram types and removal of cyclic connections based on layout. Then we showed examples of semantic feedback in the visual editor, for example, coloring blocks red that form a cyclic diagram type and displaying how a block can be moved without changing how the cyclic connections are broken.

In the future we would like to generate the visual editor from a specification, instead of coding it manually using GEF. It would be interesting to combine RAGs with EMF and GMF to create a visual editor, by building on the work by Bürger et al. [6]. It would also be interesting to use incremental evaluation [20] in the visual editor.

10. ACKNOWLEDGEMENTS

We would like to thank Ulf Hagberg, Christina Persson and Stefan Sällberg at ABB for sharing their expertise about the ABB tools for building control systems. This work was partly financed by the Swedish Research Council under grant 621-2012-4727.

11. REFERENCES

[1] The graphical editing framework (gef). `http://www.eclipse.org/gef/`.

[2] The graphical modeling project. `http://www.eclipse.org/modeling/gmp/`.

[3] Metaedit+ 5.0. `http://www.metacase.com/`.

[4] The modelica association. `http://www.modelica.org`, 2013.

[5] J. Åkesson, T. Ekman, and G. Hedin. Implementation of a Modelica compiler using JastAdd attribute grammars. *Science of Computer Programming*, 75(1-2):21–38, Jan. 2010.

[6] C. Bürger, S. Karol, C. Wende, and U. Aßmann. Reference attribute grammars for metamodel semantics. In *Software Language Engineering (SLE 2010)*, pages 22–41, 2011.

[7] T. Ekman and G. Hedin. Modular name analysis for Java using JastAdd. In *Generative and Transformational Techniques in Software Engineering, International Summer School (GTTSE 2005)*, volume 4143 of *LNCS*. Springer, 2006.

[8] T. Ekman and G. Hedin. The Jastadd Extensible Java Compiler. In *OOPSLA 2007*, pages 1–18. ACM, 2007.

[9] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 307–309. ACM, 2010.

[10] N. Fors and G. Hedin. Reusing semantics in visual editors: A case for reference attribute grammars. In

Proc. 12th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2013), volume 58. ECEASST, 2013.

[11] G. Hedin. Reference Attributed Grammars. In *Informatica (Slovenia)*, 24(3), pages 301–317, 2000.

[12] G. Hedin and E. Magnusson. JastAdd: an aspect-oriented compiler construction system. *Sci. of Comp. Prog.*, 47(1):37–58, 2003.

[13] L. C. L. Kats and E. Visser. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *OOPSLA 2010*, pages 444–463. ACM, 2010.

[14] D. E. Knuth. Semantics of Context-free Languages. *Math. Sys. Theory*, 2(2):127–145, 1968. Correction: *Math. Sys. Theory* 5(1):95–96, 1971.

[15] G. D. P. Konat, L. C. L. Kats, G. Wachsmuth, and E. Visser. Declarative name binding and scope rules. In *Software Language Engineering, 5th International Conference, SLE 2012*, volume 7745 of *Lecture Notes in Computer Science*, pages 311–331. Springer, 2013.

[16] E. Magnusson and G. Hedin. Circular Reference Attributed Grammars - Their Evaluation and Applications. *Science of Computer Programming*, 68(1):21–37, 2007.

[17] MathWorks. *Simulink documentation, Simulating Dynamic Systems*, r2013a edition.

[18] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, 2005.

[19] C. Schmidt and U. Kastens. Implementation of visual languages using pattern-based specifications. *Softw., Pract. Exper.*, 33(15):1471–1505, 2003.

[20] E. Söderberg and G. Hedin. Incremental evaluation of reference attribute grammars using dynamic dependency tracking. Technical Report 98, Lund University, April 2012. LU-CS-TR:2012-249, ISSN 1404-1200.

[21] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[22] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[23] O. van Rest, G. Wachsmuth, J. Steel, J. G. Süß, and E. Visser. Robust real-time synchronization between textual and graphical editors. In *International Conference on Model Transformation (ICMT 2013)*, Budapest, Hungary, June 2013.

[24] H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-order attribute grammars. In *PLDI*, pages 131–145, 1989.

APPENDIX

The attributes `lookupPredefType` and `lookupDiagramType` that are used in the name analysis for type names are shown in Figure 18. The attribute `lookupPredefType` access the predefined types using the non-terminal attribute (NTA) [24] `predefTypeDecls`. The value of an NTA is a subtree and is defined by an equation. The equation for the attribute `predefTypeDecls` defines a list of `TypeDecl`s corresponding to predefined declarations such as the integer and boolean type. The attributes `localParameterLookup` and `localBlockLookup` that are used in the name analysis for variable names are shown in Figure 19. These attributes access their children to find the corresponding declaration.

```
syn TypeDecl Program
    .lookupPredefType(String name) {
  for (TypeDecl td: predefTypeDecls())
    if (td.name().equals(name))
      return td;
  return null;
}
syn TypeDecl Program
    .lookupDiagramType(String name) {
  for (DiagramType dt: getDiagramTypes())
    if (dt.getID().equals(name))
      return dt;
  return null;
}
```

Figure 18: Helper attributes for looking up type names

```
syn VarDecl TypeDecl
  .localParameterLookup(String name) = null;
eq DiagramType
    .localParameterLookup(String name) {
  for (Parameter p: getParameters())
    if (p.getID().equals(name))
      return p;
  return null;
}

syn Block DiagramType
    .localBlockLookup(String name) {
  for (Block b: getBlocks())
    if (b.getID().equals(name))
      return b;
  return null;
}
```

Figure 19: Helper attributes for looking up variable names

Traceability Visualization in Metamodel Change Impact Detection

Juri Di Rocco
University of L'Aquila
I-67100 L'Aquila, Italy
juri.dirocco@univaq.it

Davide Di Ruscio
University of L'Aquila
I-67100 L'Aquila, Italy
davide.diruscio@univaq.it

Ludovico Iovino
University of L'Aquila
I-67100 L'Aquila, Italy
ludovico.iovino@univaq.it

Alfonso Pierantonio
University of L'Aquila
I-67100 L'Aquila, Italy
alfonso.pierantonio@univaq.it

ABSTRACT

In Model-Driven Engineering metamodels are typically at the core of an *ecosystem* of artifacts assembled for a shared purpose. Therefore, modifying a metamodel requires care and skill as it might compromise the integrity of the ecosystem. Any change in the metamodel cannot prescind from recovering the ecosystem validity. However this has been proven to be intrinsically difficult, error-prone, and labour-intensive. This paper discusses how to generate and visualize traceability information about the dependencies between artifacts in a ecosystem and their related metamodel. Being able to understand how and where changes affect the ecosystem by means of intuitive and straightforward visualization techniques can help the modeler in deciding whether the changes are sustainable or not at an early stage of the modification process.

1. INTRODUCTION

Model-Driven Engineering [1] (MDE) aims at capturing *problems* in terms of concepts that are much closer to the application domain rather than to the underlying technological assets. Problems are then consistently mapped to *solutions* by means of model operations generally defined by model transformations. Both problems and solutions are described with the help of *models* expressed in terms of concepts and relationships among them given in *metamodels*. Additionally metamodels characterize also a wide range of further artifacts, including model transformations, textual and diagrammatic editors, and code generators which are *typed*[1] after them. Therefore, metamodels are at the core of an aggregation of artifacts assembled for a shared purpose, called *metamodeling ecosystem* [2, 3].

Similarly to software, metamodels can be subject to internal or external evolutionary pressures [4]. However, because of the dependencies among the artifacts, changing a metamodel requires the rest of the ecosystem to be adapted in order to remain valid or consistent. As a consequence, before changing a metamodel it is of crucial relevance to measure the impact of the changes among the artifacts in order to understand whether the evolution is sustainable or not. In a previous work [5] we have already presented techniques based on simple cost functions which may help in understand how much changing a metamodel is affordable. Nonetheless, quantitatively analyzing the impact can be misleading as it does not convey any particular about *where* and *how* the adaptations have to be applied.

In this paper, we present a change impact visualization which complements the aforementioned work. In particular, traceability information about the dependencies between the metamodel and the specific artifact under study is generated and visualized. This permits the modeler to assess the significance of the change by using an adaptation density concept which visualizes and gives an *intuition* about how localized and how many adaptations are required within a given artifact fragment. The approach is generic and can be used for different kinds of artifacts.

The paper is structured as follows. Section 2 presents the background. Section 3 outlines a methodology to deal with the co-evolution problem. In Section 4 the visualization of the traceability is explained. Section 5 analyses different results obtained with our approach presenting possible visualizations and their meaning. In Section 6 we discuss alternative approaches and application using TraceVis and analize other approaches to change impact detection and traceability. Section 7 concludes the paper outlining future works.

2. BACKGROUND: COUPLED EVOLUTION PROBLEM

As already mentioned, metamodels rarely exist in isolation since they are the cornerstone upon which many modeling artifacts are built. The problem of how to let the metamodel

[1]For instance, a transformation usually have a source and a target metamodel.

GMLD '13, July 01 2013, Montpellier, France

http://dx.doi.org/10.1145/2489820.2489824.

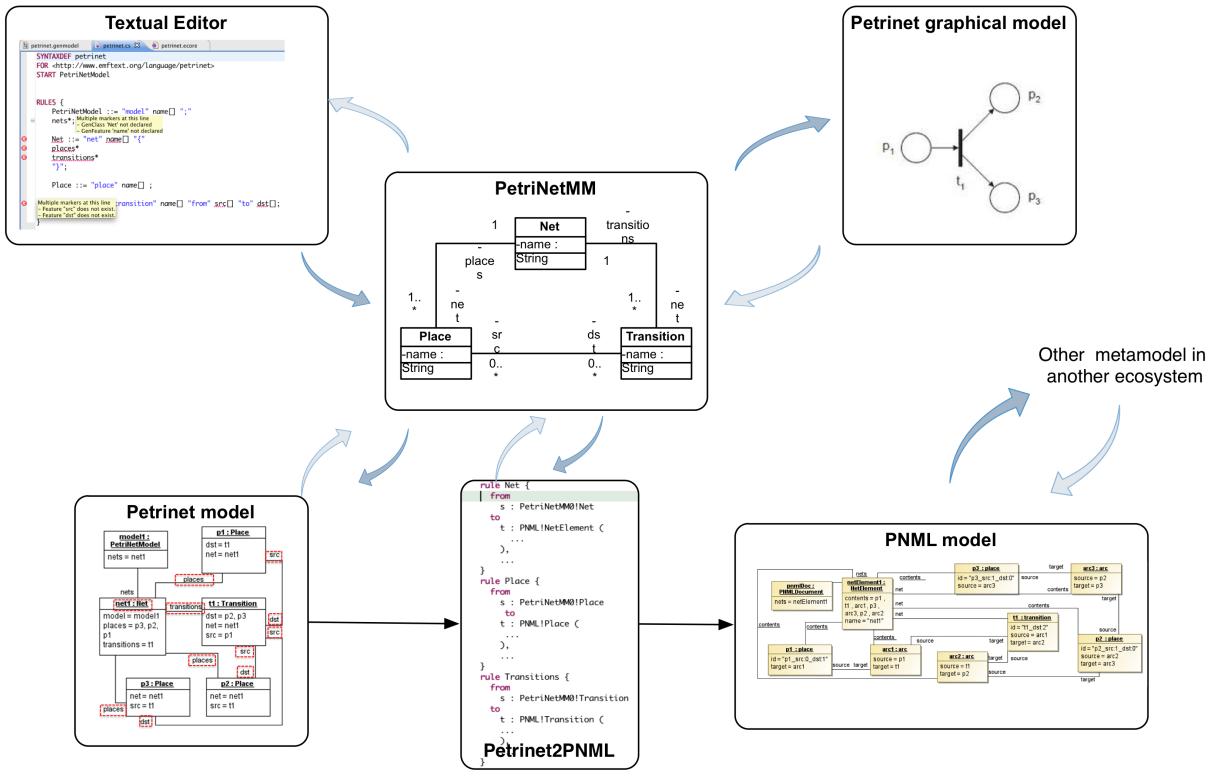


Figure 1: MDE ecosystems

evolve together with the ecosystem has been partially addressed and discussed in [3], while most of the approaches were dealing with the metamodel/model co-evolution (e.g. [4, 6, 7]) and only few work with metamodel/transformation and other artifact co-evolution (e.g., [8, 9]). To better understand the complexity of the problem let us consider the ecosystem represented in Figure 1. The *PetriNetMM* metamodel is given in the middle, PetriNet *models* can be translated by means of *transformations*, and finally different kinds of *editors* permit to enter and manipulate the models.

The metamodel changes can be classified according to the impact on the artifacts. In particular, there are changes that *a)* do not affect the artifacts at all and no adaptation is required; *b)* affect the artifact which can be automatically adapted; and finally *c)* affect the artifacts which cannot be automatically adapted. Furthermore, the adaptations which are required to recover modeling artifacts with respect to the changes executed on the corresponding metamodel, depend on the relation which couple the modeling artifact and the metamodel [2]. An example of metamodel evolution is shown in Figure 2, where the initial version of the PetriNet metamodel in Figure 2 (top) is modified by leading to the version shown at Figure 2 (bottom). Clearly, the validity and consistency of the ecosystem might be compromised because of the corruption of the relations between artifacts and the metamodel [10].

Therefore, it is of paramount importance to be able to trace and visualize the dependencies within the ecosystem in order to enable the modeler to detect those modifications that compromise existing artifacts and especially those that require more labour to be repaired. In the next section, a methodology is discussed to support the adaptation of those artifacts which have been affected by metamodel changes.

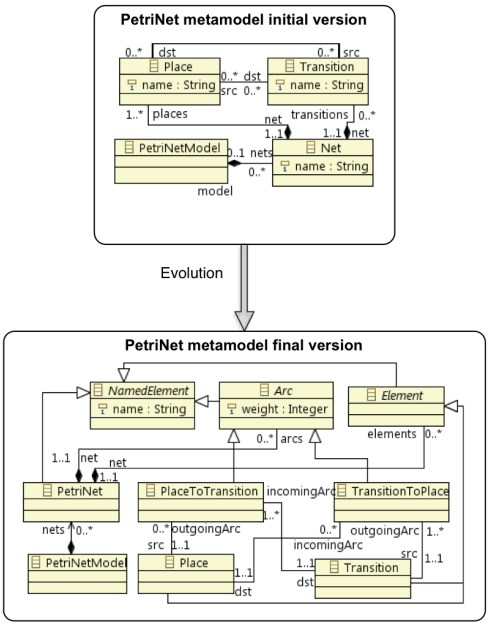


Figure 2: PetriNet metamodel evolution

3. COUPLED EVOLUTION MANAGEMENT PROCESS

If the adaptation is applied with spontaneous or individual skills can give place to inconsistencies between the metamodel and the related artifacts [4]. Therefore, a methodology which prevents the consequent information erosion and provides the modeler with a consistent guidance in the adaptation process can be significant. In Figure 3, a process consisting of a number of activities is presented. It encompasses the specification of the metamodel changes, the evaluation of their impact on the existing artifacts, the sustainability of the induced adaptations, and finally the actual migrations of the affected artifacts. This represents an enhancement of the methodology presented in [5], which did not consider the *change impact visualization* activity. In the remaining of the section, all the activities shown in the figure are discussed separately.

1. *Relation Definition*: this activity is performed only once per each kind of artifact to be adapted. For instance, in the case of ATL transformations [11], the ATL and the ECore metamodels are considered in order to establish correspondences between them. Such correspondences are used later in the process to automatically derive the dependencies between an evolving metamodel and the existing transformations. This activity can be done by using the work in [2] that exploits weaving models and megamodels to specify and manipulate correspondences among related and evolving artifacts. The upper side of Figure 4 shows a sample weaving model (as defined in [2]), which specifies the relation between the ECore metaclass `EClass` and the ATL metaclass `OclModelElement`.

2. *Dependencies Elicitation*: after the previous activity, the definition of the relations can be used to automatically derive the dependencies between the evolving metamodel and the linked artifacts. Such dependencies can be expressed as a weaving model like the one shown in the lower side of Figure 4. It shows the dependencies between the metaclasses of the `PetriNet` metamodel and the elements of a given ATL transformation having it as source metamodel. For instance, the first rule of the transformation shown on the right-hand side of Figure 4 contains an `OclModelElement` named `Net`, thus it is connected with the `EClass` element similarly named on the left-hand side of the figure. Such a dependency link specifies that changing the name of the `Net` metaclass in the `PetriNet` metamodel implies to propagate such a change to each `OclModelElement` linked to it.

3. *Metamodel Changes Specification*: the changes that the modeler wants to apply on a given metamodel should be properly represented in order to enable automatic manipulations and automate subsequent phases of the process. For instance, in this phase it is possible to adopt the metamodel independent approach to difference representation proposed in [12] already used to deal with other coupled evolution problems (e.g., adaptation of models [7], and GMF editors [9]), but other approaches can be used as well.

4. *Change Impact Visualization*: in this phase the elements of the artifact, which has been affected by the specified metamodel changes are graphically shown. In this way, modelers can have a preliminary assessment of the impact that the metamodel changes being applied, will have on the existing artifacts. They can decide to amend such changes, or to go ahead in the process with a more accurate evaluation of the required adaptation cost [5].

5. *Change Impact Analysis / Evaluation Cost*: in this activity the evaluation performed in the previous step is enhanced by considering an appropriate cost function related to the actions, which are required to adapt the affected artifacts [5]. In the specific case of ATL model transformations, according to the dependencies previously elicited, all the transformation elements which are in relation with the changed metamodel elements are identified and used as input for evaluating the adaptation cost. In particular, by considering the affected elements modelers evaluate the cost for adapting the affected transformations. In this respect, if the adaptation is too expensive (according to an established threshold) modelers can decide to refine the metamodel changes to reduce the corresponding costs, otherwise they can accept the applied metamodel changes. The evaluation is based on an adaptation cost function defined separately [5].

6. *Metamodel Changes Commit*: once the metamodel changes have been evaluated, modelers can commit them in order to concretely apply the previously evaluated artifact adaptations.

7. *Artifact Adaptation*: in this activity the existing artifacts which have been affected by the committed metamodel changes are adapted. Proper tools are required to support this step. Over the last years different approaches have been proposed to support the coupled evolution of metamodels and related artifacts. EMFMigrate [13] is one of those tools that presents the advantage of dealing with the diversity of artifacts with a unique notation.

In the next section we focus on the *Change Impact Visualization* activity, since the detection phase [2] and the migration activity [5] have been already treated. We propose a tool chain able to generate a graphical representation of the change impact out of a model representing the differences between two subsequent versions of the same metamodel.

4. CHANGE IMPACT VISUALIZATION

Over the last years, different attempts have been proposed to deal with the problem of supporting the graphical representation of inter-model relations. For instance, in some cases there is the need for representing different viewpoints of a system, or multiple views of it, each capturing a concern of interest for a particular stakeholder. In this direction, the TraceVis tool [14] has been proposed to graphically represent traceability information between structured data. Interestingly, TraceVis has been adopted to visualize traceability links between the source and target models of an executed model transformation [15].

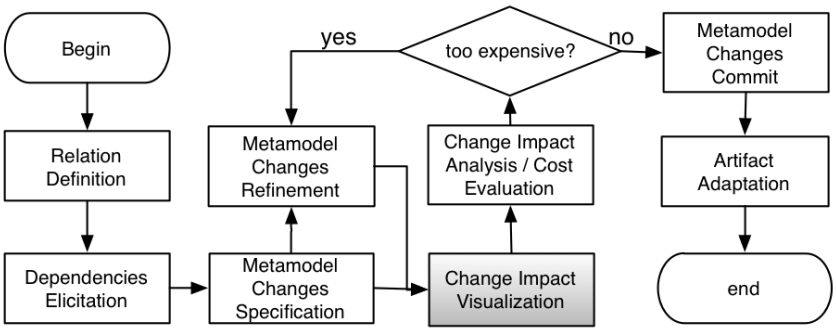


Figure 3: Methodology activities

In this section, we propose a tool chain that aims at using the TraceVis tool to graphically represent the impact that the changes being addressed on a given metamodel can have on existing artifacts. Figure 5 depicts a screenshot of the TraceVis tool at work while representing the dependencies (generated by the tool chain explained later in the section) among three different models. In particular, the part of the figure labelled ② represents the initial version of the PetriNet metamodel shown in Figure 2.a. The part labelled ① represents the difference model (also called delta model hereafter) encoding the differences to be applied on the initial version of the metamodel in order to obtain that in Figure 2.b. The part labelled ③ represents an existing ATL transformation, which has ② as source metamodel, and is affected by the metamodel changes specified in ①. The lines between ① and ② represents the relations between the differences represented in ① and the metamodel elements in ② that are changed. For instance, the `ChangedClass` modification represented in ① is related to the class `Net` because of the renaming operation that has to be performed in order to obtain the new version of the mataclass named as `PetriNet`. The links between ② and ③ represent the impact that changing the `Net` metaclass has on the existing ATL transformation *PetriNet2PNML*[2]. For instance, the `OCLModelElement Net` in the source input pattern of the transformation is linked to the evolving metaclass and indicates a possible issue in case of changing that metaclass `Net`. Of course the package containing the changed class is involved too (see the connection between the package `PetriNet` and the `OCLModel PetrinetMM0` in the transformation). To better understand the connection between ② and ③, inexpert ATL developers can consider Figure 6 that explicitly represents the considered ATL transformation (see the left-hand side of the figure) and its representation in the TraceVis tool (see the right-hand side of the figure). For instance, the input and output models of the transformation header (see `OUT` and `IN` of the second line of the transformation) are the last two elements in the TraceVis representation.

In the remaining of the section we present the proposed tool chain (see Section 4.1) able to generate TraceVis specifications like the one in Figure 5, starting from a dependency model like the one in the lower-hand side of Figure 4, and a difference model representing the metamodel changes being applied. The realization of the tool chain required the implementation of model-to-model, and model-to-code transformations, which are detailed in Section 4.2, and Section 4.3, respectively.

4.1 Proposed tool chain

In order to support the TraceVis visualization of change impacts, we have implemented a tool chain able to generate artefacts that TraceVis is able to open. In particular, TraceVis is able to manage XML documents like the one in Listing 1, which is a fragment of the XML specification of the visualization depicted in Figure 5. Essentially, the document consists of stages, each represent one of the related models. For instance, Listing 1 contains 3 stages (see lines 4, 23, 34) corresponding to the difference model, the PetriNet metamodel, and the affected ATL transformation shown in Figure 5, respectively. Each stage contains the specification of nodes, each representing one of the elements related to one or more elements contained in another model. For instance, the node `75` in line `5` of Listing 1 represents the element of the difference model encoding the metaclass renaming. The attributes that can be specified for each node include the name, the parent node, and additional custom attributes that might be useful for automatic manipulations. After the specification of stages, and consequently the contained nodes, inter-model relations are given (see lines 54-72 in Listing 1).

Listing 1: Fragment of the TraceVis XML file of visualization in Figure 5

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2  <data>
3   <stages>
4     <stage id="1" name="delta1">
5      <node id="75">
6       <attribute name="parent" type="">
7        <modification>76</modification>
8       </attribute>
9       <attribute name="name" type="string">
10       <modification>DELTA_ChangedeClass_Net</
             modification>
11      </attribute>
12     </node>
13     <node id="76">
14      <attribute name="parent" type="">
15       <modification>5</modification>
16      </attribute>
17      <attribute name="name" type="string">
```

[2]This transformation is available from `http://www.eclipse.org/atl/atlTransformations/Grafcet2PetriNet/ExampleGrafcet2PetriNet\[v00.01\].pdf`

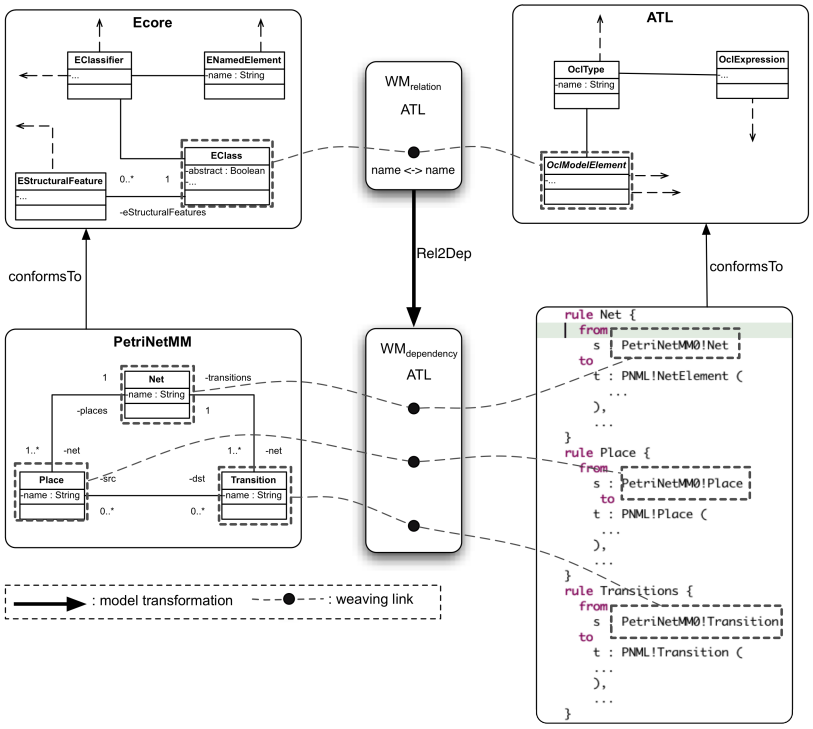


Figure 4: Relation Definition and Dependencies Elicitation

```
      <modification>DELTA_ChangedePackage_petrinet</
          modification>
     </attribute>
    </node>
    ...
   </stage>
   <stage id="2" name="petrinet">
    <node id="23">
     <attribute name="parent" type="">
      <modification>26</modification>
     </attribute>
     <attribute name="name" type="string">
      <modification>MM_eClass_Net</modification>
     </attribute>
    </node>
    ...
   </stage>
   <stage id="3" name="petrinet2GM-ATL-0.2">
    <node id="62">
     <attribute name="parent" type="">
      <modification>37</modification>
     </attribute>
     <attribute name="name" type="string">
      <modification>ATLATL_OCL_MODEL_ELEMENT_Net</
          modification>
     </attribute>
    </node>
    <node id="63">
     <attribute name="parent" type="">
      <modification>71</modification>
     </attribute>
     <attribute name="name" type="string">
      <modification>ATLATL_OCL_MODEL_ELEMENT_Net</
          modification>
     </attribute>
    </node>
    ...
   </stage>
  </stages>
  <relations>
   <relation type="Impact_On">
     <source stage="1">75</source>
     <target stage="2">23</target>
   </relation>
   <relation type="Impact_On">
     <source stage="1">76</source>
     <target stage="2">26</target>
   </relation>
   <relation type="Impact_On">
    <source stage="2">23</source>
    <target stage="3">62</target>
   </relation>
   <relation type="Impact_On">
    <source stage="2">23</source>
    <target stage="3">63</target>
   </relation>
   ...
  </relations>
</data>
```

As shown in Figure 7 the generation of TraceVis XML specifications is performed in different steps. In particular, given two subsequent versions of the same metamodel `MM`, and `MM'`, their differences are calculated by means of the EMFCompare [16]. The outcome of the comparison is the input of the model transformation `EMFCompare2EcoreDiff` [17] able to generate a difference model according to the difference representation approach proposed in [12]. Such a difference model, together with the initial metamodel `MM`, the existing `Artifact` which depends on it, and the `DependencyModel` (defined in the *Dependencies Elicitation* step in Figure 3), are taken as input by the model-to-model transformation `TraceVisGen`. The generated `TraceVisModel` is transformed by means of the model-to-code transformation `TraceVisMM2XML` able to generate a corresponding XML document like the one in Listing 1.

In the next section the `TraceVisGen` transformation is presented, whereas the model-to-code transformation `TraceVisMM2XML` is discussed in Section 4.3.

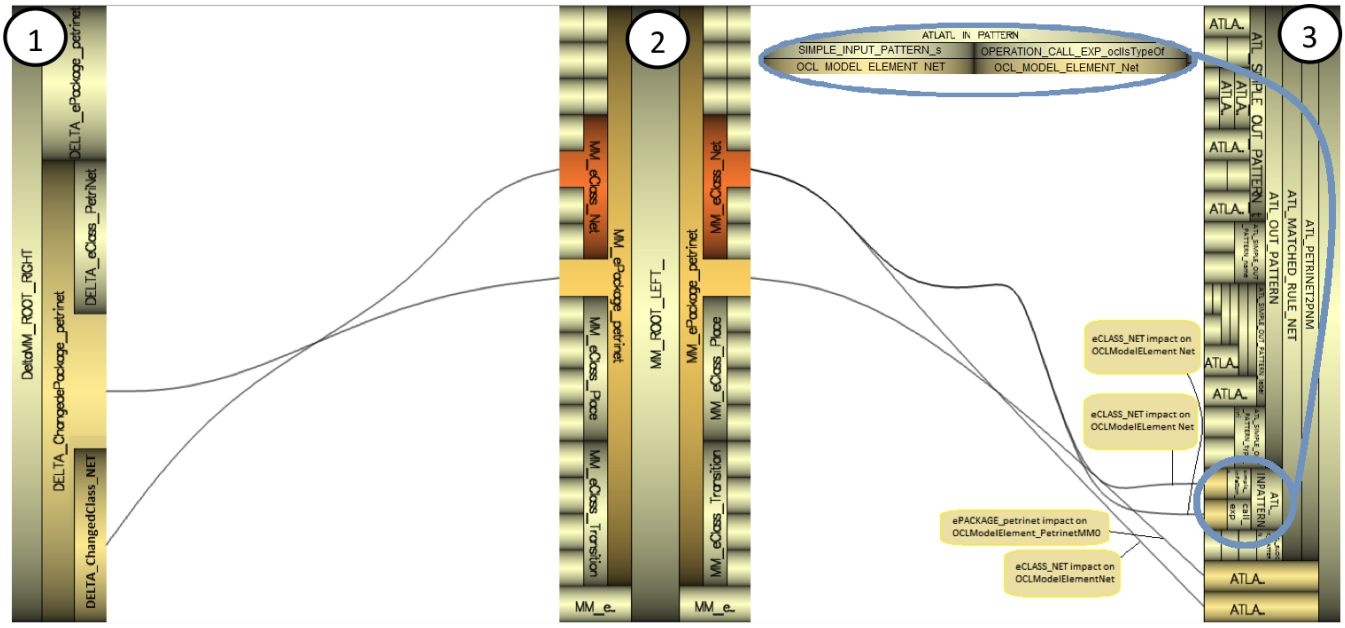


Figure 5: Metaclass renaming visualization

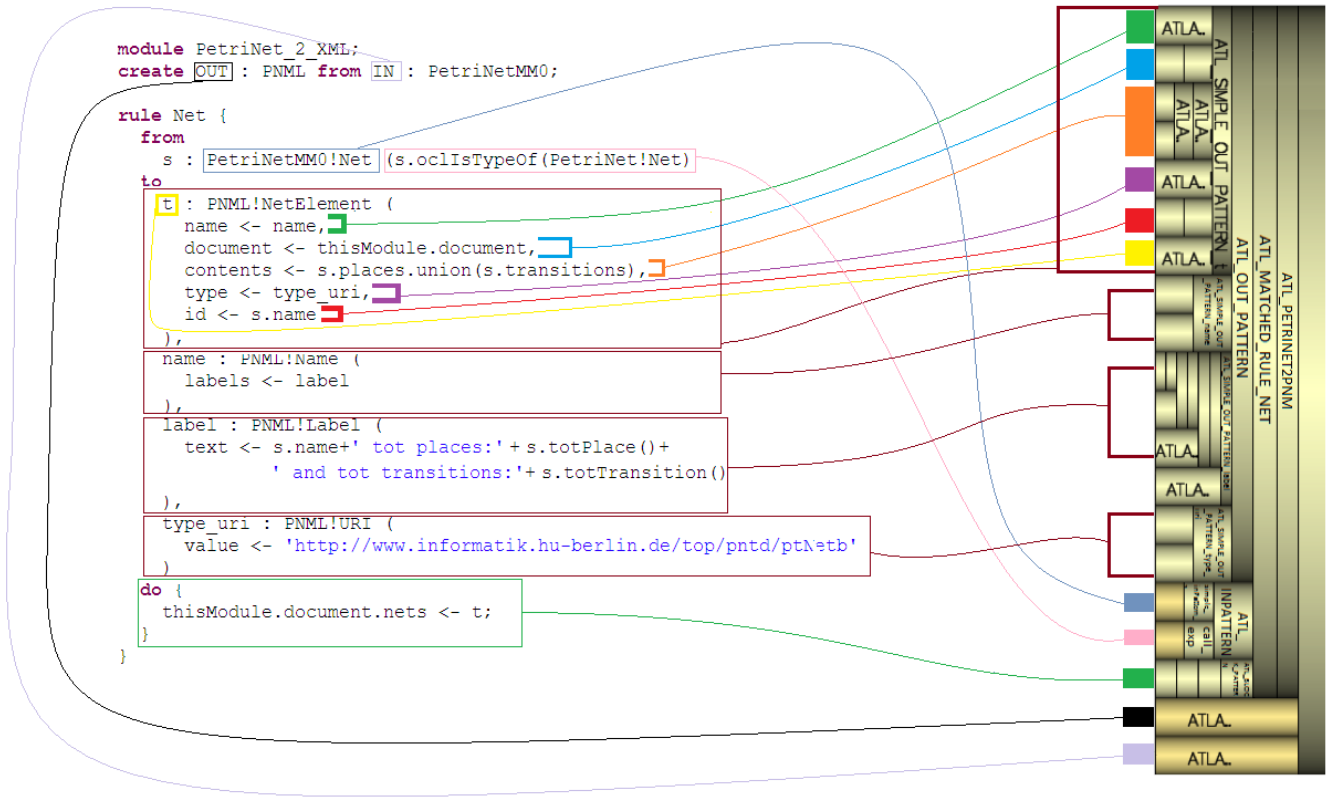


Figure 6: ATL transformation and its representation in TraceVis

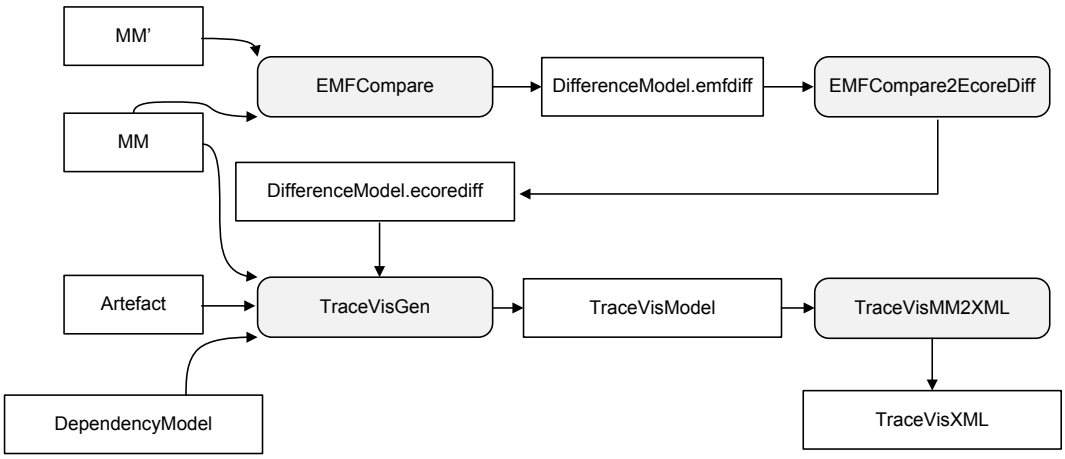


Figure 7: Generation of TraceVis XML specifications

4.2 From dependency models to TraceVis specifications

Splitting the generation of the final XML artifact in two steps resulted to be more simple than having only one model-to-text transformation. In fact, the semantic gap between the two levels of abstraction is too large, and this motivated splitting the generation process in two steps. Thus, the TraceVis metamodel in Figure 8 has been defined in order to enable the generation of intermediate TraceVis models as shown in Figure 7. The generation of such models is performed by means of the ATL transformation shown in Listing 2.

The generation of `Node`, `Attribute` and `Modification` elements is performed by the rule `NodeClass` in lines 4-24. The *conditional* statement in lines 17-21 contains the invocation of `refImmediateComposite` to retrieve the container of an element, and then makes use of the `resolveTemp` function to get the target model that will be generated by an ATL rule combined with a given source model element.

Listing 2: Fragment of the TraceVisGen transformation

```
1 module Adapter;
2 create OUT : TRACEVIS from wmdep : WeavingModel, MMD
      : ECORE, atl : MM;
3
4 rule NodeClass {
5  from
6   s : ECORE!EClass
7  to
8   node: TRACEVIS!Node(
9    id <- thisModule.ID,
10   attribute <- Sequence{attr,attrName}
11  ),
12  attr: TRACEVIS!Attribute(
13   modification <- modi,
14   name <- 'parent'
15  ),
16  modi: TRACEVIS!Modification(
17   parent <- if(s.refImmediateComposite()=
         OclUndefined)
18       then thisModule.leftRoot
19       else
20       thisModule.resolveTemp(s.refImmediateComposite
           (),'node')
21       endif
22  ),
23  ...
24 }
25 ...
26 rule NodeChangedClassDiff {
27  from
28   s : EcoreDiff!ChangedEClass
29  to
30
31  --Relation from inter-model reference
       ApplicationElement
32   rel : TRACEVIS!Relation(
33    source <- sour,
34    target <- targ
35   ),
36   sour: TRACEVIS!Source(
37    node <- thisModule.resolveTemp(s,'node'),
38    stage <- thisModule.MarkstageDelta
39   ),
40   targ: TRACEVIS!Target(
41     node <- thisModule.resolveTemp(s.
          applicationElement,'node'),
42     stage <- thisModule.MarkstageL
43   )
44 }
45 ...
46 rule RelationLink{
47  from
48   s : WeavingModel!Link
49  to
50   node : TRACEVIS!Relation(
51    source <- sour,
52    target <- targ
53   ),
54   sour: TRACEVIS!Source(
55    node <- s.left.getReferredElement(),
```

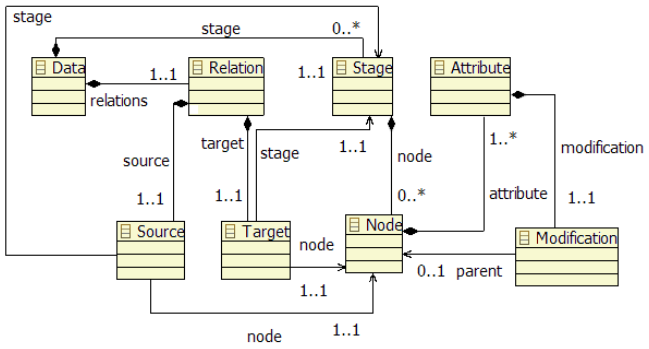


Figure 8: TraceVis Metamodel

```
    stage <- thisModule.MarkstageL
   ),
   targ: TRACEVIS!Target(
    node <- s.right.getReferredElement(),
    stage <- thisModule.MarkstageR
   )
 }
...
```

Target `Relation` elements are generated by the rules `NodeChangedClassDiff` and `RelationLink`. In particular, the former generates the link between the source difference model and the evolving metamodel. The latter generates the link between the evolving metamodel and the affected artefacts.

It is important to note that our approach is generic and can be used for supporting the *Change Impact Visualization* of any affected artefacts involved in the ecosystem. In fact, by observing the structure of `TraceVisGen` it is possible to identify a recurring pattern (shown in Listing 3) for each metaclass *MC* of the artefact metamodel *MM*. Thus, it is possible to generate the `TraceVisGen` transformation for each artefact using a higher-order transformation (HOT[3]) which takes the artifacts metamodel and generates the appropriate `TraceVisGen` transformation.

Listing 3: Recurring pattern in the artifact element node

```
rule NodeModule{
 from
  s : MM!MC
 to
  node : TRACEVIS!Node(...),
         ...
}
```

4.3 Generation of the TraceVis XML specifications

According to Figure 7, the `TraceVisModel` generated by means of the ATL transformation presented in the previous section is the input of the `TraceVisMM2XML` model-to-text transformation. Such a transformation is implemented by means of the Acceleo[4] code generator. Acceleo is a template-based approach for generating text from models. Acceleo templates identify repetitive and static parts of text, and embody specific queries on the source models to fill the dynamic parts.

Listing 4: TraceVisMM2XML transformation

```
...
[module generate('http://www.di.unavaq.org/Tracevis')
    /]
[template public generate(aData : Data)]
 [comment @main /]
 [file ('export.xml', false, 'UTF-8')]
 <?xml version="1.0" encoding="ISO-8859-1"?>
 <data>
  <stages>
   [for (s:Stage | aData.stages)]
    <stage id="[s.id/]" name="[s.name/]">
     [for (n:Node | s.node)]
     <node id="[n.id/]" inserttime="[n.inserttime/]">
      ...
     </node>
     [/for]
    </stage>
   [/for]
  </stages>
  <relations>
   [for (r : Relation | aData.relations)]
   <relation inserttime="[r.inserttime/]" type="
       Impact_On">
    ...
   </relation>
   [/for]
  </relations>
 </data>
 [/file]
[/template]
```

This Acceleo template in Listing 4 generates an XML file called *export.xml*. In particular, (i) for each stage in the source `TraceVisModel` it generates a `<stage>` tag, (ii) for each source instance of the metaclass `Node` it creates a `<node>` tag, and (iii) for each source instance of the metaclass `Relation` a target `<relation>` tag is generated.

5. DISCUSSION

By using the change impact visualization presented above, different recurrent patterns have been identified. In particular, the required changes in the affected artifacts are not always directly proportional to the changes in the metamodel as discussed in the following.

Uniform impact The example in Figure 9 is a small set of evolution steps where the delta model contains few connections with the evolving metamodel and there is a direct correspondence with the impact between metamodel and artifact. This situation is recurrent when a metamodel evolves by means of few atomic changes. In this case the mapping denotes one change in the metamodel correspondes to one change in the artifact, that is not strictly a 1-1 relation but the number of connections is similar in the two stages. This situation is strongly related to the artifact nature: e.g., changing a metaclass in the metamodel corresponds to a corruption point in a transformation, in particular in the rule where the metaclass in matched as input or output. On the contrary, considering a model as artifact the correspondence is different because likely the instances of the metaclass will be frequently instantiated in the model (see Figure 11).

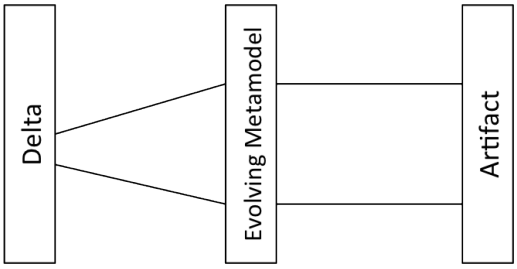


Figure 9: Uniform impact

Heavy metamodel evolution - Light impact on the artifact The situation shown in Figure 10 contains a lot of changes in the metamodel and less impact on the artifact. This situation is not very frequent but also not rare, according to our experience. This case implies that the involved elements in the metamodel are not referred in the artifact. For instance, in case of an extract superclass change[5] in the

[3]A HOT transformation is a special kind of transformation which has other transformations as input and/or output

[4]http://www.eclipse.org/acceleo/

[5]`http://www.metamodelrefactoring.org/`

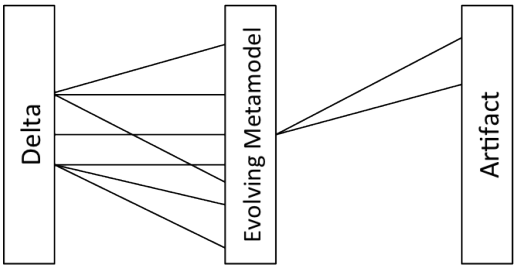


Figure 10: Heavy metamodel evolution - Light impact on the artifact

metamodel, all the attributes in the subclasses have to be removed, and a new one in the added super class has to be added. If the existing artefact is a model transformation, then we will have a light impact because the constructs relating to the extracted attribute will not be corrupted and still matched. Same situation in case of models where the the class instances continue to be valid because of the inheritance relation.

Light metamodel evolution - Heavy impact on the artifact The example in Figure 11 depicts a strong impact on the metamodel coming from a light metamodel evolution. A typical application of this category can be found in changing or renaming an attribute in a superclass. In case of ATL

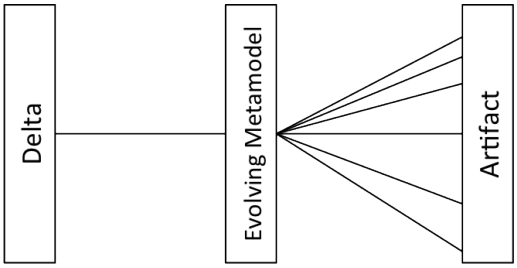


Figure 11: Light metamodel evolution - Heavy impact on the artifact

transformations, the renamed attribute can be duplicated in all the rules involving the subclasses and this leads to a strong impact for the duplicated information that has to be fixed. Removing an attribute from a metaclass can fall in the same category when this attribute is used many times in the artifact (e.g., in the case of helpers and rules in ATL transformations).

Heavy metamodel evolution - Heavy impact on the artifact The example in Figure 12 represents a very heavy impact on the metamodel coming from a heavy metamodel evolution. Such a situation is not common in literature, and represents that the metamodel has been heavily changed and consequently the existing artifact has been strongly corrupted. This example would be diffused if the metamodel evolution activity was not a step by step evolution.

For example this case would be produced if OMG were passing from UML 1.0 directly to UML 2.0 avoiding the intermediate versions. Since the metamodel definition is a cyclical activity where the metamodeler starts by defining concepts and refining the corresponding metamodel definition, this case is related to a well defined metamodel that undergoes strong changes coming from a turn upside down

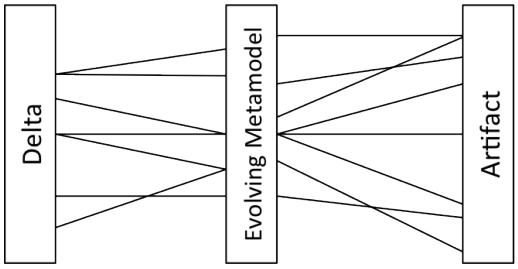


Figure 12: Heavy metamodel evolution - Heavy impact on the artifact

of the concepts in the application domain.

Non breaking changes with no impact The example in Figure 13 represents a set of non breaking metamodel changes that do not impact the existing artifact. We can imagine to transform an abstract metaclass into a real one. In this case, existing transformation rules are not affected. Of course this case can be considered as the best case that

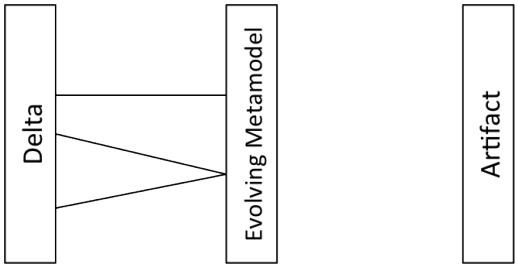


Figure 13: Metamodel non-breaking changes and no impact on the artifact

a metamodeler can see during the evolution phase, however this is a very rare case.

6. RELATED WORKS

Much research has been conducted on traceability, typically focused on software engineering. However the vast majority of the research is focused on either trace retrieval [18] or trace management [19] and usually neglects the visualization part. Trace retrieval is an activity that uses data mining techniques to generate the relations between artifacts created during the software engineering process. By analyzing keywords, relations can be defined between different parts of the artifact. Trace management focuses on maintaining traces during the software development process. In general, this is done by the authors or the maintainers of the artifacts. Results of trace retrieval or trace management are typically visualized using common visualization methods. According to Wieringa [20], visualization approaches can be categorized in matrices, cross-references and graph-based representations. Beyond these three types of visualization, there are only a few approaches to visualize traces in a different way.

The commercial application *TBreq* is focused on software engineering. The artifacts are listed horizontally and if a relation is defined between items of two artifacts an edge is

drawn. TBreq does not use the hierarchical structure of artifacts to reduce the size of the model visualization nor does it use any edge aggregation techniques or edge bundling to reduce cluttering. Although this works for small examples, large examples quickly give visual clutter. Moreover, due to the fact that individual items of an artifact are displayed as a list, only a limited number of items per artifact can be displayed. Nevertheless *TBreq* does convey traceability information since it gives a (partial) overview and can be used to quickly follow traces within the data. This tool can also show relations between two non-consecutive artifacts.

Pilgrim et. al. [21] use a three dimensional approach to gain insight in transformation chains containing different UML diagrams. Artifacts are all UML diagrams and are projected on a plane. Relations are visualized using edges between consecutive planes. Using a three-dimensional approach has little benefits over a two dimensional approach and still gives visual clutter when models become too large and too many relations are defined between planes. Moreover, using the third dimension will lead to occlusion problems when projecting the image to screen.

Marcus et al. [22] propose a visualization method that only shows information considered to be important to developers working on a part of a software project. The visualization method is similar to cross reference based methods and shows the various parts of an artefact as a rectangle, and related parts are assigned a similar colour. It is used to show relations of multiple artefacts, however it can only visualize a small number of dependencies and therefore cannot provide a global overview. Moreover, traces can only recursively be followed.

Briand et al. in [23] propose an approach to analyze the impact of the changes on the design model before applying the changes to the implementation model. The authors provided a classification of change types for three UML diagrams: class, sequence diagram and state machine diagrams. For each change type, an impact analysis rule is specified, describing how to extract the list of elements that are impacted by that particular change type. The definitions of change types and impact analysis rules are expressed in the Object Constraint Language (OCL). The propagation of changes to indirectly related entities is controlled by a distance measure, which is used to either stop the change propagation or to prioritize impact paths according to their depth. A prototype tool (iACMTool) was developed to automate and evaluate the feasibility of such an approach.

Fourneret and Bouquet [24] present an approach to perform model-based impact analysis for UML Statechart diagrams using dependence algorithms. The rationale of this type of analysis is to consider dependencies between transitions instead of states. In such context, two transitions are said to be data dependent if one transition defines a value of a variable that can be potentially used by the other transition and they are said to be control dependent if one transition may affect the traversal of the other transition. Based on this concept, the authors identified the data and control dependencies for UML statecharts and formulated the corresponding algorithms to be used in computing the dependence graphs of the statecharts elements. Accordingly, they identified and classified the possible changes to UML statecharts such as adding new transitions, deleting, and modifying existing transitions. It is worth to note that adding or deleting a transition can impact both the data and control dependence graphs while modifying a transition can impact only the data dependence graph. By having the two versions of statecharts (the original and the modified one) and their computed dependence graphs, the authors illustrated how their proposed dependence algorithms are used to extract the impacted elements. Our approach is different starting from the fact that is artifact-independent; in the cited works all those approaches are related to specific artifacts or particular set of models. The editor is a two dimensions editor that offers an advanced set of features like zooming and selection. Moreover the important characteristic is that for the first time we apply those features to the impact visualization in MDE ecosystems. Indeed given the independence characteristic of the tool chain, it is possible to replace the final transformation in order to provide the compatibility with other visualization tools.

This work is in part related also to coupled evolution process and the techniques and methodology for coupled evolution have been previously investigated in [25]. Coupled evolution is related to adapt models in response to metamodel changes [4, 26, 7, 6] and only recently, the problem of metamodel evolution/transformation co-adaptation has started to be investigated. Few attempts have been provided to deal with it in a dedicated way [8, 27, 28]. The methodology proposed in [5] has been extended and in part completed in this work adding the visualization activity as a new feature available in the methodology.

Evolution of modelling languages is also treated in [29], where the authors provide a taxonomy for evolution and discuss the various evolution cases for different kinds of modeling artifacts. The methodology discussed in this paper proposes to decompose the migration actions into primitive scenarios that can be resolved by the framework.

7. CONCLUSIONS

The problem of adapting the diversity of artifacts composing a metamodeling ecosystem when the metamodel undergoes modifications is intrinsically difficult. Trying to repair models, transformations, and editors without the guidance of a methodology can introduce inconsistencies and lead the ecosystem into an ungovernable status. This paper presented a generic approach to change impact visualization intended as a technique for visualizing the dependencies between Ecore metamodels and related artifacts in terms of traceability information. The modifications in a metamodel sometimes are not sustainable and being able to assess this aspect in advance can spare the modeler with unmanageable difficulties because of the intricacy of the adaptations. Compared with the work in [5] where a cost function is proposed, the presented technique conveys to the modeler a more intuitive notion of *intensity* and *locality* (see Sect. 5): the visualization permits to informally estimate how many adaptations are required in a restricted portion of an artifacts and especially the multiplicity of the impact on an artifact of a given metamodel change. Another contribution of the paper is the formalization of the TraceVis metamodel which permits to bridge Ecore models with this specific visualization tool.

In the near future, we are interested in the *live* monitoring of the change impact, i.e., as soon as a change is made on the metamodel information about the impact on related

artifacts in the ecosystem are automatically displayed in a context view. Despite the simplicity of the idea, its realization requires an operation recorder for registering the model differences in the metamodel, the incremental generation of the dependencies between the elements of the metamodel and of the related artifacts, and finally a proper presentation of impact analysis. The visualization can be enriched with the categorization of the refactorings in order to provide a more complete idea of the impact on the artifact. Representing this categorization with different colors can be useful to distinguish different forms of impact.

8. REFERENCES

[1] Schmidt, D.C.: Guest Editor's Introduction: Model-Driven Engineering. Computer **39**(2) (2006) 25–31

[2] Iovino, L., Pierantonio, A., Malavolta, I.: On the impact significance of metamodel evolution in mde. Journal of Object Technology **11**(3) (October 2012) 3:1–33

[3] Di Ruscio, D., Iovino, L., Pierantonio, A.: Evolutionary togetherness: How to manage coupled evolution in metamodeling ecosystems. In Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: ICGT. Volume 7562 of Lecture Notes in Computer Science., Springer (2012) 20–37

[4] Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In Ernst, E., ed.: Proceedings of the 21st ECOOP. Volume 4069 of LNCS., Springer-Verlag (July 2007)

[5] Di Ruscio, D., Iovino, L., Pierantonio, A.: A methodological approach for the coupled evolution of metamodels and ATL transformation. In: Proc. 6th International Conference on Model Transformation (ICMT'13). (2013)

[6] Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - Automating Coupled Evolution of Metamodels and Models. (2009) 52–76

[7] Cicchetti, A., Di Ruscio, D., Eramo, R., Pierantonio, A.: Automating Co-evolution in Model-Driven Engineering. In: 12th IEEE International EDOC Conference (EDOC 2008), Munich, Germany, IEEE Computer Society (2008) 222–231

[8] Levendovszky, T., Balasubramanian, D., Narayanan, A., Karsai, G.: A Novel Approach to Semi-Automated Evolution of DSML Model Transformation. In: Second International Conference on Software Language Engineering, SLE 2009, LNCS. Volume 5969., Denver, CO, Springer, Springer (05/2010 2010)

[9] Di Ruscio, D., Laemmel, R., Pierantonio, A.: Automated co-evolution of GMF editor models. In Malloy, B., Staab, S., van den Brand, M., eds.: 3rd International Conference on Software Language Engineering (SLE 2010). Number 6563 in LNCS, Springer, Heidelberg (October 2010) 143–162

[10] Di Ruscio, D., Iovino, L., Pierantonio, A.: What is needed for managing co-evolution in MDE? In: Proceedings of the 2nd International Workshop on Model Comparison in Practice, ACM (2011) 30–38

[11] Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: Atl: A model transformation tool. Science of Computer Programming **72**(1-2) (2008) 31–39

[12] Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A Metamodel Independent Approach to Difference Representation. Journal of Object Technology **6**(9) (October 2007) 165–185

[13] Di Ruscio, D., Iovino, L., Pierantonio, A.: Evolutionary togetherness: how to manage coupled evolution in metamodeling ecosystems. In: Intl. Conf. on Graph Transformations (ICGT 2012). Volume 7562 of LNCS., Springer (2012)

[14] van Ravensteijn, W.: Visual traceability across dynamic ordered hierarchies. Master's thesis, Eindhoven Univ. of Technology, The Netherlands (2011)

[15] Amstel, M., Brand, M., Serebrenik, A.: Traceability visualization in model transformations with tracevis. In Hu, Z., Lara, J., eds.: Theory and Practice of Model Transformations. Volume 7307 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 152–159

[16] Brun, C., Pierantonio, A.: Model Differences in the Eclipse Modeling Framework. UPGRADE, The European Journal for the Informatics Professional **9**(2) (April 2008)

[17] Pierantonio, A., Iovino, L., Di Rocco, J.: Bridging state-based differencing and co-evolution. In: Models and Evolution Workshop - 15th International Conference on Model Driven Engineering Languages and Systems. (September 2012)

[18] Lucia, A.D., Fasano, F., Oliveto, R., Tortora, G.: Recovering traceability links in software artifact management systems using information retrieval methods. ACM Trans. Softw. Eng. Methodol. **16**(4) (September 2007)

[19] Lozano, A., Pinter, R.Y., Rokhlenko, O., Valiente, G., Ziv-ukelson, M.: Seeded tree alignment and planar tanglegram layout

[20] Wieringa, R.: An introduction to requirements traceability (September 1995) Acknowledgements: This report benefited from input given by Eric Dubois, Anthony Finkelstein and Hanna Luden.

[21] Pilgrim, J., Vanhooff, B., Schulz-Gerlach, I., Berbers, Y.: Constructing and visualizing transformation chains. In Schieferdecker, I., Hartman, A., eds.: Model Driven Architecture âĂŞ Foundations and Applications. Volume 5095 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2008) 17–32

[22] Marcus, A., Xie, X., Poshyvanyk, D.: When and how to visualize traceability links? In: Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering. TEFSE '05, New York, NY, USA, ACM (2005) 56–61

[23] Briand, L.C., Labiche, Y., O'Sullivan, L., Sówka, M.M.: Automated impact analysis of uml models. J. Syst. Softw. **79**(3) (March 2006) 339–352

[24] Fourneret, E., Bouquet, F.: Impact analysis for uml/ocl statechart diagrams based on dependence

algorithms for evolving critical software. Laboratoire dâĂŹInformatique de Franche-Comté, Besançon, France, Tech. Rep. RT2010-06 (2010)
[25] Favre, J.M.: Meta-Model and Model Co-evolution within the 3D Software Space. In: Procs. of the Int. Workshop ELISA at ICSM. (September 2003)
[26] Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model Migration with Epsilon Flock. In: ICMT. (2010) 184–198
[27] García, J., Díaz, O., Azanza, M.: Model transformation co-evolution: A semi-automatic approach. In: SLE. (2012) 144–163
[28] Méndez, D., Etien, A., Muller, A., Casallas, R.: Transformation migration after metamodel evolution. In: International Workshop on Models and Evolution (Me'10) - ACM/IEEE MODELS'2010. (2010)
[29] Meyers, B., Vangheluwe, H.: A framework for evolution of modelling languages. Science of Computer Programming **76**(12) (2011) 1223 – 1246